**HOW TO RUN**

-Load mexicantrain.lsp into LISP interpreter and then execute s-expression (mexicanTrain)

-Menus are operated almost entirely numerically except for filenames to load serialization files. Simply enter the appropriate number corresponding to the menu choice you wish to select in playing the game.

**BUG REPORT**

**FEATURE REPORT**

Serialization/deserialization

Main Menu

Start of new game

Start of new Round

Computer/Human drawing 16 tiles from boneyard

Correctly displaying all round information

**DATA STRUCTURES/MODULES**

-The client module contains only client code to load additional files and start the game

-The game module manages new games, resuming games, and victory when the player is done playing

-The round module manages new round, current rounds, ending rounds, alternating turns, and determining victory of round

-The utilities module contains a variety of helpful bottom level functions to aid in design by providing all the tools to design in a purely abstract functional way. Functions include list shuffling/reversal, getNth, setNth, remove, element checking if a list contains element, getting last element, and validating console input

-The boneyard module maintains the bone

**DAILY LOG**

**01/26/21**

-Created Client and Game class

-Added main loop to Game class

-Created git repo for source control

**01/27/21**

-Created round class

-Incorporate some accessors from round into main loop

-Began some more bottom-up design after realizing top-down was slowing me down. Created Tile, TileSet, Hand, Train, and Boneyard class with inheritance

-Added Player, Human, Computer classes with inheritance.

**01/28/21**

-Used vector to represent the dominos in TileSet. TileSets, Tile, and Boneyard can print themselves.

-Added all of round setup to Round, and many accessor and mutator functions to the Tile and TileSet based classes to accomodate Round.

-Split up Round into many smaller logical functions. Some still may be too busy. Will work towards reducing as many to simple english sentences as possible. Consider a Table class to separate round logic from Table mechanics

-Ran into circular dependency issues. Began using forward declarations as necesarry for header declarations and then including the header in the implementation file. In order to keep things simple as far as the forward declarations, I decided to stop using things like typedef Player\* PlayerPtr.

**01/29/21**

-Definitely want a Table class down the road.

-Since representation of a train requires 3 lines, changing some print functions to functions returning string. Need an elegant way to keep iterating 3 times throughout code. Add a constant at the very least.

**01/30/21**

-The round flow was completed along with game flow. Stubs still exist for going first as well as taking turns. But the top most logic is all there.

-Game asks round if it is over, inside round it checks for both players having passed along with empty boneyard. will have to add empty hand condition. i assume that one will be easier to check after a tile is played and victory logic can occur if and when a player uses last tile. this can raise a boolean for round's isOver() to check in addition to its current check for passes/boneyard empty

**01/31/21**

-Reorganized order of some things to follow proper C++ implementation

-Still no headers on files or functions, and few comments. Will add hopefully soon

-Added player confirmation to keep playing. Still using input and output in classes. Need to figure out how to structure a viewer class.

**02-01-21**

-Added dice rolls. Until viewer class is figured out, at least the input is limited to the Human class, which seems more reasonable than inside Round.

-Player can draw from Boneyard passed to it. Wasnt sure if should just pass any TileSet but Player only ever draws from Boneyard so this seemed most verbose and readable.

-Have began thinking ahead of how Player turn will work and what responsibilities each class will have as far as working together (tiles being added to trains, who verifies it? etc)

**02-02-21**

-Added file headers

-The function to correctly draw game state including scores, hands, trains (on correct side of engines), mexican train, and top of boneyard (and boneyard count).

-Player class has printHand. Computer and Human have slightly different implementations of printHand (for sole purpose of displaying whether it is human or computer). Considering adding a virtual getID() or getName() for both classes to implement but this is very low priority at the moment.

-Added destructors. Verified all looks correct but will not have ran profiler yet by time of submission

**02-04-21**

-Changed the tiles vector in TileSet to vector<Tile\*> \*tiles so that it could be initialized in the constructor

**02-06-21**

-Added Train's ability to know have market set and checked, as well as checked for orphan doubles

**02-07-21**

-Expanded Human Player and Computer for basics of turn menu, Stubs for help and save

**09-05-21**

**12:35** Reacquainting myself with my previously written code, and referencing this document and the rubric to aid in the process. These documents have helped immensely at orienting myself quickly. Soon I will be able to jump in.

**14:04** I am now familiar with my code again and have double-checked every item on the rubric to ensure it is completed. I am now going to move on to serialization as I believe this should have been a higher priority from the beginning. Why? Because I found myself editing my code in order to test various conditions on the rubric. With serialization I can streamline this process by manipulating the hands and situations and isolate each mechanic to be tested one by one.

**14:14** I have made the decision to import FileAccess.cpp and FileAccess.h from my Software Design class. These files are more than adequate to take care of my file access needs for this project, and I feel no need to reinvent the wheel when I do serialization. Professor Miller is credited within the file and I will make sure to credit him wherever required in this document.

