Team Elephants Final Build Report

COMP 345 W2017 S

Marco Tropiano, Julian Hoang, Tom Bach

April 18, 2017

Table of Contents

Summary

Our group, Team Elephant, implemented the PANDEMIC game with attention to best practices. We accomplish this through an object-oriented design with MVC architecture. In addition, we use multiple design patterns in the code.

The primary design focuses on a command-based system that supports deterministic state saving and loading. Game actions are then implemented exclusively through this command system.

We support user interaction using a UNIX-style shell. Players type in commands which are then validated and executed by the engine. Commands are saved to a buffer that can be written to a file and later loaded.

Our implementation uses many of the newest C++ features including user-defined literals, auto variables, auto functions, range-based for loop, smart pointers, and more.

Essential Game Components

Game Setup

Our game is setup from file using a scripting system. The engine loads scripts from the *script/* directory. Provided is a default.*pandemic.txt* file that sets up a 4-player game as specified in the PANDEMIC instruction manual.

Out of the box, we support 2, 3, and 4-player games using the corresponding script in *script/*. We also support all the difficulty levels by loading the *d-intro*, *d-standard*, and *d-heroic* scripts. The player can use any combination of player count and game difficulty. They can also set their role by using the **give-role <player> <role>** command.

The user can also setup a custom game using editor commands. Game features including, but not limited to, cities, players, research stations, and game cards can be edited. See **Appendix A: Editor Commands** for a listing of all editor commands. To validate the map correctness, enter **validate-city-connections**.

The current state can be saved any any time using the **save <name>** command. It can then be loaded using the **load <name>** command. The player can then continue playing the game.

Game Dynamics

At any time, the player can enter **help** to get a list of commands.

The player begins a game by loading a setup file. Entering **load** will load the default 4-player setup file. Start the game by entering **begin**. The prompt will show the current user’s turn, for example:

```
orange>
```

Enter any of the player actions to execute that action. All of the player actions in PANDEMIC are implemented, see **Appendix B: Player Commands**.

To see the game state, such as cities, players, and decks, enter one of the view commands. See **Appendix C: View Commands.**

After performing 4 actions, enter **end** to end the player's actions. The game will then draw 2 cards into the player's hand, infect the appropriate cities, and give the turn to the next player. Messages will be displayed to the console showing what happened.

The application will automatically save the game state after finishing a turn. It will have the filename *save-<id>* in the scripts directory. In addition, the player can save the game using any filename by entering **save <name>**.

Implementation of Commonly Overlooked Rules

- "You do not draw a replacement card after drawing an Epidemic card."
    - Implemented in: **void draw_card(context &, end_actions::args_type const &, end_actions::ostream_type &)**
- "You may Discover a Cure at any Research Station — the color of its city does not need to match the disease you are curing."
    - Implemented in: **void discover_cure::run(context &, args_type const &, ostream_type &) const**
- "On your turn, you may take a card from another player, if you are both in the city that matches the card you are taking."
    - Implemented in: **void share_knowledge_from::run(context &, args_type const &, ostream_type &) const**
- "On your turn, you may take any City card from the Researcher (only), if you are both in the same city."
    - Implemented in: **void share_knowledge_from::run(context &, args_type const &, ostream_type &) const**
- "Your hand limit applies immediately after getting a card from another player."
    - Implemented in: **static void hand_limit(context &, args_type const &, ostream_type &)**

Implementation of Bonus

Play back the recorded games by entering (a) **replay bonus-cured** or (b) **replay bonus-outbreak**.

Modules

See **Appendix D: Directory Structure** for information on the project folders.

Application

The application class contains the driver for the game. It is responsible for taking user input, performing actions, and outputting information to the screen. The constructor requires that the input and output streams are specified so that the class does not depend on global state.

In the constructor for application, the enabled commands are inserted using the **insert_command** template member function. The function also specifies the command type, which is used to categorize the command for the help function and keep track of actions remaining.

