UNIVERSITÉ Concordia UNIVERSITY

Pandemic! Board Game

FINAL Build

Philip Michael (40004861)
Amanda Friesen (27596189)
Johnny Mak (40002140)
Jason Kalec (40009464)

FINAL PROJECT REPORT

Group: Ants

Department of
Engineering & Computer Science

Presented in partial fulfillment of the requirements for COMP345
Professor: Nora Houari

Advanced Programming Design in C++

April 2017

Table of Contents

1. Project Summary

This final build is a continuation of the previous Build #1 with several functionalities now operational as well as additional technical improvements and the addition of Design Patterns. The goal was to create a fully functional and playable version of Pandemic to the best of our abilities. As such, there are some limitations, but essentially the game is perfectly playable and as fair (with regards to winning and losing) as the classical original.

The game is text-based on the console with very nice output and easy-to-follow on-screen prompts and instructions. All necessary information is presented clearly and concisely to all players. Our game supports a minimum of 2 players and a maximum of 4. Turn functionality is so that player 1 starts, then 2, and so forth. To follow proper board game procedure, please have player 2 sit to the left of player, and the same for players 3 and 4 (to the left of the previous turn player).

In the following sections, we will be outlining the game setup and game dynamics. Thereafter we will be going over the programmed modules and design decisions (including the chosen and implemented Design Patterns), presenting you with a brief rundown of any libraries we used, outlining how we do exception handling, present UML diagrams for ease of use, refer to our instruction manual (which is a separate document joint to this one), and going over any limitations (regarding what is and is not working and how to approach them in the future) and a conclusion to tie it all together.

2. Essential Game Components

2.1 Game Setup

The setup for the game is very straightforward. In fact, not much has really changed from the previous build with regards to how the game is setup initially for play. A bit more flexibility was added, however. In fact, upon launching the game you will be greeted by our welcome splash screen. There, we ask if you'd like to start a new game or load a previously saved game. If you chose to start a new game, you will be prompted to enter a number of players from 2 to 4. Then, a difficulty setting is requested from the user. 1 is for beginners, 2 is for returning players that aren't that experienced, and 3 is for experts of the game. If you chose to load a saved game, you will pick up from where you left off (or rather at the most recent point after the last turn rotation completed).

2.2 Game Dynamics

As aforementioned, gameplay for 2 to 4 players is supported. The turns are performed in order, i.e. player 1 goes first, then 2, then 3, and finally 4. In a realistic setting, player 2 would be sitting to the left of player 1, and so on. Turns are easily identifiable with prompts confirming the player turn change. In addition, all necessary information is readily available to each player as we consistently update our views with data based on previous turn actions, epidemics (if any), and infections. Roles for each player are functioning as intended, and paired with event cards they guarantee a fun gaming experience. Each player's role and event card information (if available) is presented to them clearly in the view. Prompts to play event cards (if any) are given frequently and at correct intervals (except for *Resilient Population*, more on that in later sections).

Ready to save humanity?

3. Core Game Modules

In this section, we will be outlining every essential core module and their functionalities. This means that for every .cpp file, the appropriate .h file exists. As such, we will only be briefly going over key functions and explaining their use, purpose, and reason for existing. Furthermore, the modules incorporating Design Patterns will have an explanation of the present pattern, reason for it to be there, and the utility it provides. Each sub-section will go over a core module (.h/cpp pair). Do be aware that each design pattern has its own UML diagram in the appropriate section. For more visual representations of the functionalities of these patterns, kindly refer to the associated diagrams.

3.1 Strategy Pattern

Pandemic! is a board game in which players make choices about what actions to take on their respective turns in their mission to save humanity. For our program, this creates a requirement to dynamically run complicated action logic on each player's selection (actions, event cards, etc.). Strategy is an excellent pattern for this as it allows us to encapsulate each unique action and add a layer of abstraction and indirection. In our GameController, we prompt the user to choose an action. When an action is selected, to execute it we call the strategy for it which is then added to the ActionContext and executed. This allowed us to have better encapsulation since we would be removing the action logic and action handling from the GameController whose primary responsibility is game loop flow and game control. This makes our code very readable and allows us to make changes and modifications with little to no impact on core game functionalities. The files implementing the strategy pattern are ActionContext.h/cpp and Strategy.h/cpp which contain the interface and the concrete strategy classes, and, as aforementioned, the GameController.h/cpp which executes the actions on request.

