Team Elephants Build #1 Report

COMP 345 W2017 S

Marco Tropiano, Julian Hoang, Tom Bach

March 18, 2017

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. 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 *setup.pandemic.txt* file that sets up the game as specified in the PANDEMIC instruction manual.

The user can also setup a custom game using editor commands. Game elements such as cities, player cards, starting location, research stations, disease cube counts, and more can be edited. The current state can be saved any any time using the **save <name>** command. It can then be loaded using the **load <name>** command. To validate the map correctness, enter **validate-city-connections**.

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 setup file. Start the game by entering **begin-play**. The prompt will show the current user's turn, for example:

> blue>

This indicates that it is blue's turn.

Enter any of the player actions to execute that action. For example, **drive-to montreal** will move the current player pawn to Montreal if that is a valid action. The player can see their actions remaining by entering **status**. After exhausting their actions, the player draws 2 cards by entering **draw-turn**. Finally, the player enters **infect-turn** to infect cities. The cards drawn and cities infected are printed to let the player know what happened. Then, enter **end-turn** to give the turn to the next player.

The user can enter any of the view commands to show game state: **show-deck <deck>**, **cities**, **players**, and **status**.

Architecture

Model-View-Controller (MVC)

We use 4 models, 4 views, and 27 controllers in our MVC implementation. The models are responsible for maintaining game state invariants. Next, the views provide game information to the user in table format. Finally, the controllers implement the game actions.

The rationale for using MVC is that it allows a clear separation of concerns in maintaining game state, displaying information to the user, and executing game actions.

Application

The application class contains the driver for the game. It is responsible for taking user input, performing actions, and outputing 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.

Commands

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.

Code

Libraries

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

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 library

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

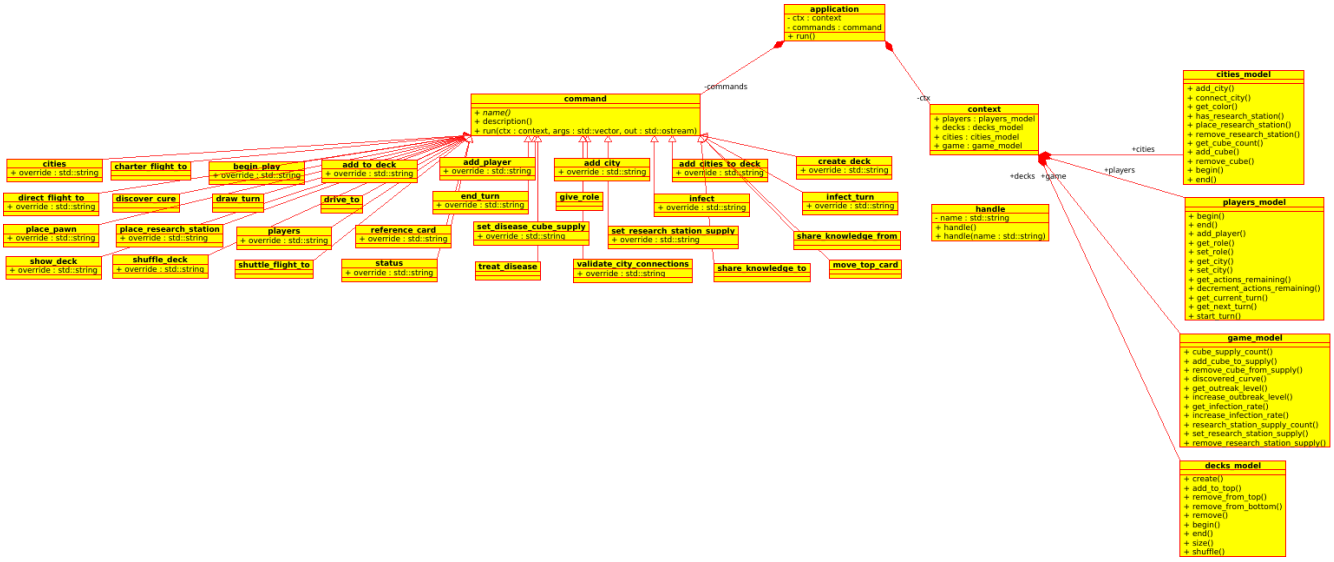
We use a template member function in **application** to instantiate commands in the constructor.

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.

UML

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



Conclusion

We achieve a complete build #1 implementation using a cohesive design that focuses on deterministic commands. This use of commands naturally fits into the action-driven model of the PANDEMIC game. It provides automatic save and load support by replaying commands.

The main challenges of the project came from implementing individual commands. In particular, the handle-based approach to object references is different than the traditional pointer approach, and takes some time to familiarize with the system.

Overall, the system is well-suited to implementing the remainder of the project, and should not require major changes.