ECE 421
Assignment 4 Design Report

By: Vishal Patel, Jose Ramirez, Rizwan Qureshi, Curtis Goud, Jori Romans

Design Rationale:

We've designed the GUI application with the goal of providing an easy to use, simple UI capable of playing both Connect4, and TOOT/OTTO. The UI has been designed in such a way that allows for maximum extendability to add new game features, games, and players. The features we've chosen to support with our UI is as follows:

- Title screen, game screen, settings, win/loss conditions, and customization
- We've included customization such as: color blind mode, custom board sizes, custom screen sizes, and custom colors for usability.
- Game inputs such as Keyboard controls and mouse controls
- An AI player to play the games against.

To make our game application extendable, we've designed this interface with the goal of modularity in mind. This will allow us to add additional features and games in the future if necessary.

The main application loop is a state machine consisting of application states. When you launch the application you land in the TitleScreen state. From here you can access the SettingsScreen state or the GameState. The SettingsState allows you to edit the settings of the board size, theme and any other relevant settings. The game state launches a new state machine that allows the user to play the game. When the user is finished, the application loops back to the TitleScreen state.

In the game state, we use the template design pattern to implement a general template for all valid games that can be played with our NxN matrix game board. A valid game will setup the board with an initial state. Then it will define an algorithm on how to play the game. Finally, it will clean up the board and ask if the user will like to play again. For the game algorithm to be valid, it has to define at minimum the number of players, what a valid turn is and the victory condition.

Finally, we define players as separate objects that we can query for moves. This abstracts away the implementation for acquiring moves and allows us to define customized players that can get input from the keyboard or from the mouse or AI players that generates their computed move.

Overall, this modular design supports extendability because if changes need to be made, we can either add on an additional module or adjust an existing module. For example, if we want to create AI players or online multiplayer, we simply need to create

more player objects that extend from the base player class and with polymorphism, we can switch out one of the real players with our custom player.

System Research:

1. What can we do on a computer than we can't do on a printed board?
    - Automated score tracking which can be useful when playing "best of" games or when a game needs to keep track of points.
    - Easier to reset then a real board, especially when there's alot of pieces.
    - Can have a nicer user interface with animations and special effects.
    - Automation of tedious tasks. For example shuffling the deck, handing out cards, setting up the pieces on the board.
    - Ability to have AI opponents and online multiplayer.

2. What is a computerized opponent? Remember, not everyone is an expert. What are its objectives? What characteristics should it possess? Do we need an opponent or opponents?
    - A computerized opponent is one that replaces a human player and generates valid moves.
    - Allows the player to experience a multiplayer game without needing other humans.
    - For some games, such as fighting games, they're meant to teach the player how to play and improve their skills.
    - Generally, all human players can be replaced by computerized players.

3. What design choices exist for the Interface components? Colour? Font? Dimensions of Windows? Rescale-ability? Scroll Bars? ….
    - The UI for the game must be simple for the user to choose what they want, and they must be easily acceptable. The window screen must be either set in a "windowed" mode with clear visuals, and be set with a specific width and length based on the size of the game board. That is to be done in the application
    - Colour is important as icons that require more attention can be painted with more "aggressive" colours, while the background and the menus are using "softer" colours. Also, colours will help give a better visual feel of the overall application.
    - Font is also important as it must be clean and not boring (i.e. Comic Sans, Times New Roman) and is part of the overall style of the game. The font must be also a good size for the viewer to be able to read and understand.

- Once the board has been set, the user can resize the window, in a specific aspect ratio according to the game board's size, so both the horizontal axis and the vertical axis will have to be modified at the same time.
- Limiting scrolling is better for making the UI simpler. Scrolling may be helpful within the settings of the game for stacking lists of customization options.
- The importance of showing how the user a transfer from "state to state" is valuable, and can be implemented by visually showing the user where to click to go play the game, choose game settings, or other states that may be added onto the game.

4. What are the advantages and disadvantages of using a visual GUI construction tool? How would you integrate the usage of such a tool into a development process?
    - Advantages
        - Using a GUI construction tool will allow easier implementation of widgets and modules on a window and abstracts the method of "putting pixels to the screen".
    - Disadvantages
        - Can cause a lot of prototypes to be used and never fully implemented. Possibly resulting in wasted time.
        - Could be difficult to create custom GUI components if not already implemented within the GUI construction tool.
        - Prone to more bugs if GUI components not implemented correctly
    - To integrate the usage of such a tool, we should first do storyboarding in a graphics editor program to solidify the design then use the visual GUI construction tool to build our UI to match our storyboard.

5. What does exception handling mean in a GUI system? Can we achieve consistent (error) messaging to the user now that we are using two components (Ruby and GTK3 or other GUI system)? What is the impact of the Ruby/GTK3 interface on exception handling?
    - The Qt framework will likely be used in our scenario. Currently, Qt does not fully support C++ exception handling. Instead Qt uses error codes for exception handling. Luckily we are using Ruby and we can use the language's exception handling tools to create and catch exceptions as necessary. Qt allows us to create dialog boxes and so we can create special exception classes within Ruby which produce dialog boxes

containing the exception message, whenever the exception occurs. This will ultimately allow us to create consistent error messages for the user.