**16:28** I have begun modifying Game.cpp to include some new functions to facilitate the option to resume or start a new game. Instead of just initializing the players and round in the Game constructor, start() is called, which uses displayNewGameMenu(), validInput(), newGame(), loadGame(string), resumeGame() and validFile(). newGame() contains all the old new game logic, where the players and round are initialized to default values then dealt random hands.

**19:09** loadGameFromFile is created from loadGame and now correctly extracts all of the required data. It is all passed along to an alternate constructor for Round, which calls a new member function of Round called resumeGame. It will then use all of the data extracted from serial file to start the game and resume where it left off.

**09-06-21**

**11:02** I have made a lot of progress in the deserialization and loading game process. I have been experimenting with some ways to handle the new feature in Round and Game, and have created resumeGame() for Round. I had started to add some constructors to Train, Boneyard, TileSet, and Hand to facilitate the deserialization of the strings, but after I got my feet wet I realized that the format of the strings presents some new problems to face. I was hoping to parse the strings very late in the process, and in class-specific implementations of deserialize() (called from the constructor of classes that inherit TileSet), but I realize that not all strings of tiles are created equal. There is the issue of the marker as well as the issue of the order. Order matters for everything but player/computer Hand (although I should try to preserve the Hand order as well, even if it is not game critical). Headers are getting messy and includes as well. I need to move to a precompiled header soon and also be a little more uniform in my usage of std classes. Moving forward, I am going to consider making a static helper function for TileSet that accepts a string of tiles and returns a Vector. The vector can be reversed with a simple line, and the vector is what can and should be passed into the TileSet-derived classes, not the string. This only leaves the marker to deal with. The marker will have to be removed before accessing this Utility function. I will do this in Round::resumeRound(). This is also where I will call the utility helper function to get my deserialized vectors, and where I will determine which need to be reversed. The TileSet classes will be able to rely on a properly ordered vector. NOTE: Boneyard and Computer need to be reversed, human is fine as is.

**15:43** The deserialization process is “done” as far as foundation goes. All of the tasks required to do it are implemented, and it compiles and runs, but there are some errors. It got things right for the most part, but some end-tiles are repeated, and the human hand is all wrong. I suspect my use of while(!stream.eof()) needs some work. Not bad for a first try. It is also noted Round::resumeRound() has become long and is taking on too many aspects of the process by itself. I ended up not needing a utility function as I did not use vectors, which contributed to this problem. I chose to std::reverse() the strings instead, eliminating the need for unnecessary vector in the process, but did not create a utility function to help with this. I will worry about ways to refactor after I fix the bugs.

**09/07** **16:00** Notes lost due to mistake during transfership of project to new repo. Code was not lost. Summary as best as I remember: The bugs were all fixed. Deserialization is complete and accurate. Round winning is added. Just need to add scoring, option to serialize and the serialization process (should be easier than deserializing!) I need to double check if the code I wrote in winter-time supports 10th round, or attempts to make 0-0 engine or something terrible. Once I have verified those tasks, I can complete computer AI steps. The rest after that is cleaning up/refactor/applying consts in the right places, and any other style guidelines.

**18:15** I have completed many of the rubric fill-ins, including victory conditions, draw, scoring. I have a clean hierarchy from round to Player to TileSet to Tile in order to get a scoring. My structure is not perfect (isn’t conducive yet to MVC model) but it does in my opinion show good use of inheritance and polymorphism in a game environment.

**9/25 15:00**

An override of getOutputLine was added to Train class. This override simply inserts

“ M “ or “ “ (3 spaces) to the front or back of train output string, depending on the parameter leftToRight which was already there from before with the pip orientation. I chose to put the M in the middle so I could use the same string agnostic of orientation. I am beginning to get sick of left to right orientations, but I am happy that I am 100% sick with that part of the project!

I have complete the top level of serialization. The menu now asks for a filename, opens the file (or throws an error and asks again), calls the save function, and quits. The save function correctly outputs the structure, but needs to be fed with the right information.

**10:31pm** Despite constant nosebleeds that last for hours, I have triumphantly completed the serialization process. There were unexpected problems, and I discovered some issues with deserialization, so that’s good. I am load/save the same file in cycle with no problem. I am going to test with all 3 cases from the website just to make sure they load no problem.

I am sleeping now as I have lost a lot of blood.

**9/26**

**17:30**

I have completed the computer’s ability to make moves. It correctly follows its strategy, but I failed to plan ahead for duplicating this structure in such a way that it will work for player as well as human. I am redesigning it to support the option to provides reasons, and to return a valid move rather than simply take it. This will integrate nicely into the Help system. I simply need to make the string uses modularity.e.g. cout << stringPiece << “ move “ << tileString << “ to “ << trainString << “ because “ >> trainReason << “ and “ << tileReason << endl;

stringPiece would be “The Computer decides to“ or “The Computer recommends you“

tileString is the left to right output of a tile, trainstring is the name of the train.

Train reason is simple: if the train has an orphan double, that is the reason why that train was chosen. If not, then one of 3 cases: human train: “because it has a marker”, Mexican: “because it is better to play on Mexican train than own train” and own train: “because it is the only valid train to play to”. Then tile string is simple as well: if it is a double, then “because it is the highest double able to be played to the most advantageous train” and if not “because it has the highest pip count able to be played to the most advantageous train.

