Criterion C: Development

Techniques

In my programming, I have utilized various techniques to ensure my password manager's efficiency, maintainability, and security. I will explain them in further detail in the sections below.

Technique	**Implementation**
Object-oriented programming	Creating classes and objects to organize and encapsulate functionality
Imported Libraries	Importing pre-written code from external sources through import statements
Exception handling	Using try-catch blocks to handle errors and exceptions that may occur during runtime
File input/output	Using classes such as FileReader, BufferedReader, and FileWriter to read from and write to files
Error-checking	Using simple (else-if) and complex selection (nested if) to verify user input
Hashing	Using the jBCrypt library to hash password before storing it
Encryption/decryption	Using a password-based key to encrypt and decrypt sensitive data
Enhanced for loop	For-each loop used to close all open windows
Graphical user interface (GUI)	Using NetBeans and Java Swing to create a user-friendly interface for the program
XML parsing	Using classes such as DocumentBuilderFactory and Element to parse and manipulate XML data
Polymorphism	AddView method takes a String parameter and can be called with different arguments
Multi-threading	Creating a separate thread for the password scan feature

Program Structure

Figure 1 - List of classes and data files

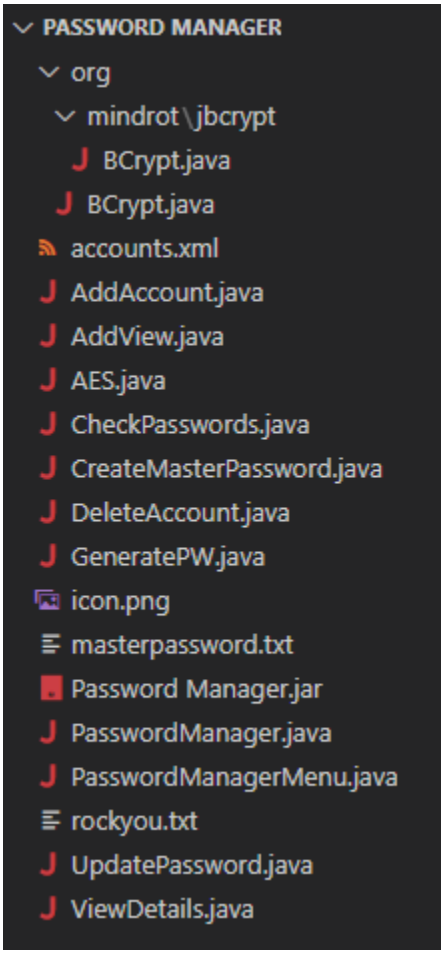
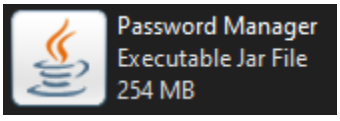
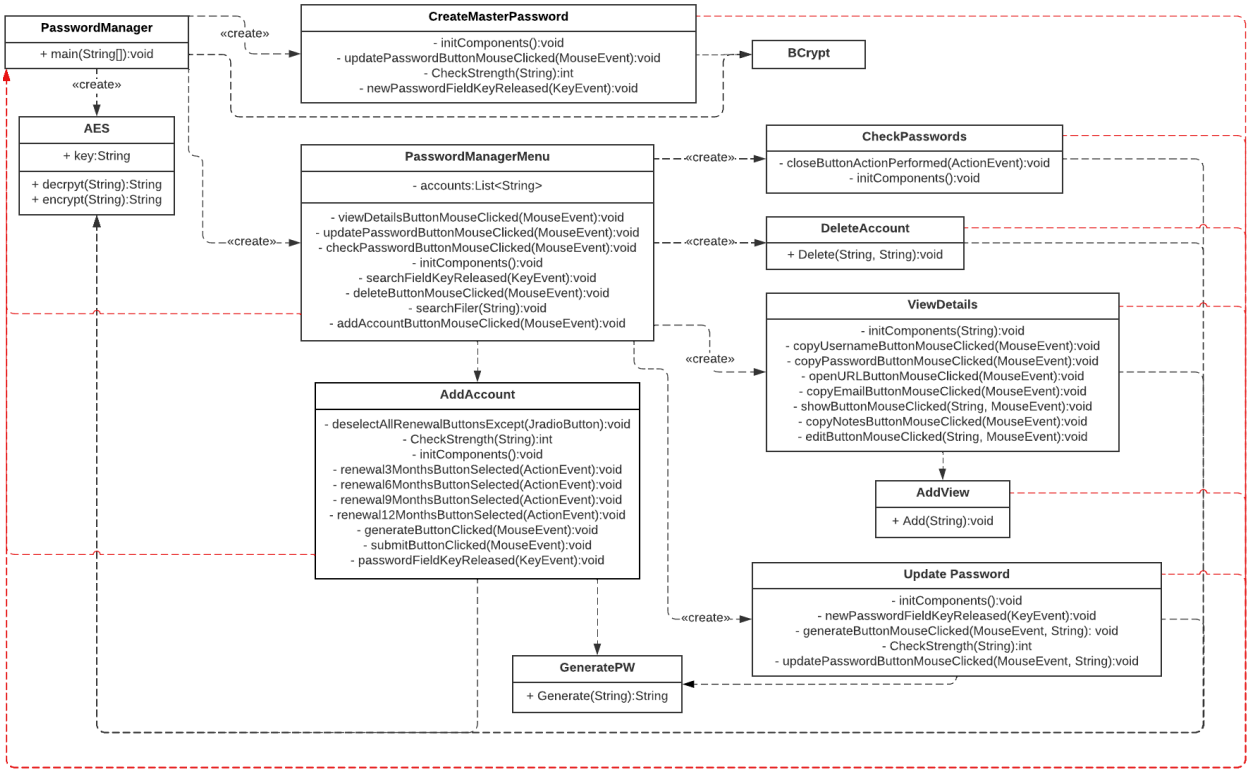


