Design Documentation

The purpose of this document is to properly cite my resources and reasoning for my additions:

Why did I choose an ATM?

An ATM is indeed a weird thing to choose to reproduce in a UI class. Originally my goal was to make a Trading View Interface for stocks because I am really interested in making something like this. Additionally, it was a focus for my previous class in Neural Networks where I made an algorithm for trading stocks which was very successful. Furthermore, I am currently in the interview process for companies like Citadel, Jane Street, Hudson River Trading, ArrowStreet, and Citibank. This project would allow me a bigger talking point in these interviews. I know it's very capitalistic and not creative for my personal use, but working at these firms allow me to accomplish some of my personal goals.

I wanted to push this theme further by making a trading view and being able to implement an algorithm from Python to JavaFX because I thought it would be possible. I did a good amount of research and realized it would be unrealistic to finish the project within the given timeline especially without the use of SceneBuilder, which is what I saw a lot of people using to make similar applications. Finally, I came to the conclusion that if I couldn't get to build what I wanted to build, then I would make something that can be used as a precursor.

That is when the idea of an ATM came in because I could recycle the code into a final project where I could push the project to become a trading viewer. I further decided, that I would make this first project coordinated with the Colgate Theme which made it more of separate piece rather than just feeling like an incomplete project. It was also made to to practice a lot of the User Design principles that I had been reading up on, in class. One of my goals was to use the login screen as pretty much the same thing. Another one of my goals, were to add and take out money, so the deposit and withdraw features would be exactly the same.

Elements used

I chose this because I wanted to practice a lot of the other features that JavaFX had to offer because there were so many that I just had no idea about. I did a lot of research into the documentation as well as just looking around at the different things I could use to meet my needs of a project. Some of these things were RadioButtons, CSS Styling in Java, Text Formatting, and Button formatting

Previous Experience

I didn't really have too much previous experience in JavaFX, but I did have experience working with web development before. I have used the react framework, and have built apps on the MERN stack. I also have lots of experience on the processing framework which had a lot of similar built-in features. A lot of the concepts in web development transferred over as skills here, so I was able to pick up a lot of the concepts quickly.

Design Inspiration

Obviously an ATM is something I have used variously throughout my life, so it is something that I am very accustomed to.

However, this link from MIT, really helped break things down for me mentally: MIT Source I use a variety of Bank ATMs to look up as inspiration but the BoFA ATM was one I took inspiration for a lot.

