ECE 421 TEAM PROJECT #3: Let’s Make a Rusty Connect-4

Submitted by Group: 8

Date submitted: April 10th 2022

Team Members	**Student #s**	**e-Signatures**
Dale Richmond Naviza	**1520045**	[illegible]
Anuj Pradeep	**1578030**	Anuj P
Taranjot Singh	**1543004**	[illegible]
Ralph Milford	**1534131**	[illegible]

TABLE OF CONTENTS

I. **Major Innovations:** Additional to the Project Specification

In addition to the requirements stated in the project description our team implemented a number of additional features. These features were not requested as part of the original request and were of our own design.

1) Credentials and login:
   Each player now has a corresponding password to their account. This is to ensure the safety of each individual's account. In order for a user to play either of the designed games they are required to sign in or sign up. In order to improve security further each password is encrypted using the Argon2 crate.
   We implemented this feature to protect our user's identity. Specifically to ensure that a player's account would not be compromised and therefore their records tainted. This may seem a bit excessive for the low priority of board games but we wish for our competitive players to feel safe that their hard earned victories are protected.

2) Colour Blindness considerations:
   In order to ensure ease of use by individuals who are visually compared, especially with regards to color, we implemented a number of design decisions. Specifically, we included labeling to our game pieces. These simple labels ensure that anyone who is having trouble differentiating the two sides have a way of clearly identifying player one and player two.
   We implemented this feature to ensure that our product is inclusive to the visually impaired. This way a disability does not take away their enjoyment of a game they purchased. Additionally, by being more inclusive our potential audience also grows. Though first and foremost we implemented these designs simply to help people enjoy our game.

3) Clearing the Database:
   We have given the option for players to clear all records inside the database. This would include the list of players as well as every game played and saved. Once the database had been cleared the automatic computer player would be added back to the database to ensure that people can still play against an opponent.
   Our choice to implement this functionality was a practical one. As more and more players are being added, as more and more games are being played, the database would start to swell. This swelling would eventually affect the entire performance of the game. Querying the database would take longer and slow processes such as logging in and saving games. By clearing the database this query lag would be prevented.

4) Delete:

We have given the ability to delete either a game or an entire player from the database to our players. In order to delete a game a user would require the game's ID. While in order to delete a player a user would require both the player's name and the player's password. The database would then be queried and the given game or player would be removed.
Similar to the previous feature, part of this implementation is to help performance. If a player has chosen not to play any further this would free up space for those who will. However, this was also implemented to remove upsetting or controversial occurrences. Whether that's a problem player or a contentious game. This way our game won't be a reminder of an unpleasant experience and instead be a sign of joy.

II. **Design Decisions:** With Regards to the Design Questions

When originally planning this project we considered a number of questions. Each of these questions had us considering how we would implement different parts of the project. How and why we went with certain decisions and not others. Asking ourselves these questions had us truly consider our methods and end goals. These are those questions:

1) What can we do on a computer that we can't do on a printed board?
   Now this is an important question to ask. Afterall, why fix something if it isn't broken right? However, there are several benefits switching from the classic printed board to our digital system. First and foremost is the ability to check the history of the games played. Going further for those who are extra competitive we incorporated a leaderboard based upon the number of wins. With a printed board such history would be reliant on the memories of the players, however now checking just how many times who bested your friends is just a button click away.
   Another benefit for going the digital route over the printed is customizability. A printed board cannot be changed easily or cheaply while a digital board can. From changing the colors of the pieces to incorporating modes for the visually impaired. Such options are at the tips of the user's fingertips when using a digital game.
   Overall, there are many benefits from going the digital route over the classic physical way. Pieces cannot be broken or lost, no risks to small children, and a much lower chance of an accident ruining the game. All these positives and more demonstrate the viability of this digital board game.