Figure 2 - Executable jar file



Object-oriented programming allows for easier debugging, maintainability, code reuse, modularity, and organization by abstracting the complexity of the password manager's functionality into smaller, more manageable pieces. The classes are divided based on the tasks the password manager needs to perform. For instance, separate files create separate GUI windows.

Figure 3 - System Diagram



As shown in the diagram above, many classes are extensions of the main PasswordManager file, and the properties are inherited.

Imported Libraries

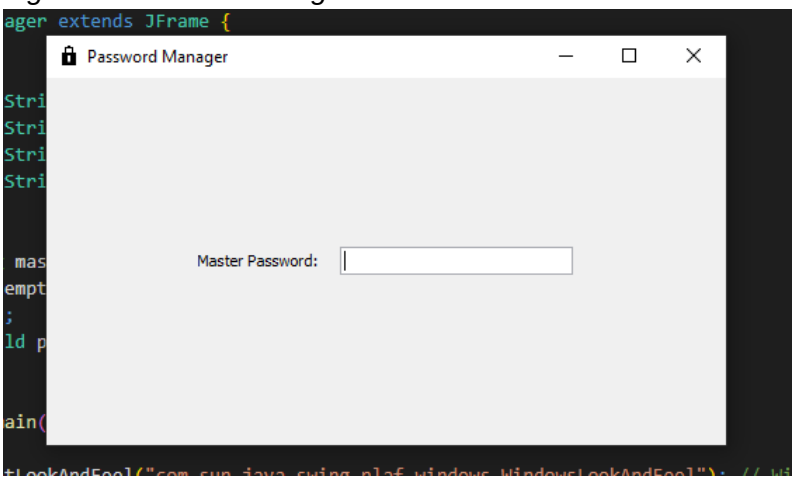
Imported libraries save valuable time and effort. Because I used NetBeans to generate my GUI, I also used the generated AWT and Swing imports.