I will take a moment to discuss a weakness of this strategy:

Computer player prefers to play a 0-0 to human train over playing a 9-9 to its own train, which is not good strategy. A proper strategy would not isolate 1 train at a time, but assign a weight to each train and to each tile. Having a marker would probably increase your trains weight. Doubles would be highest weighted tiles followed by highest pipcount. Some hybrid of these 2 weights would be a more ideal way to calculate best move, but I will not be going that in depth here.

**19:09**

I have functionally completed the entirety of the game, all specs, all tests work.

**SCREENSHOTS**

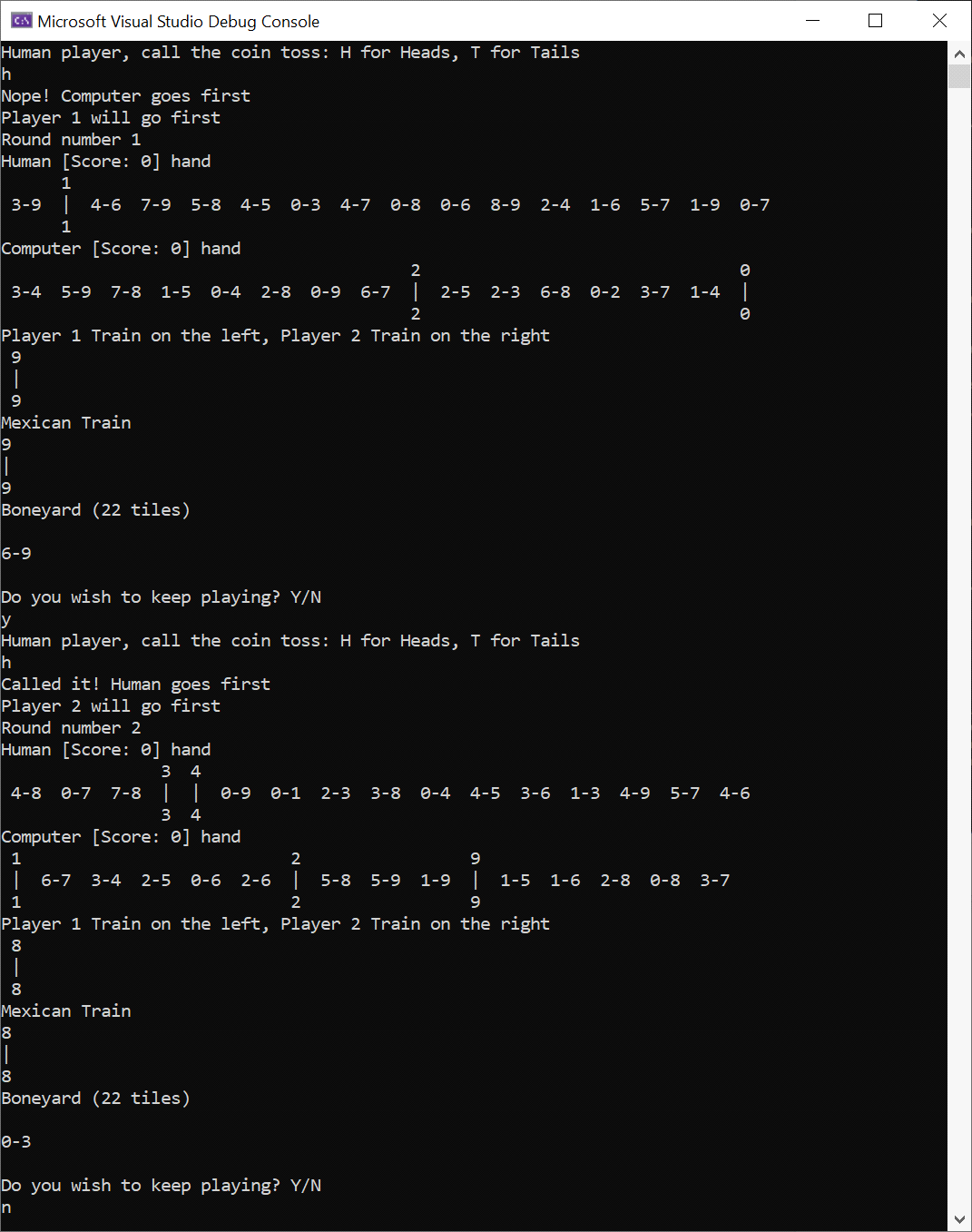


Figure Early tests of turn cycling

Text

Description automatically generated

Figure 2Case 1

Text

Description automatically generated

Figure 3Case 2 Computer plays a double then a non double, justifying reason

Text

Description automatically generated

Figure Case 3, where no player gets to play

Text

Description automatically generated

Figure 5Player gets advice

Text

Description automatically generated

Figure 6Game is saved then exited

Text

Description automatically generated

Figure 7Menus require input validation