The rationale for using an application class instead of putting the driver in main is to clearly specify the dependencies of the driver (input and output stream) and encapsulate the functionality of the driver into a reusable component.

Actions

All game actions and editor actions are implemented as commands. All commands inherit from the **command** abstract base class. Each command must implement the **name()**, **description()**, and **run()** virtual member functions. We require that commands be deterministic in order for the save/load system to function. Commands must produce the same output for a given input.

The rationale for using commands is that it matches the action-driven nature of the game. Player actions are represented as commands, as is editor actions. An additional benefit is that it makes development much easier, as team members can work on commands in isolation, which reduces merge conflicts.

Save and Load

Since commands are deterministic, we save the game state by writing to file all the commands that the user entered that session. The game is then loaded from file by reading the save file and replaying all the commands in order. This brings the game up to the exact state when it was saved.

The rationale for using this command replay system for saving and loading is that it enables state serialization without hooking into the model. In fact, any new commands that are added to the game automatically have serialization support because they are deterministic and can be replayed. This

drastically simplifies development because team members do not need to concern themselves with serializing state.

Handles

All game objects are references using handles. They are immutable opaque objects that are passed to the model to specify an object to perform an action on. We implemented user-defined literals to allow handles to be constructed easily in source code using the **_h** string suffix. The **handle** class also contains a constructor taking a **std::string** to allow handles to be constructed from user input.

The rationale for using handles is that it simplifies memory management and memory safety. The internal pointers in the model are never exposed to the view or controller. This ensures that these classes never have stale pointers. The liveliness of an object is automatically checked when handles are passed to the model. If an object is not live, an exception is thrown. This use of handles reduces the risk of undefined behavior associated with dereferencing invalid pointers.

Design Patterns

Command

The command pattern is the backbone of the application. All player and editor commands inherit from the **command** class. Each concrete command is then instantiated and executed when the player enters the appropriate input. Since all commands are deterministic, we can play back saved games by playing back the command log. In addition, game scripts can be created by editing a text file with commands.

The rationale for using commands is that it encapsulates the command name, description, and functionality of game actions. This decouples the invoker of the command from the concrete command itself. It also allows for a clean implementation of save/load by replaying command input.

Observer

We use the observer pattern to implement the **map_view** class. It provides an automatically updated display of city state by observing an **observable_cities_model**. The model notifies observers of the following changes:

- Adding a city
- Connecting a city
- Placing a research station
- Removing a research stations
- Adding disease cubes
- Removing disease cubes

The rationale for using an observer is that it decouples the map_view from the observable_cities_model. The map_view does not know of the existence of observable_cities_model. It

only wants to be notified when a city state changes. In addition, observable_cities_model does not need to know the specific classes that will be observing its state. By using an observer, the two classes are decoupled.

Builder

We use the builder pattern to construct cities with their connections. The **city_builder** class provides functions to incrementally build city state by providing the name, region, and any number of connections. In addition, it provides a function to get the completed city object.

The rationale for using a builder is to manage the complexity of constructing connected cities. There is a two-way relationship between connected cities. We do not want to have city A connected to city B without city B also being connected to city A. To manage this, we use the builder pattern to incrementally construct the cities, and then only get the resulting object once the connections are completed.

Decorator

We use the decorator pattern to implement additional behavior on top of the card decks. The **epidemic_decks_decorator** class provides functionality to get the epidemic cards in a deck or player hand. In addition, the **event_decks_decorator** class provides functionality to get the event cards in a deck or player hand.

The rationale for using a decorator is to keep each class to a single responsibility. The deck class should only be concerned with managing the deck state. The functionality of getting a specific type of card from the deck can be implemented as a decorator to keep the responsibilities separate. Each decorator is only concerned with getting its specific type of card.