Library	**Purpose in Program**	**Why Chosen**
jBCrypt	Password hashing and checking	Industry standard for one-way password hashing due to its password salting and slow computational speed, making it difficult for attackers to brute-force
java.io	Reading and writing to files	Simple yet powerful functionality for writing to files (automatically handles buffering)
javax.crypto	Encrypting and decrypting data using AES	Implementation of widely used cryptographic algorithms

java.time	Handle date and time-related operations	Modern API for working with dates and times, with a more intuitive, better performing, and natural syntax than older date/time libraries
org.w3c.dom	Parsing and manipulating XML documents and elements	Standard for XML handling and provides a robust set of features for working with XML documents and elements
java.security.SecureRandom	Generating secure random numbers	Standard Java library for generating secure random numbers, making it a trusted and reliable option

Setting up the Password Manager

Figure 4 - GUI for the login window



Several constants are defined at the beginning of the main class. These are used to specify file names that will be used throughout the program.

Figure 5 - Constants

```
// Constants
public static final String databaseFile = "accounts.xml";
public static final String rockYouFile = "rockyou.txt";
public static final String imageFile = "icon.png";
public static final String masterPasswordFile = "masterpassword.txt";
```

The Windows theme is used for the GUI, giving it a more polished and professional look than other options. **Exception handling** ensures the program works even if the user is not using Windows. Then, the JFrame is initialized using encapsulation.

Figure 6 - Setting the look and feel and then initializing the window

```
// Main Method
public static void main(String[] args) {
    try {
        UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel"); // Windows theme
    } catch (Exception e) {
        e.printStackTrace();
    }

    // Initialize the JFrame
    PasswordManager prompt = new PasswordManager();
    prompt.setTitle("Password Manager");
    prompt.setSize(500, 300);
    prompt.setDefaultCloseOperation(EXIT_ON_CLOSE);
    prompt.setLocationRelativeTo(null);
    prompt.setLayout(new GridBagLayout());
    prompt.setVisible(true);
    prompt.setIconImage(new ImageIcon(imageFile).getImage());
```

The master password is checked by **reading the content of a file** at the specified path. The Path.of() method creates a Path object from the file path string and then the Files.readString() method is used to read the content. Any white space characters in the file content are removed using the replaceAll() method and stored in the masterPassword variable.

Figure 7 - Code for getting the text from the file

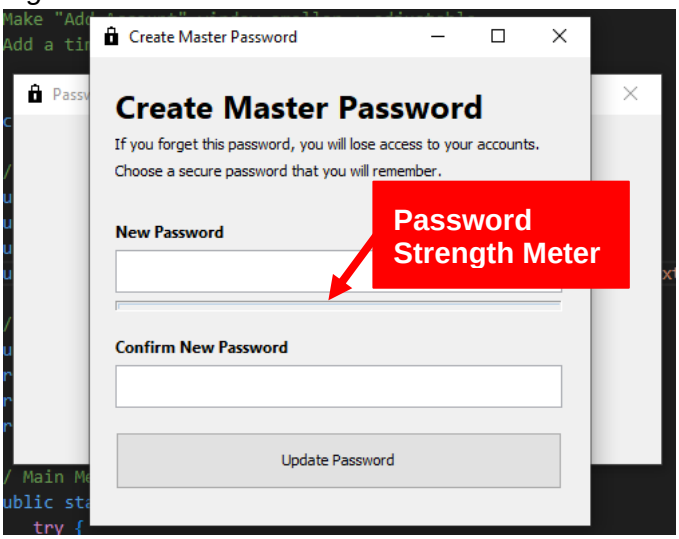
```
// Get BCrypt for master password
try {
    Path filePath = Path.of(masterPasswordFile);
    String content = Files.readString(filePath);
    masterPassword = content.replaceAll("\\s", "");
} catch (Exception e) {
    e.printStackTrace();
}
```

Through **simple selection**, it is checked if the String is empty. If it is, the user has not yet created a master password, so a prompt is given to create one by invoking the CreateMasterPassword() constructor.