3.2 Factory Pattern

The Factory pattern is very efficient at handling the creation of different objects that inherit from the same parent class (in our case: Cards). For this reason, we used a CardFactory to handle the creation of the different cards such as the InfectionCard, RoleCard, etc. To create a card using CardFactory, we supply the fstream which is used to read from the file where the cards are stored as well as a type which would tell the factory what kind of card to create (for instance an Infection card). Once the parameters are passed to the CardFactory, it will take over and use the type to choose which method it will use and eventually return the created card to be used when we populate the various decks. The CardFactory implementation is done in CardFactory.h/cpp and Cards.h/cpp.

3.3 Observer Pattern

Another design pattern we have chosen to use is the Observer Pattern, not because it was covered in class but mainly because of the versatility in functionalities it provides. The Observer Pattern allows us to display our views automatically whenever a change in the model occurs (which is essential). This greatly helps the flow of the game because the information that was changed will instantly be presented to the user (and prompts may be given to the players).

In addition, we have used a slightly modified version of this pattern. Instead of just simply notifying the observer, we also included a string-type message that is passed through the parameters of the functions which will then be displayed to the user for additional, more specific, information on the case. These messages are customized to the specific actions that were taken by having the action send a different message through Notify(string message), which will then trigger the Update(string message). The observer will then check the string and output accordingly to the players.

Each view on their own are two different observers. This ensures that the correct information from specific subjects are being updated onto specific observers, which allows consistency to be present throughout the game. The main files that are related to the Observer Pattern for the views are: Observer.h/cpp, Subject.h/cpp, MapView.h/cpp, and PlayerView.h/cpp.

In addition to the views, we used the Observer in one other case: the MedicObserver class. In the game the role of Medic causes the automatic removal of any cubes of a cured disease in the city they are in. Since the position of the Medic can be changed by the player or by another player through actions, the cleanest way to implement this logic was to set it up as an Observer and attach it the medic player if they exist in the game. This way, any time the medic moves location, the MedicObserver is notified and any affected cubes are removed. You can find the MedicObserver in MedicObserver.h/cpp and it is initialized by the GameController (GameController.h/cpp) in the initializeViews function.

3.4 Command Pattern

The command pattern was implemented to allow the controller logic implemented in the strategy classes to call on the setters of the Player class and change its state in an encapsulated way. The main advantage of using this is that we were able develop consistency in the way player state changes were made, by taking the responsibility of changing the state away from the strategy classes, letting them focus on determining which state needed changing and whether it should change (game logic). Therefore, multiple strategy classes could create and execute command classes as needed, which encapsulate the call to the state(s) change and all the required checks and resources handling required. The files for the Command pattern are Command.h/cpp.

4. Library Usage

For our version of Pandemic!, we chose to link to the external library “Boost” in order to make use of its extensive serialization functions. Boost was chosen because it allows for the serialization of cyclical data structures such as graphs which we used for our map. Boost's serialization was a powerful tool that handles all the pointer reassignments upon loading, making it very useful for our more interconnected models (city nodes, etc.). Other than Boost, we primarily used built-in C++ libraries such as <algorithm>, <iostream>, <exception>, <algorithm>, <deque>, <fstream>, <string>, and any others that were applicable for our project’s core functionalities.

5. Exception Handling & Processing

Throughout our project, ever since Build #1, we coded it in such a way that input is processed intelligently by the application to prevent critical errors or exceptions that would otherwise cause fundamental system crashes or logical errors. As such, in our InputOutput.h/cpp file pair, we have a method called getIntInput (and its modified version: getEventInput) that will intelligently process what the player(s) enters when prompted for an integer input (which is the core of what we require as input from players). The method will read in the variable data using std::cin, and will process it with lower and upper bounds. If it is not able to be read into an integer variable type, or if the variable is out of the bounds, we return an error message and ask the player to input his choice once more. In the case of getEventInput, if the players enter anything that is not a correct choice (in this case: ID of the player wishing to play the Event card), we don't return an error message, but rather treat it as though the players do not want to perform an Event card action. Therefore, the only true exceptions that we could process have been dealt with in that intelligent system-processing way.

The only place where we implemented exception handling is in Map.cpp and Cards.cpp when reading the .txt files for populating the data of our game. There, we perform exception handling for file I/O in general, and so this prevents faulty game launches or corrupt data from running, which could essentially ruin a game.