2) What is a computerized opponent? What are its objectives?
   A computerized opponent is when you are facing an artificial intelligence instead of a regular person. In other words you are playing against the computer itself rather than another human being. However, computers are obviously not human beings and

therefore have to be instructed on how to play the game. A computerized opponent's objective is not to necessarily win. Given enough training and programming a computerized opponent can be taught to never lose to a human, the pure amount of calculations and analysis they are capable of makes it impossible for us to compete. As such, winning is not the number one priority when developing an opponent for a game. Instead, a computer opponent's objective is to make the game fun and interesting for the user. This way they ensure the user continues to use the system and might buy more products in the future. However, this does not mean that the game cannot be challenging. An opponent that lets the user win every single time no matter their actions would result in the user quickly losing interest and respect for the game. So the computer does try to win against the user, what changes is the competence of the computer. By limiting choices, calculations and analysis we limit the overall opponent, taking them down from unbeatable to a challenge. A challenge that can be scaled. For different levels of difficulties there will be different opponents, each with their own limitations. The higher the difficulty the less amount of restrictions put onto the computer, while the lower the difficulty more restrictions are put onto the computer. Overall, a computerized opponent is designed to make the game fun for everyone. Whether that's by being a challenge, being a pushover, or simply being an opponent.

3) What design choices exist for the Interface components?
   There are many design considerations to take into account when designing an interface. Of course there is the aesthetics to consider. However, that does not mean simply making the game pretty to look at. Instead the aesthetics are a balancing act. They have to be pleasing so as to not put off the user but they also cannot be distracting. If the aesthetics are distracting then that's worse than if they weren't there at all. It's important to remember that this product is a game first and foremost. If a user cannot enjoy playing the game then even the most brilliant of aesthetics are useless.
   Then there are limitations to take into account for the UI. This can be both technological but it can also be biological. Color blindness, impaired hearing or other medical conditions have to be taken into consideration as well. These people deserve to get just as much enjoyment out of their purchase as anyone else. There are multiple ways to take these people into consideration when designing a UI. For example: despite offering multiple colors to players our game also labels whose piece is whose. This way a color blind individual can still keep track of whose piece is whose. There are also technical limitations to take into consideration. Not everyone's hardware will be able to handle the same amount. If a UI looks amazing but slows the game down to a slog then the player will only be annoyed by it. As such, a UI has to be efficient and optimized to ensure the least amount of impact on performance as possible. This can be achieved in a variety of ways such as lowering resolution or less complicated designs.

Overall, there are many considerations to consider when designing an interface. Color, font, dimensions, scalability are all tools for developers to use. Tools to use to ensure that the user, the player, enjoys playing the game.

4) What does exception handling mean in a GUI system?
Unfortunately, no system is absolutely perfect. Things will go wrong for a variety of different reasons. Such as: errors in the code, unexpected user inputs, and unexpected technical occurrences. The point of exception handling is as the name implies to handle errors when they occur. They are the safety blanket of the entire system, so that when something goes wrong the entire system does not come crashing down.
In a GUI system this is even harder to implement than normal. Once again this difficulty is for a variety of reasons. A GUI takes in user inputs which opens it up for unexpected or even incorrect user input. Therefore, more error handling is required. So that when input is incorrect it can be caught and flagged before it becomes an issue for the whole system. A GUI is also a visual representation of the system and such must be changed to represent that something went wrong. This can be done in a variety of ways. From something as simple as a warning notification to an entire page dedicated to announcing the problem. Either way error handling in a GUI system also requires keeping the user in the loop, so that if they are the issue it doesn't repeat itself. Finally, an error might not be caused by the program itself. Often a program is reliant on a third party resource such as a database or an API. Therefore, the system has to treat an error from them as if it's one their own. The program has to keep the system and the user safe from such a scenario. For example: if a connection cannot be established between a server and the system then the GUI should reflect that status. This could be in the form of having any functions related to that server grayed out/unavailable so that the user can't accidentally cause a crash.
Overall, error handling is an important feature in all of coding. However, for a GUI it is of particular importance that it is implemented thoroughly. Due to a GUI directly interacting with a user, whose actions can dictate the system. Having error handling ensures an error can be dealt with cleanly, efficiently and most importantly clearly. Not all users will have a perfect understanding of the whole system. Therefore, it's important that the error handling is clear about what went wrong and why it went wrong. This way the user has the knowledge on both what to do and what not to do.