6. Do we require a command-line interface for debugging purposes????? The answer is yes by the way – please explain why.
    - Yes, a command-line interface (CLI) is necessary for debugging. The following points are justifications for using a CLI for debugging purposes:
        - Integrating the debug mode into the GUI is pointless and may lead to more bugs. Placing the debug content in the command-line is less of a burden (in terms of implementation and performance), and the GUI can remain in a state as if it were being “deployed”.
        - The debugger can be hidden from the user if it exists as a CLI.
        - Consider deploying the GUI on a device that uses Linux. If no debug mode exists, we wouldn’t be able to debug the GUI after deployment. If a command-line debug mode did exist, it would be possible to debug the program over an SSH connection.

7. What components do Connect 4 and “OTTO and TOOT” have in common? How do we take advantage of this commonality in our OO design? How do we construct our design to “allow it to be efficiently and effectively extended”?
    a. Both Connect4 and TOOT/OTTO share the commonalities of connecting 4 pieces of a certain type in a specific order. They also share most of the same rules, and game sequence where each play takes turns dropping a piece down a chosen column. The only differences would be the win condition of the game, the pieces used, and the pieces players have available to them. We can take advantage of these similarities by using the Template design pattern to lay out the sequence for each players move (and also the win condition), allowing us to reuse code for the common elements between the two games. To allow this to be efficiently extended we make use of the Template design pattern (described above) which would allow us to easily add new games that are similar to both Connect4 and TOOT/OTTO. We also use the State Design pattern to lay out the steps each game goes through, this allows us to easily extend gameplay elements by adding additional states.

8. What is Model-View-Controller (MVC, this was discussed extensively in CMPUT301)? Illustrate how you have utilized this idea in your solution. That is, use it!

a. MVC will be used in our project to easily separate the major elements of our software into cohesive modules. The view component will be handled by the QT module. The controllers will be implemented by state machines that transition between application states and for the game state, executes the game algorithm . The model in our program will act as the data layer and will be responsible for storing current game data and settings.

9. Different articles describe MVC differently; are you using pattern Composite?, Observer?, Strategy? How are your views and controllers organized? What is your working definition of MVC?
    - The Qt framework offers their own MVC template for building GUIs:
        - https://doc.qt.io/archives/qt-4.8/model-view-programming.html
    - Qt's MVC perspective is targeted more towards reusing existing models and views in their library such as the QAbstractTableModel, and it's corresponding QTableView.. These implemented views limit the nature of our board-game and are really only effective in C++. Since we have Ruby at our disposal, it will be easier to create our own MVC, tailored for board games, using the Observer pattern and combining it with Qt's signal/slot mechanism. This will ultimately increase extensibility as well.
    - In our case, models will hold the data of game components. This includes the game score, game state, player settings, and board settings (eg. Colors).
    - The view displays the model's data on the screen. This corresponds to widgets and windows in our program.
    - The controllers are what modify a model's data. These can be viewable on the screen eg. Buttons), or non viewable (eg. Keyboard, and mouse).
    - We would like to create views and controllers in separate classes, however in circumstances where views and controllers are closely coupled (for example: Excel spreadsheet cell), using a single class with a composition of the view and controller would be preferred.

10. Classes start life on CRC cards or a competing notation. Provide a full set of CRC cards produced by your group for this problem. These cards must be supplied as part of part 1.
    - See the docs/Assignment 4 CRC.pdf for the CRC cards.

11. Iterators – are they required for this problem? Fully explain your answer!
    - Iterators are important to our scenario. Since we are creating a board games, the algorithm of each game will be based off the components on

the board. Being able to iterate over each “tile” on the board will allow us to compute the winner more easily. Moreover, modifying the game board will be easier to accomplish iteratively; these modifications include cleaning up the board, rotating the board, or even animating the game itself.

12. Explain your strategy for testing the GUI component of the system. How are you going to test that your system is Usable?
    - Blackbox testing:
        - Checkbox - Run through the functionality of the GUI and make sure each feature is implemented.
        - Monkey testing/dirty testing - Attempt to discover bugs by testing with random, possible erroneous, input and observing the output.
    - White box testing:
        - Integration testing - Between the model, view and controller classes.

13. Colour (color) blindness (colour vision deficiency, or CVD) affects approximately 1 in 12 men (8%) and 1 in 200 women in the world. In Canada this means that there are approximately 1.5 million colour blind people (about 4.5% of the entire population), most of whom are male. Explain how your design accommodates this user group and how you will test this accommodation to ensure that the final implementation meets this objective
    - We’ll need to be able to change all UI colors to accomodate for color blindness. This means that all colors should be dynamic and easily modifiable through themes.
    - To ensure the final implementation meets this objective:
        - The ideal scenario would involve surveying people with CVD and testing whether they’re able to differentiate between all colors.
        - Otherwise, we’ll have to research common colors used by other applications to address certain CVDs and ensure only those colors are used.