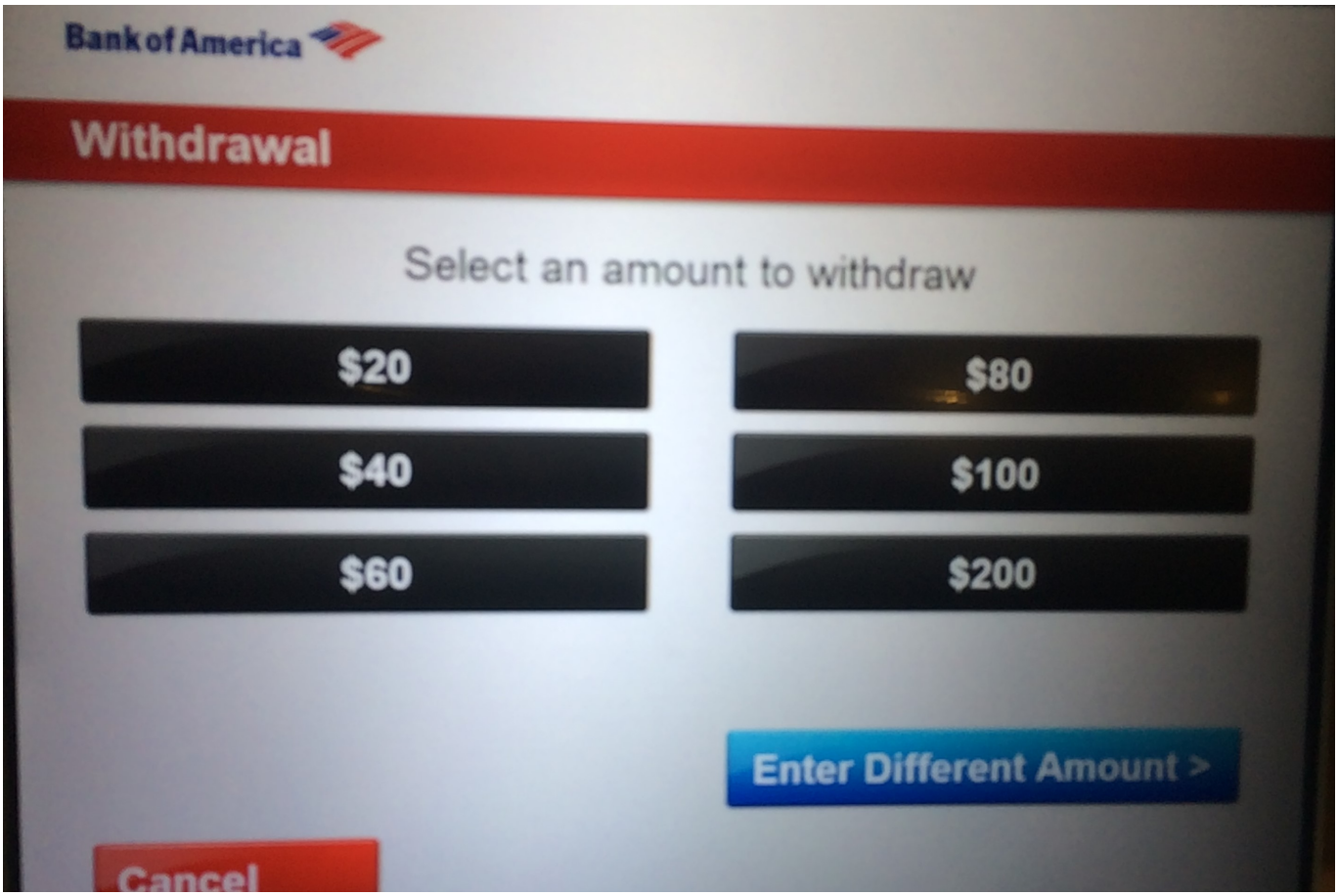
Panes

General

Generally I simply took inspiration from the MIT Source and my previous knowledge of how ATMs look and act like and tried my best to implement that into an application.

Withdraw Pane

I took inspiration from the MIT source mentioned earlier. I am using Radio Buttons to select the amount of money to withdraw. I also offer the user a chance to enter a custom amount. I also added a back button to go back to the main menu. This is the exact MIT source picture I looked at to determine what I was going to do:



Radio Buttons for me were used as a replacement for the buttons on the side of screens when using ATMs. This created an interactive experience for the user as a way to replicate the same experience, there was also hovering built-in for this, unless I accidentally implemented this portion.

To handle which radio button was selected, I used a ToggleGroup. This allowed me to make sure that only one radio button was selected at a time. This was done by using the selectedToggleProperty method and retrieve by using .get/setToggleGroup. This was further done through a lambda function.

Source:

Java2s Although I don't state this resource often in this document, I used it a lot to help me understand how to use JavaFX. I would recommend it to anyone who is trying to learn JavaFX. Java2s has a quick way of viewing libraries and methods that are available to you.

- This wasn't the only resource I used, but it was one of the most helpful.

However, I didn't really know how to manage this properly and I came up with a temporary solution of a state management one where I would check the first few letters of the radio button selected when trying to access these numbers. The main problem with my temporary solution is that since I am using the substring method to check for which radio button I am accessing, I can't retrieve the $1000 button, because it shares the same first few letters as the $100 button. This is a problem that I will have to fix in the future.

Source Code:

```
    final ToggleGroup group = new ToggleGroup();

    RadioButton rb1 = new RadioButton("A");
    rb1.setToggleGroup(group);
    rb1.setUserData("A");

    RadioButton rb2 = new RadioButton("B");
    rb2.setToggleGroup(group);
    rb2.setUserData("B");

    RadioButton rb3 = new RadioButton("C");
    rb3.setToggleGroup(group);
    rb3.setUserData("C");

    group.selectedToggleProperty().addListener(new ChangeListener<Toggle>
() {
      public void changed(ObservableValue<? extends Toggle> ov,
          Toggle old_toggle, Toggle new_toggle) {
        if (group.getSelectedToggle() != null) {

System.out.println(group.getSelectedToggle().getUserData().toString());
        }
      }
    });
```

The code I implemented:

```
final ToggleGroup buttonGrouping = new ToggleGroup();
...
...
        // listens if radio buttons are selected or not to whether disable
withdrawButton or not
        // also includes the "Other Amount" choice to keep the button
```

```
disabled until user typed amount in textfield
        buttonGrouping.selectedToggleProperty().addListener((ob, o, n) ->
{
            if (otherAmount.isSelected()) {
                withdrawButton.disableProperty().bind(

Bindings.isEmpty(otherAmountTxtField.textProperty()));
            } else {
                // makes sure that button unbinds to the textfield if
other choice is selected
                withdrawButton.disableProperty().unbind();
                if (withdrawButton.isDisabled()) {
                    withdrawButton.setDisable(false);
                }
            }
        });

                withdrawButton.setOnAction(event -> {
            // basically gets the text of the choice, if it's "other
amount", gets the input from textfield
            RadioButton selectedRadioButton = (RadioButton)
buttonGrouping.getSelectedToggle();
            String toggledButtonAmount = selectedRadioButton.getText();
            if (toggledButtonAmount.equals(otherAmount.getText())) {
                withdrawButtonEvent(otherAmountTxtField.getText());
            } else {
                withdrawButtonEvent(toggledButtonAmount);
            }
        });
```