Strategy

We use the strategy pattern to implement the methods of drawing a card from the deck. The **draw_card_from_top** strategy class and **draw_card_from_bottom** strategy class implement their respective drawing methods. The **draw_card** is then set to use one of these strategies when drawing a card from the deck.

The rationale for using the strategy pattern is to be able to allow different parts of the code specify how they draw a card from the deck at runtime. For example, the infection deck is normally drawn from the top, except during an epidemic, where it is drawn from the bottom. To implement this runtime behavior, the strategy pattern can be utilized.

Code

Libraries

Our project uses purely the standard library. We do not use any other third-party libraries.

Operator Overloading

The **handle** class has three operator overloads, **operator<**, **operator==**, and **operator<<**. The < operator is overloaded so that handles can be stored in the std::set and std::map containers that require a total order relation between their elements. The == operator is overloaded to allow identity checks of handles. For example, to see if the player role is a medic. Finally, the << operator is overloaded to allow writing the handle's string representation to the console for user display.

Sequence Containers

We use **std::vector** to store sequences of elements whenever they are needed. In addition, **std::dequeue** is used to store the card sequence. This container was chosen in particular because it supports addition and removal at both ends of the sequence. This is needed to add or remove cards from the top or bottom of the deck.

Associative Containers

Our code makes heavy use of associative containers. In particular, **std::set** and **set::map** are often used in the model classes. These containers are used to track collections, such as the player hand, as well as map city names to their internal objects.

Iterators

We made a significant effort not to expose the internal containers to systems outside of the model. The only way to iterate over a model is to use the **begin()** and **end()** member functions that return a constant iterator to the underlying container. As constant iterators, they can not be used to modify the container elements, which protects the state invariants in the model. All of our models support iteration using ranged-based for loop.

Algorithms

A huge benefit using iterators is that many algorithms can be run on the models. An example is shuffling the card deck, which is done in a single function call using **std::shuffle**. We can also use **std::find** to get specific elements in a model.

Templates

We use a template member function, **insert_command<T>**, in **application** to instantiate commands in the constructor.

Smart Pointers

The **application** class uses **std::unique_ptr** to store the command instances. This ensures that the underlying objects are deleted when the application goes out of scope.

Exception Handling

The first level of exception handling happens in **application::call_command**. If the user does not enter a valid command, **std::out_of_range** is thrown. The function catches this exception to provide a "command not found" error message to the user.

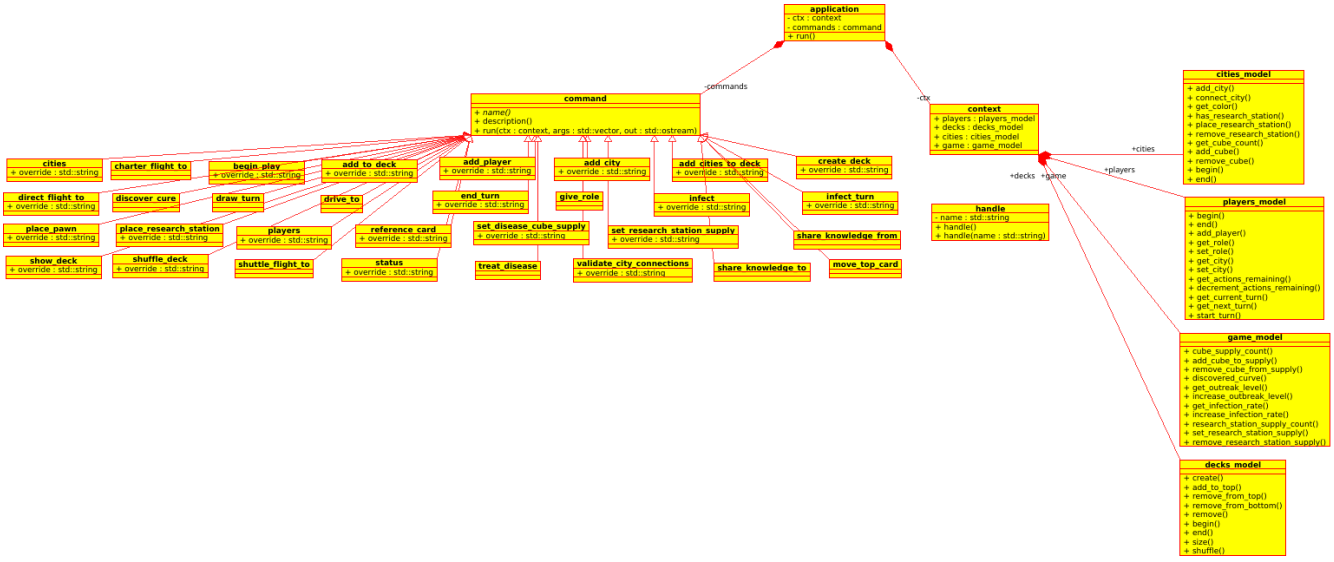
The next level of exception handling happens in the individual command classes. If the user does not enter enough arguments for a command, the command class will catch the exception, and display a help message to the user. For example:

```
orange> drive-to
usage: drive-to <city>
```