Figure 8 - Code for if the master password doesn't exist

```
// If master password does not exist, give prompt to create one
if (masterPassword.length() == 0) {
    new CreateMasterPassword();
}
```

Figure 9 - Window that is created



Instead of using the Windows theme for the password strength meter, I used the Metal UI theme to be able to change the color of the password strength meter based on the score from the **password strength algorithm** (below). The color-changing did not work with the Windows theme, which was a struggle to uncover.

Figure 10 - Password strength algorithm

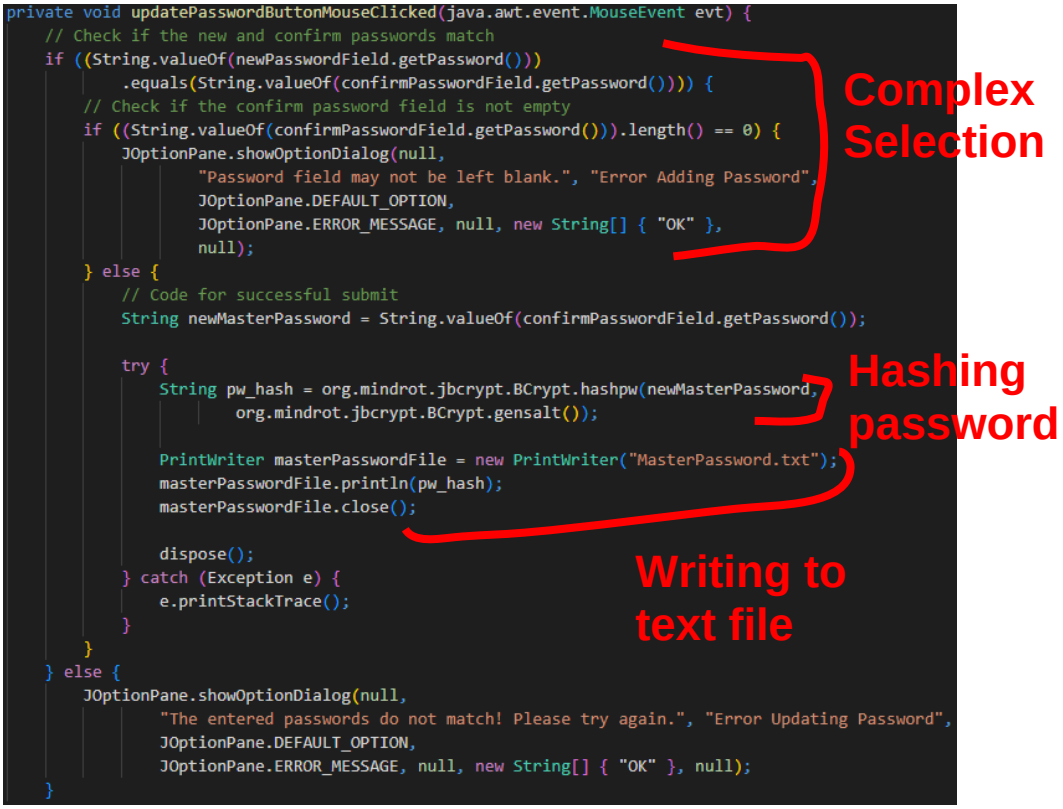
```
// +2 points for each character
for (int p = 0; p < userPassword.length(); p++) {
    if (passwordScore > 48) { // cap at 48 points
    } else {
        passwordScore += 2;
    }
}

// +10 points for three+ digits, +2 points for one
if (userPassword.matches(".*[0-9]{3}.*")) {
    passwordScore += 10;
} else if (userPassword.matches(".*[0-9].*")) {
    passwordScore += 2;
}

// +10 points for three+ lowercase letters, +2 points for one
if (userPassword.matches(".*[a-z]{3}.*")) {
    passwordScore += 10;
} else if (userPassword.matches(".*[a-z].*")) {
    passwordScore += 2;
}

// +10 points for three+ uppercase letters, +2 points for one
if (userPassword.matches(".*[A-Z]{3}.*")) {
    passwordScore += 10;
} else if (userPassword.matches(".*[A-Z].*")) {
    passwordScore += 2;
}

// +20 points for two+ symbols, +2 points for one
if (userPassword.matches(".*[~!@#$%^&*()_-]{2}.*")) {
    passwordScore += 20;
} else if (userPassword.matches(".*[~!@#$%^&*()_-].*")) {
    passwordScore += 2;
}
```