Radio Button Source:

Radio Button Youtube Tutorial

Pane Switching

What I essentially did was that I had eventHandlers created at the end of the Scenes.java file. These eventHandlers would be called when the user clicked on a button. At first, these eventHandlers would then call the setScene method from the main class. That was causing issues that I couldn't figure out, so I changed my approach to something I learned in processing where I just make the method return a draw method (in this case - Scene) and call upon that when needing to switch. This would then change the scene to the desired scene.

Source:

Youtube This video is what I tried to replicate at first, and it did work for some parts but not all. I still used this for my login screen and reciept window, but I had to change my approach for the rest of the scenes.

Hovering

My code hovers between a lighter version and a darker version of its button and determines whether the user is hovering over the button or not. This is done by using the mouseEntered and mouseExited methods.

Source used to learn how to do this:

StackOverflow

Exact Code I learned from

```
final String IDLE_BUTTON_STYLE = "-fx-background-color: transparent;";
final String HOVERED_BUTTON_STYLE = "-fx-background-color: -fx-shadow-
highlight-color, -fx-outer-border, -fx-inner-border, -fx-body-color;";


button.setStyle(IDLE_BUTTON_STYLE);
button.setOnMouseEntered(e -> button.setStyle(HOVERED_BUTTON_STYLE));
button.setOnMouseExited(e -> button.setStyle(IDLE_BUTTON_STYLE));
```

What I implemented

```
    /**
     * This method sets the hover effect for the button.
     * It changes the button color between light gray and dark gray
     * when the mouse is hovered over it.
     *
     * @param button This is the parameter for the back button when it is
hovered upon
     */
    void buttonHoverEffectGray(Button button) {
        button.setOnMouseEntered(e -> backButton.setStyle("-fx-background-
color: #D2D4D6;"));
        button.setStyle("-fx-background-color:#5A646E;");
        button.setOnMouseExited(e -> backButton.setStyle("-fx-background-
color: #5A646E;"));
    }

    /**
     * This method sets the hover effect for the button.
     * It changes the button color between maroon and maple red
     * when the mouse is hovered over it.
     *
     * @param button This is the parameter for the back button when it is
hovered upon
     */
    void buttonHoverEffectRed(Button button) {
        button.setOnMouseEntered(e -> button.setStyle("-fx-background-
color: #E10028;"));
        button.setStyle("-fx-background-color:#821019;");
        button.setOnMouseExited(e -> button.setStyle("-fx-background-
```

```
color: #821019;"));
    }
```

Invalidating Panes

This was a technique I learned from a video made by Oracle about the usage of CSS Styling in JavaFX.

Source used to learn how to do this:

Oracle Video TimeStamp = 1:06:42 This Oracle video is something I thought I would originally use, because I thought I would be able to use CSS styling to make my project look better. However, I just went with using in-line styling instead within JavaFX, but this still helped a little bit in other aspects like these.

What I implemented

```
        mainBorderPane.setTop(headerStackPane);
        mainBorderPane.setCenter(loginForm);
        mainBorderPane.setLeft(null);
        mainBorderPane.setRight(null);
```

State Management

This is something I learned to do when I learned react, and learned to use in-language where state management isn't fully supported in p5.js. I then learned how to do this in JavaFX from a video made by Oracle. I did learn about the switch and case methodology when watching a video on it in Python a long time ago, which stuck in my head.

Source used to learn how to do this:

I did see it being used here which allowed me to understand how to use it in my own project: Oracle Video TimeStamp = 47:08

What I implemented

```
/*
         * Enhanced Switch-state management for processing
         * - type is the String value that determines what type of message
to be displayed
         * - type can be either "process", "transactionSuccess",
"printSuccess"
         * The elements will be added in centerPane
         */
        switch (type) {
            case "process" -> {
                messageDisplay.setText("Processing...");

                // Add elements to centerPane
```

```
                centerContentVBox.getChildren().clear(); // clear first
before adding designated element
                centerContentVBox.getChildren().addAll(messageDisplay);
            }
            case "transactionSuccess" -> {
                messageDisplay.setText("Transaction Successful.");

                // Add elements to centerPane
                centerContentVBox.getChildren().clear(); // clear first
before adding designated element
                centerContentVBox.getChildren().addAll(messageDisplay,
messageLogOffAutomatically);
            }
            case "printSuccess" -> {
                messageDisplay.setText("Printing Successful.");

                // Add elements to centerPane
                centerContentVBox.getChildren().clear(); // clear first
before adding designated element
                centerContentVBox.getChildren().addAll(messageDisplay,
messageLogOffAutomatically);
            }
            case "registerSuccess" -> {
                pleaseRememberMessage.setFont(biggerBodyFont);
                displayAccountNum.setFont(biggerBodyFont);
                messageDisplay.setText("Thank you for registering with
Colgate");
                messageLogOffAutomatically.setText(" ");
                displayAccountNum.setText(BankDatabase.loggedAccount);

                // Add elements to centerPane
                centerContentVBox.getChildren().clear(); // clear first
before adding designated element
                centerContentVBox.getChildren().addAll(messageDisplay,
pleaseRememberMessage, displayAccountNum, loginFormButton);
            }
        }
```