5. Operator Overloading

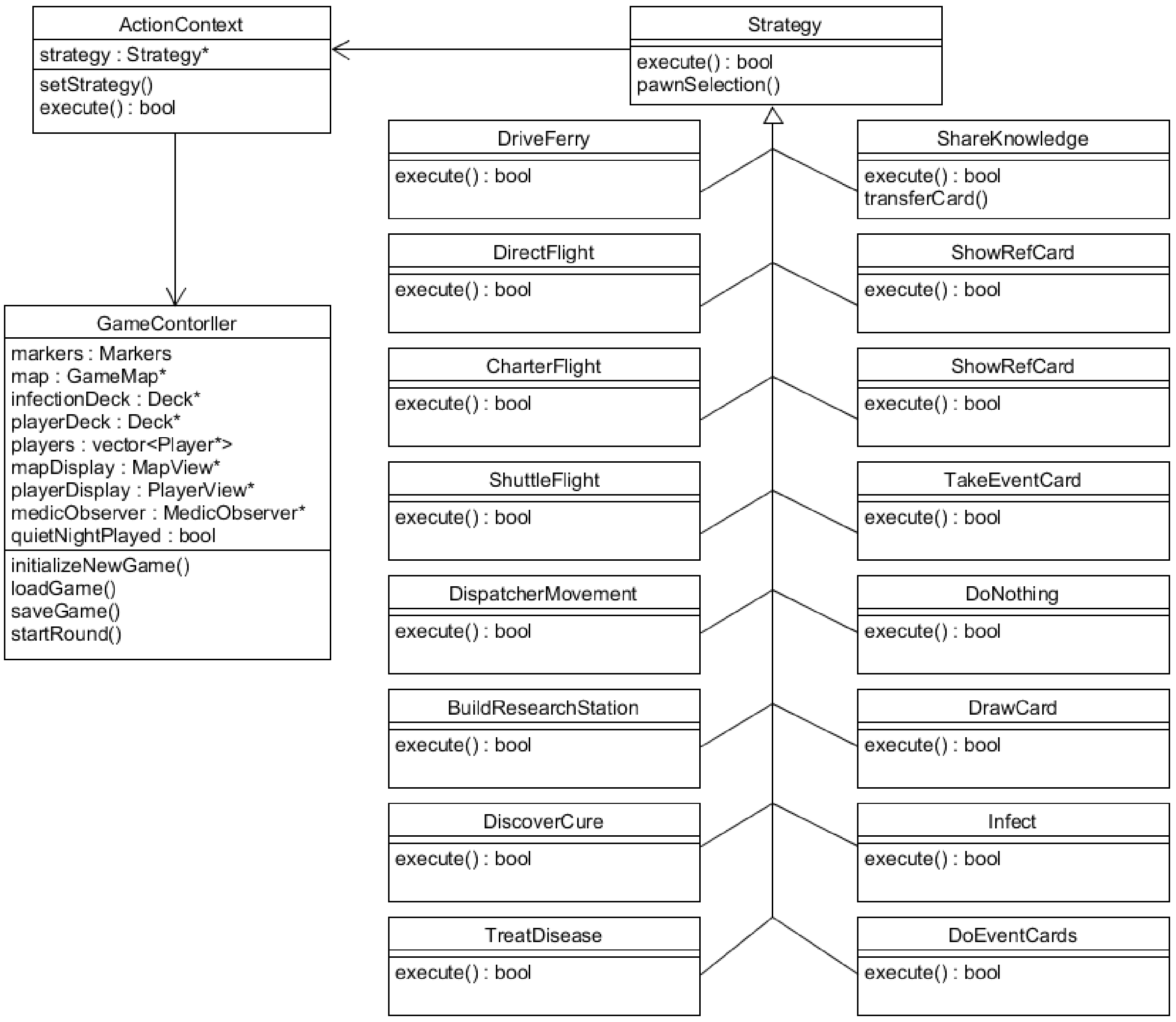
Operator Overloading was used in two models for very similar purposes. First, in the Card.h/cpp files, for each class of cards, the '==' operator was overloaded to allow comparison with a string. Some of our cards have string names or city names associated to them, so the operator was used to compare a Card object to a known identifier without needing to cast, given the polymorphic nature of the virtual overload. This allows a simple comparison to see if a card object matches the one you are looking for, among other uses. This modification allowed for a massive simplification in our code from build #1 as it allowed us to eliminate checking for types and dynamic casting when going through the comparison process.