Figure 11 - Code for progress bar color

```
// Update the password strength bar
scoreBar.setValue(passwordScore);

// Change the color of the bar depending on the score
if (passwordScore >= 75) {
    scoreBar.setForeground(Color.GREEN);
} else if (passwordScore >= 45) {
    scoreBar.setForeground(Color.ORANGE);
} else {
    scoreBar.setForeground(Color.RED);
}
```

Once the master password is submitted, the user's entries are error-checked. **Complex selection** makes it so that if the fields are left blank or don't match, the user has to try again (this helps to improve reliability, stability, and usability and provides a better user experience overall). Otherwise, the **hashed** (to ensure it is secure and protected from brute-force attacks) password is **written to the text file** using PrintWriter, a simple and efficient way of writing to a file with automatic data flushing. Finally, the window disposes of.

Figure 12 - Code for master password form submission




```
private void updatePasswordButtonMouseClicked(java.awt.event.MouseEvent evt) {
    // Check if the new and confirm passwords match
    if ((String.valueOf(newPasswordField.getPassword()))
            .equals(String.valueOf(confirmPasswordField.getPassword()))) {
        // Check if the confirm password field is not empty
        if ((String.valueOf(confirmPasswordField.getPassword())).length() == 0) {
            JOptionPane.showOptionDialog(null,
                    "Password field may not be left blank.", "Error Adding Password",
                    JOptionPane.DEFAULT_OPTION,
                    JOptionPane.ERROR_MESSAGE, null, new String[] { "OK" },
                    null);
        } else {
            // Code for successful submit
            String newMasterPassword = String.valueOf(confirmPasswordField.getPassword());

            try {
                String pw_hash = org.mindrot.jbcrypt.BCrypt.hashpw(newMasterPassword,
                        org.mindrot.jbcrypt.BCrypt.gensalt());

                PrintWriter masterPasswordFile = new PrintWriter("MasterPassword.txt");
                masterPasswordFile.println(pw_hash);
                masterPasswordFile.close();

                dispose();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    } else {
        JOptionPane.showOptionDialog(null,
                "The entered passwords do not match! Please try again.", "Error Updating Password",
                JOptionPane.DEFAULT_OPTION,
                JOptionPane.ERROR_MESSAGE, null, new String[] { "OK" }, null);
    }
```

Returning to the login page, when submitted, the program will check that the entered password is correct. If incorrect, an error message displays with a decrement of how many attempts there are remaining. The word "attempts" is corrected to "attempt" when one attempt remains - the minor details aren't overlooked.

Figure 13 - Code for the handling of an incorrect password

```
} else {
    // Handle incorrect password
    loginAttempts++;
    if (loginAttempts == 3) { // Discourages brute-forcing
        dispose();
        JOptionPane.showOptionDialog(null,
                "Whoops! You've had too many failed login attempts.",
                "Incorrect Master Password",
                JOptionPane.DEFAULT_OPTION,
                JOptionPane.ERROR_MESSAGE, null, new String[] { "OK" }, null);
        return;
    }
    String attempts = "attempts";
    if (loginAttempts == 2) {
        attempts = "attempt";
    }
    JOptionPane.showOptionDialog(null,
            "Incorrect Master Password! You have " + (3 - loginAttempts) + " " + attempts
                    + " remaining.",
            "Incorrect Master Password",
            JOptionPane.DEFAULT_OPTION,
            JOptionPane.ERROR_MESSAGE, null, new String[] { "OK" }, null);
    passwordField.setText("");
}
```

After a successful login, a **password-based encryption key** generates. The program iterates through and closes each open window, using a more efficient and easier-to-read **for-each loop** than manual iteration. It then opens the main menu.