5) Do we require a command-line interface for debugging?
Absolutely yes, a command-line is instrumental when debugging the system. The value of a command line cannot be understated. There are dozens of reasons why having access to the command line is vital for effectively debugging a program. Especially when the system in question is trying to implement a GUI such as our case.

Command-line interfaces are basically what allows developers to check the results of a system. When debugging, relying solely on an implemented GUI is not good enough. Mostly because that GUI is also part of that debugging process. The GUI needs to be checked to ensure that it isn't the source of a problem such as an incorrect value or placement. Therefore, without a command-line there would be no way of double checking this data.

However, the command line has more uses than simply double checking results. It can also be used for checkpoints. By printing to the command line (a far simpler task than printing to a GUI) the process a system takes can be mapped out. This allows developers to know exactly where an issue is occurring.

The command line is also where error statements are typically printed out. Without such statements many developers would be at a complete loss at why their programs failed. Often an error statement will include the file, line number and even digit of where the error is occurring. As well as including the exact type of error that happened and offering suggestions of possible solutions.

Finally, there is the ease of navigation that a command line brings. With decent knowledge a developer can navigate through a system's files much quicker than using the actual file system. This ease and quickness simply improves the overall efficiency of debugging. As the developers aren't wasting time hoping from folder to folder and sitting around doing nothing.

Overall, the command-line interface is one of the strongest tools in a developer's toolkit. It allows them not to only operate efficiently but also opens up numerous options for them. In our case, the command-line interface allowed us to double check positions when we're printing out the board. Double check that the correct results are being sent to the database. That the correct process is being followed when a game is being played. As well as numerous and numerous other examples. There is without a doubt that a command-line interface is required.

6) What are Model-View-Controller and Model-View-Viewmodel? Are either applicable to your design?

Model-View-Controller and Modevl-View-Viewmodels are software design patterns that are used in developing user interfaces. The implementation of Yew, which very closely resembles React, is closer to a Model-View-Viewmodel design than it is to a Model-View-Controller design.

With respect to the internal representation of the game, we store a game_state string which holds the internal representation of the game board. We did this in order to parallelize the work to do on this project. Part of our initial plans also included saving game progress in the database and storing the game state as a simple string could have

helped us do that in a more efficient way. This game string, representing the model, is then converted into a table which is the view. Clicking on columns in the game corresponds to signals being passed between the Model, View, and Viewmodel.
As such our project is very much a Model-View-Viewmodel design. Users directly interact with the view which is how input is handled. The backend, the model, is also only responsible for data handling, specifically sending and retrieving data from the database. This is more in line with the description of how a Model-View-Viewmodel handles the model than how a Model-View-Controller does. Overall, our project shares much more characteristics of a Model-View-Viewmodel design than it does of a Model-View-Controller.

III. **List of Known Issues:** Including Errors, Faults, Defects, and Missing Functionality

Unfortunately, our project is not without issues. Here we explain the errors, faults and missing functionalities that we are aware about. Additionally, we will attempt to explain how we would implement/fix these issues if circumstances were different.