The player model (player.h/cpp) also overloads the '==' operator to allow for comparing a player object to a string. As each player has a role card (which grants them additional abilities), this allows us to easily check if a player is holding a specific role card (has the given role) or not, thus simplifying the multiple function calls that are needed if this was not present.

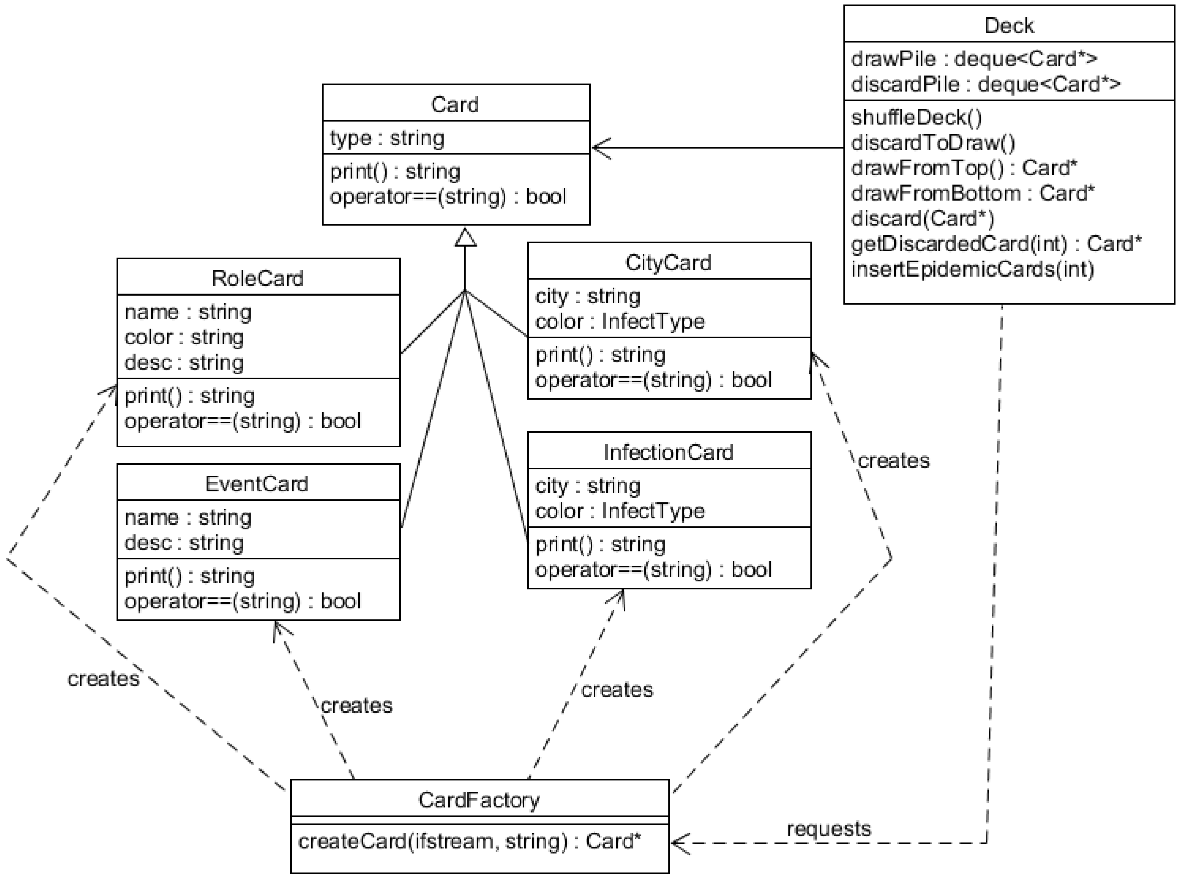
6. UML Class Diagrams

In this section, we will be providing the UML Diagrams of every design pattern. For extensive information on why we use them or how they work for our system, please refer to the Core Game Modules section.