Figure 14 - Code for the handling of correct password

```
// Check if password is correct
if (org.mindrot.jbcrypt.BCrypt.checkpw(passwordString, masterPassword)) { // http://www.mindrot.org/projects/jBCrypt/
    // Handle correct password: send password to AES, dispose window, and open
    // PasswordManagerMenu
    AES key = new AES();
    try {
        key.setKey(passwordString);
    } catch (Exception b) {
        b.printStackTrace();
    }

    for (Window window : Window.getWindows()) {
        window.dispose();
    }
    new PasswordManagerMenu(masterPassword);
```

Generate encryption key

Enhanced for loop

Open main menu

The key is encoded in Base64 format and truncated to 32 bytes (256 bits), the length required for AES encryption.

Figure 15 - Code for encryption key generation

```
private static String key;

/**
 * Generates a secret key based on the master password.
 *
 * @param passwordString - the master password
 */
public void setKey(String passwordString) throws Exception {
    byte[] salt = new byte[100];

    KeySpec spec = new PBEKeySpec(passwordString.toCharArray(), salt, 1000, 256);
    SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
    byte[] keyBytes = factory.generateSecret(spec).getEncoded();

    key = Base64.getUrlEncoder().withoutPadding().encodeToString(keyBytes).substring(0, 32);
}
```

PBKDF2 with HMAC SHA-256 (**cryptographic algorithm**) helps prevent attacks that attempt to recover the original master password from the encrypted data. The derived key is unique, unpredictable, and later used in the encryption and decryption methods.

Figure 16 - Code for encryption method

```
/**
 * Encrypts the given plain text using the specified secret key.
 *
 * @param plainText - the plain text to be encrypted
 * @return - the encrypted text as a Base64-encoded string
 */
public static String encrypt(String plainText) {
    try {
        Cipher cipher = Cipher.getInstance("AES");
        SecretKey secretKey = new SecretKeySpec(key.getBytes(), "AES");
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);
        byte[] encryptedData = cipher.doFinal(plainText.getBytes());
        return Base64.getEncoder().encodeToString(encryptedData);
    } catch (Exception e) {
        e.printStackTrace();
        return "";
    }
}
```

Figure 17 - Code for decryption method

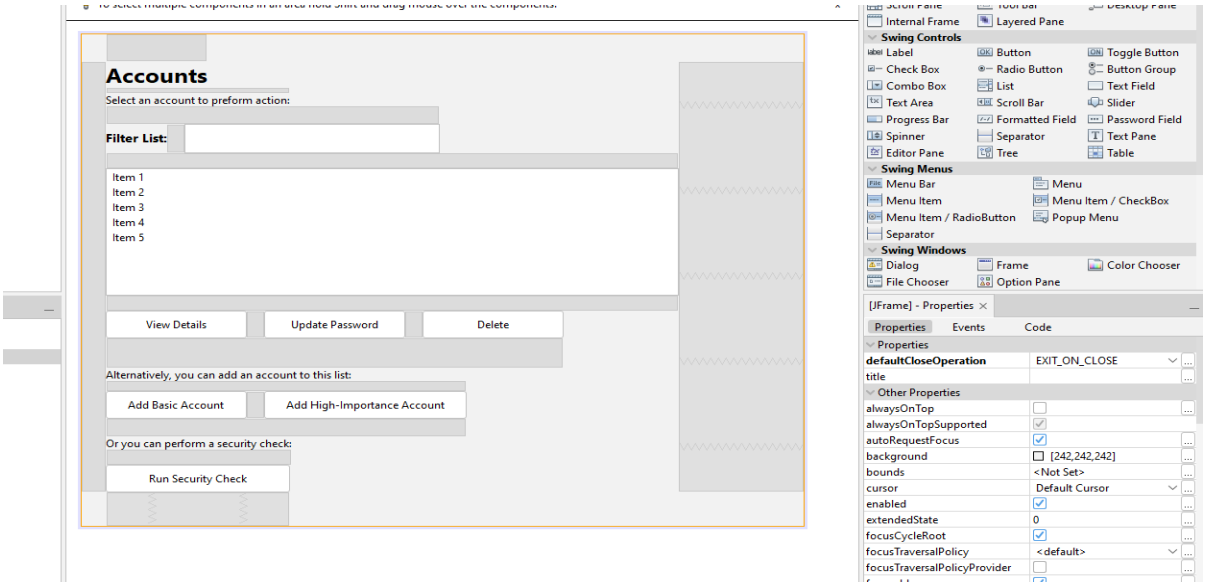
```
/**
 * Decrypts the given encrypted text using the specified secret key.
 *
 * @param encryptedText - the Base64-encoded encrypted text
 * @return - the decrypted text
 */
static int exceptionCount = 0;

public static String decrypt(String encryptedText) {
    try {
        Cipher cipher = Cipher.getInstance("AES");
        SecretKey secretKey = new SecretKeySpec(key.getBytes(), "AES");
        cipher.init(Cipher.DECRYPT_MODE, secretKey);
        byte[] decryptedData = cipher.doFinal(Base64.getDecoder().decode(encryptedText));
        return new String(decryptedData);
    } catch (Exception e) {
        e.printStackTrace();
        exceptionCount++;
        if (exceptionCount == 1) {
            JOptionPane.showOptionDialog(null,
                    "An encryption was likely created under a different master password!", "Error While Decrypting",
                    JOptionPane.DEFAULT_OPTION,
                    JOptionPane.ERROR_MESSAGE, null, new String[] { "OK" }, null);
        }
        return "";
    }
}
```