The final level of exception handling happens if the user enters an invalid input. For example, if the user enters a city name that doesn't exist in **drive-to**, the command class will catch the exception and display an error message to the user.

UML

See **uml.png** for larger version.



User Manual

Getting Started

Welcome to the PANDEMIC game! You can type **help** at any time to get a list of commands. To get started, load the default game script and begin playing:

```
> load
> begin
```

The player with the green pawn can now begin playing. Try moving to Chicago by using the **drive-to** command.

```
green> drive-to chicago
```

Save your game at any time by entering **save <name>**. The game also automatically saves when you end your turn.

Ending Your Turn

When you have completed the actions that you want to do, enter **end** to end your action phase. The game will draw 2 cards for you. If an epidemic card is drawn, additional cities will be infected at this time. The game will print what is happening so you can stay up-to-date. Finally, the game will infect cities by drawing the appropriate city cards using the current infection rate.

An epidemic will occur if an epidemic card is drawn. An outbreak can also occur during infection phase. You have a hand limit of 7 cards – if you go over this, a card will be discarded from your hand.

Views

To see your current city, use the **players** view command.

```
green> players
PLAYER ROLE        CITY
green  researcher  chicago
…
```

There are several view commands you can use to see the state of the game world. See **Appendix C: View Commands** for a full listing, or simply enter **help** in-game. The common view commands are:

- **status**
- **players**
- **cities**
- **show-deck [deck]**

Actions

There are several player actions you can do during the game. See **Appendix B: Player Commands** for a full listing, or simply enter **help** in-game. The common movement actions are:

- **drive-to <city>**
- **direct-flight-to <city>**
- **charter-flight-to <city>**
- **shuttle-flight-to <city>**

You can also do other actions:

- **build-research-station**
- **treat-disease <color>**
- **share-knowledge-to <player> <card>**
- **share-knowledge-from <player> <card>**
- **discover-cure <color>**

Your role may give you additional actions, such as **store-event** (contingency planner) or **move-player** (dispatcher). It may also modify the behavior of existing actions or give you passive abilities. See the Roles section below.

Events

You may play event cards if you have them in your hand. Enter **play-event <player> <card> [args...]** to play the card.

Airlift

Enter **play-event <player> airlift <mover> <city>**. Moves the **mover** player to the given **city**.

One Quiet Night

Enter **play-event <player> one_quiet_night**. Causes the next infection phase to be skipped.

Forecast

Enter **play-event <player> forecast**. Allows you to rearrange the top 6 cards in the infection draw pile.

Government Grant

Enter **play-event <player> government_grant <city>**. Places a research station at the given **city** for free.

Resilient Population

Enter **play-event <player> resilient_population <card>**. Removes the given **card** in the infection discard pile from the game.