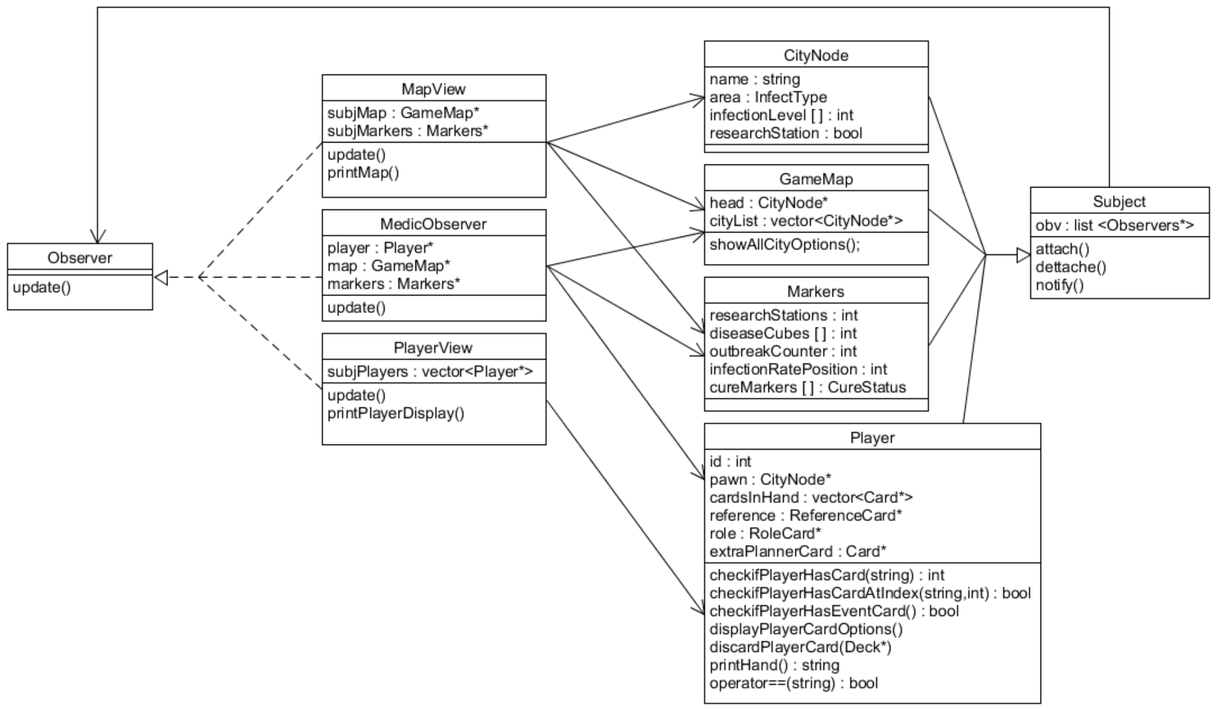
6.1 Strategy Pattern UML Diagram



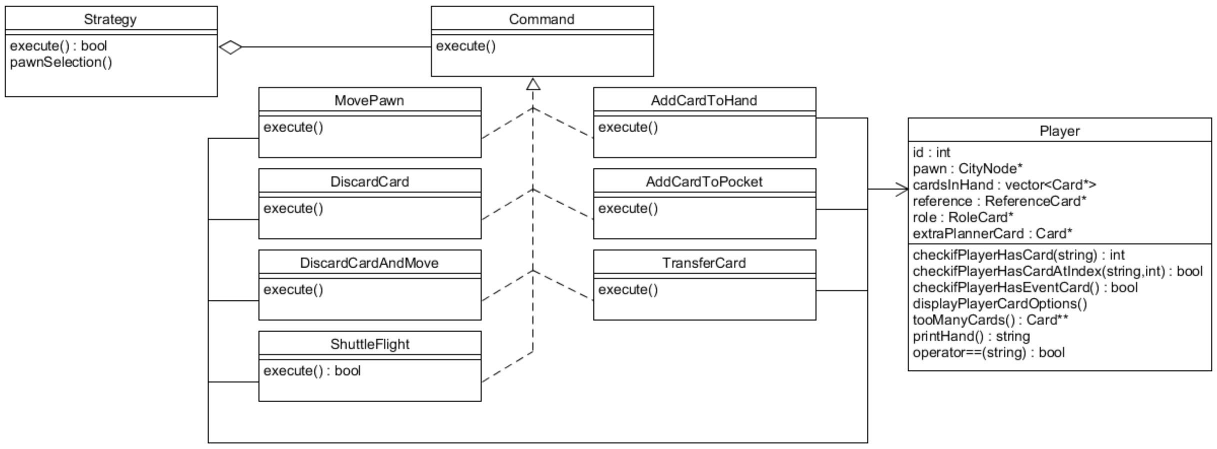
6.2 Factory Pattern UML Diagram



6.3 Observer Pattern UML Diagram



6.4 Command Pattern UML Diagram



7. Project Implementation Limitations

The project is currently fully functional in terms of the original board game. The game is completely playable from start to finish, and equally fair. You can lose, or you can win, in much similar ways to the original board game. The *only* thing that is not working currently in our project, primarily due to lack of time and the fact that it would need intensive reworking of some core functionalities, is the prompt timing for playing the Event card: “Resilient Population”. This Event card is unique in that you can play it between the Infect and Intensify steps of an epidemic, however in the way we coded the epidemic process it does not allow for anything to come in between of its core functionalities.

This, however, does not mean that correctly implementing the timing for Resilient Population is impossible, it just means that due to time constraints we were unable to implement it. Do note that the event card itself does exist and is functional, so only the timing for it is currently not operational, but you can play it whenever an Event card prompt is presented if you have it in your hand, much like the other Event cards in the game.

8. Limitation Solving Approach

As a group, we strongly believe that the only reason the correct prompt timing for Resilient Population was not implemented is due to a lack of time. Most of our group members are currently taking COMP371 (Computer Graphics) and as such needed to commit a lot of time to that class' project and implied work. That meant that our group was essentially starved at a certain point due to the incredible workload of COMP371's project. That said, were we to try to resolve the issue, we would likely try to recreate the epidemic process to include the Event card's prompt were it to be in any of the player's hands. As such, we would be splitting the epidemic step into several phases where we could properly prompt for Resilient Population while maintaining the core functionality of the remaining sequence of events.

9. Project Challenges

One major challenge we faced when using the Observer Pattern was the part where the views are printed to the screen. At the beginning, we had the most basic version of the Observer Pattern where it would print the entire view (Map or Player view) in unwanted places, which resulted in a very clustered and messy display. This caused a huge problem with the flow of the game since random prints of the map would appear in places you would least expect it.