Data Structures

For the account database I simply just used a Hashmap

- One with a string and a string, to input the sample accounts
- Another with a string and a BankAccount object, to input the accounts that the user creates

Icon

This was very simple, I just downloaded it from Colgate's website and used it as my icon for my project.
Colgate Logo

Source:

StackOverflow Tutorial

TableView

Malcolm gave me some pointers and told me which document to look at which was made by you.

I replaced my center pane within the Receipt Window with just labels to a table view to fit the requirements of the project.

Cursor

This was something pretty small to implement, but I did learn how to do it from simply the Java Documentation by Oracle. It was a cool thing to add that made it feel more snappy and responsive.

Source:

Oracle Documentation

Input Filtering

This was something that I learned thanks to just messing around and doing trial and error until it worked. However, I still looked into it a good amount, but I don't fully understand how I did it.

I ended up using this in multiple parts of my code. Mainly in areas where I only wanted the user to input integers or floats, which was quite a few considering its a bank application with deposit and withdraw.

Sources:

StackOverflow Java Documentation Java Documentation2

Code I used from Source

```
UnaryOperator<Change> integerFilter = change -> {
    String newText = change.getControlNewText();
    if (newText.matches("-?([1-9][0-9]*)?")) {
        return change;
    }
    return null;
};

myNumericField.setTextFormatter(
    new TextFormatter<Integer>(new IntegerStringConverter(), 0,
integerFilter));
```

Implementation:

```
    /**
     * This UnaryOperator, named 'digitFilter', checks if the user input
```

```
contains only digits.
     * It is useful for restricting text fields to accept only numerical
values.
     */
    UnaryOperator<TextFormatter.Change> digitFilter = change -> {
        String userInput = change.getText();
        if (userInput.matches("[0-9]*")) {
            return change;
        }
        return null;
    };
    /**
     * This UnaryOperator, named 'doubleFilter', checks if the user input
contains only numbers
     * and an optional decimal point. It is useful for restricting text
fields to accept
     * only floating-point values.
     */
    UnaryOperator<TextFormatter.Change> doubleFilter = change -> {
        String userinput = change.getText();
        if (userinput.matches("\\d*\\.?\\d*")) {
            return change;
        }
        return null;
    };
```

IntelliJ IDEA Shortcuts

IntelliJ had the biggest effect on me when using it.

I learned how to use the shortcuts to make my life easier when coding.

It was insanely easy to edit my code and add to it, the plugins enabled on this IDE automatically looked for errors and fixed them for me, or at least told me what was wrong. From there it was a simple google search away. Furthermore, There were several features that were really important:

Auto-Complete

This automatically helped me complete my code, and it was really helpful when I was trying to figure out what to do next. It also helped me figure out what methods I could use for my code. Since the JavaFX packages were directly in my project when using IntelliJ I was able to simply press Command+B to search through class files of the JavaFX Documentation, it saved me lots of time googling

Lambda Conversion

This was definitely my favorite part about IntelliJ, it was so easy to convert my code to use lambdas. I was able to simply highlight the code I wanted to convert and press Command+Option+V, and it would automatically convert it for me. It was so easy and it saved me so much time. It sounds like I am cheating, but this was only possible after I wrote my code normally, and the IDE suggested I write in a Lambda format. I was able to use this to make my code more readable and easier to understand. Furthermore, it helped me

understand when to implement lambda functions rather than just using them when I have seen them in code examples online, or in the textbook. It is important to note that I did use lambdas by myself as well, but I was able to use this feature to help me understand when to use them, when not used in a textbook context.

Searching

This feature is simple yet very underrated because it allowed me to open multiple windows on one screen to edit multiple files at once. I was able to search through my project and find the files I needed to edit.

Other Sources

There were so many other sources I had used to gain understanding, but there were too many to state. Some of the other sources I used were things I looked at but didnt use directly in my code. Some I might have to solve a problem with a line added here and there.