Roles

Contingency Planner

Take an event card from the player discard pile by entering **store-event <event>**. You can then play this card by entering **play-event <player> <event> [args…]**.

Dispatcher

Enter **move-player <player> <city>** to move another player's pawn. Either (a) the target city must contain another player or (b) use a movement ability (using cards as appropriate). You can not do non-movement actions (such as treat disease).

Medic

Using **treat-disease <color>** removes all cubes of that color from a city. If the disease is cured, this does not take an action. You also prevent placing disease cubes and outbreaks of cured diseases at your location.

Operations Expert

Using **place-research-station** does not require or discard a city card. You can also move from a research station to any city by using **shuttle-flight-to <city> <card>**.

Quarantine Specialist

You prevent outbreaks and the placement of disease cubes in your city and all connected cities. Does not affect startup phase.

Researcher

You can give any city card in your hard to another player by entering **share-knowledge-to <player> <card>**.

Scientist

You can use **discover-cure <color>** with only 4 city cards of the color.

Winning

To win, you need to discover all the cures in the game. The game will let you know when this condition is reached by displaying this message:

```
All four cures has been discovered. Humanity is safe... for
now.
```

Losing

There are three ways to lose the game:

```
Worldwide panic! Outbreaks marker reached last space of
Outbreaks Track!

Disease spread too much! Unable to place number of disease
cubes needed on the board!

Out of time! Can not draw 2 Player cards after doing actions!
```

Customization

Try different player counts by loading the *2-player*, *3-player*, or *4-player* setup script. Then set your difficult by loading either *d-intro*, *d-standard*, or *d-heroic*. Or create your own setup script and completely customize your game.

We hope you enjoy playing PANDEMIC by Team Elephants!

Conclusion

We achieve a final build implementation that implements the complete PANDEMIC game rules. In addition, our customizable game engine allows the user to create their own variations of the game using the scripting system. Overall, our team made use of design patterns to create a quality end-product.

The main challenges were (1) implementing complex and interdependent game rules such as role actions and (2) learning how to use version control.

In conclusion, Team Elephants has achieved a complete PANDEMIC game implementation through careful design and implementation of software architecture.

Appendix A: Editor Commands

Command	Description
place-pawn	Place player pawn
add-city	Add a city and its connections to the map
add-player	Add a player to the game
create-deck	Creates a new card deck
shuffle-deck	Shuffle a deck
add-to-deck	Add a card to the deck
move-top-card	Moves the top card from a deck
give-role	Give a role to player
add-cities-to-deck	Add city cards to the deck
place-research-station	Place a research station on map
set-research-station-supply	Set the number of research stations in supply
set-disease-cube-supply	Set the number of disease cubes in supply
infect	Infect a city during setup phase
validate-city-connections	Print any city connection validation errors

Appendix B: Player Commands

Command	Description
direct-flight-to	Direct flight to a city
drive-to	Drive to a city
shuttle-flight-to	Flight to city using research station
charter-flight-to	Flight to city using city card
treat-disease	Remove disease cube
share-knowledge-to	Give player card to another player
share-knowledge-from	Take player card from another player
discover-cure	Discover a cure using a research station
play-event	Play an event card
store-event	Contingency planner: stores an event card
move-player	Dispatcher: moves another player pawn

Appendix C: View Commands

Command	Description
players	Show player state
show-deck	Show deck or player hand state
cities	Show cities state
status	Show game status
reference-card	Print reference card

Appendix D: Directory Structure

Directory	Description
Build1_Report	Build #1 report
Final_Report	Final build report
model	Model layer – data storage
view	View layer – data display
controller	Controller layer – command dispatch
controller/editor	Editor controllers – for editing game rules
controller/action	Action controllers – in-game abilities
pattern	Design pattern base classes
script	Configuration component and save/load data
text	Reference card user text
cmake-3.7.2-win64-x64	CMake for Windows