1) First and definitely foremost is our issue between our frontend and our backend. To be more precise it's an issue with the communication between the backend, our database, and the frontend, our game. We were having an extremely difficult time having the two to properly communicate. Specifically when it came to having Rocket and Yew interacting with each other. We just could not achieve getting the two of them to properly talk to each other. Which resulted in our game not being able to access the database at all. Our issues are primarily due to Yew no longer supporting its Services crate. Which we would have used in order to properly fetch the required data. Without that support and without any sufficient examples or documentation we couldn't find another way to implement the connection between the frontend and the backend. Though attempts at finding an alternative solution have been met with similar results. We have tried to skip using Rocket and simply use Yew to directly query the database. This also failed due to the sqlite crate and Yew being incompatible, with the sqlite crate being unable to build when used in the frontend. Our next solution was trying to find different ways to call Rocket and therefore the database. We tried to implement: callbacks, get requests, and even the previously mentioned fetch requests. Each was met with failure due to a variety of reasons, such as certain crates not being designed for Web Assembly or having to incorporate asynchronous behaviour. We have included both the completed front end and the completed backend. Both of which work fine on their own, it is simply their connection which is broken. However, we still fail at the requirement of using Rocket for the backend and we fully accept that.

If we were to try and fix the communication between Rocket and Yew it would be through the use of a Worker file. As the name implies, a worker file would include the functions that would be actually fetching the data from Rocket. This would be at the behest of Yew (note: worker would be written in Yew as well) through sending a message (Msg). The worker file would respond differently depending on which message the frontend had sent it. This would be implemented by using an Enum to cover all potential requests and any parameters they require. Depending on the Enum the worker would call a certain function which would fetch the data from Rocket. Rocket would then query the database before sending the desired data to the worker. The worker would receive the data and new functions would trigger through the worker's update. A response would be built up and of course include the data recently retrieved. No matter what type of query was being sent it would return a Result wrapping a String or an Error. Therefore it was up to the worker to send an appropriate response to the frontend in order for the data to be treated correctly. Similar to the worker the frontend would also contain an Enum of potential requests. The frontend's update would be triggered and one of those Enum's would be chosen. With that frontend would start to process the data and update the game. Overall, the process should have gone like: Frontend → Worker → Rocket → Database → Worker → Frontend. With each step of the way dictated by the original request the frontend had sent to the worker.

Because of this database limitation, we decided to put in placeholder data for the game history and the scoreboard page.

2) For the text boxes, the inputs have to be manually typed in; autocomplete will temporarily fill in data in the text box but it will not be a saved state. Additionally, backspacing will result in temporarily removing text, but the 'erased' letter will still be saved in the text box's state.

3) The Medium and Hard versions of the computerized opponent run for a very very long time; however, the 'Easy' version of the opponent is still pretty challenging.

IV. **Remaining MEAN Stack Code:** A Detailed Description

We started the project from scratch so there is no MEAN stack code in our project. The backend is purely in rocket; the frontend is purely in yew.

V. **User Manual:** Complete Guide to Operating the System

This is designed to be a guide for new users. So that they understand not simply how to play the game but also how important each step in the setup process is. With even one requirement missed there's a high potential for the game to simply not function.

Frontend:
Assuming that the user already has cargo installed, they can run 'cargo install --locked trunk' to install trunk. After downloading the project folder, go into the frontend/ folder and run 'trunk serve' in the command line. The website should run on http://localhost:8080/

Backend:
The backend was focused around the implementation of the database. Therefore, first and foremost is to ensure that the database, gp3.db, exists. It should be located in the data folder located in the root directory of the program. If the database no longer exists then a new one must be created before the game can be played. Going further, the two tables, Players and Games, have to be re-established. Using sqlite3 open the database (sqlite3 gp3.db), then use the command ".read tables.sql;". This will automatically recreate the tables inside of the database. Afterwards, close sqlite3 using ".exit" and start up the game. First and foremost the database clear function has to be called. This is done in order to re-add the Computer as a valid player in the Players table. Then the user will have to re-add themselves as well using the new player function. Finally, the player is free to play the game.

It is key for players to understand how important the database is to the entire game. Without the database then no games could be played at all. Players would not be able to login in to play any games and unable to make a new account either. Without being able to access a valid player the program would not allow a new game to start. Therefore, users should always ensure that the database is kept tracked off and safe.

VI. Sources:

Yew.rs
Rust Programming Language
Rocket.rs
Model–view–controller - Wikipedia
Model–view–viewmodel - Wikipedia