As a result, we ended up researching more on the Observer Pattern and found out that a message can be passed through its parameter. As a result, we could regulate where the entire map would be displayed and display small messages where appropriate. This had greatly improved the flow of the game and allowed the user to stay close to important information without scrolling through the console to see past information.

Another challenge that we faced, but unrelated to the design patterns, is the User Interface/User Experience Design phase. The main drawback from using the console as our visual output is that we do not have a lot of flexibility in terms of displaying our visuals. It is very limited in character spaces and provides very little options to customize the output. So, more emphasis on organization and structure was necessary to ensure fluid play experience. As a result, we decided to use a full screen console to display our game and tried to utilize as much horizontal space as we can (compared to vertical space in standard console applications). This allowed us to reduce the amount of useless space on the sides and provided the user a smoother experience without the need to scroll up and down on the console. Our design ensures that the information displayed is within the reach of the user without the need for additional interaction.

10. Project Conclusion

Overall, we are all incredibly proud of what was accomplished with this project and of the amount of skills we've developed through the course in general. Design patterns opened a whole lot of doors for our futures as we can now approach problems in a different way by first taking a step back and thinking in a more architecturally-oriented way. By so doing, we've gained invaluable information and have advanced our problem-solving capacities.

Furthermore, being forced into groupwork and into situations where deadlines approach, where structure is crucial, and where design decisions and implementation methods are essential has forced us to develop a better sense of groupwork. The result is incredible, and we could not be prouder of the outcome. We admit, many of us played the game with their siblings or friends and it's become a very enjoyable experience and activity overall. Seeing the project go from an idea, to rough scratch work, to basic functionalities, to preliminary outputs, and finally to such a complete and robust application is a truly exhilarating experience.

The course has taught us a lot about planning, collaboration, problem-solving, and polishing. We believe that functionality indeed trumps visuals, but would love the opportunity to rework this project into something with a graphical interface and beautiful visuals. But, in any case, we knew what we had to do. Our task was simple, and our mission was clear. Together, we have managed to *save humanity*!

We hope you enjoy our version of Pandemic as much as we have!