Exception handling ensures that the program notifies users of any issues, improving the user experience.

Password Manager Menu

The main menu was created using **Apache NetBeans IDE 16**, as were all other **GUIs** except for the login page. I chose to use NetBeans because it allowed me to focus on the usability and features of my product rather than the GUI.

Figure 18 - NetBeans interface



The centerpiece of this menu is a list of all accounts ordered by usage.

Figure 19 - Code for sorted account list


```
// Get the list of accounts
private List<String> getAccounts() {
    List<String> accounts = new ArrayList<>();

    try {
        // Parse the XML file
        File xmlFile = new File(PasswordManager.databaseFile);
        Document database = DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(xmlFile);

        // Form a list of the account elements
        NodeList accountList = database.getElementsByTagName("account");

        // Create a map of account names and usage counts
        HashMap<String, Integer> accountMap = new HashMap<>();

        // Run for every account element
        for (int a = 0; a < accountList.getLength(); a++) {
            Node accountNode = accountList.item(a);
            Element accountElement = (Element) accountNode;

            String accountName = accountElement.getAttribute("name");
            NodeList accountDetails = accountNode.getChildNodes();

            // Run for every child node of the account element
            for (int b = 0; b < accountDetails.getLength(); b++) {
                Element usecountElement = (Element) accountDetails.item(b);

                // Make sure there is a value for the count attribute
                if (usecountElement.hasAttribute("count")) {
                    // Add the account name and usage count to the map
                    int accountUse = Integer.parseInt(usecountElement.getAttribute("count"));
                    accountMap.put(accountName, accountUse);
                }
            }
        }

        // Sort map by usage count
        List<Map.Entry<String, Integer>> accountDisplayList = new ArrayList<>(accountMap.entrySet());
        accountDisplayList.sort(Map.Entry.comparingByValue(Comparator.reverseOrder()));

        // Add the account names to the list to display
        for (Map.Entry<String, Integer> accountEntry : accountDisplayList) {
            accounts.add(accountEntry.getKey());
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    return accounts;
}
```

The XML file containing the account information is **parsed** using the **DOM parser** since it provides a way to navigate the entire document and access any element. A **HashMap** is used as it offers fast lookups based on keys (views) and is an excellent way to store data that needs to be accessed frequently and quickly. It is sorted using the **TimSort algorithm**, one of the most stable, fastest, and efficient sorting algorithms - chosen for those reasons.

Utility Class

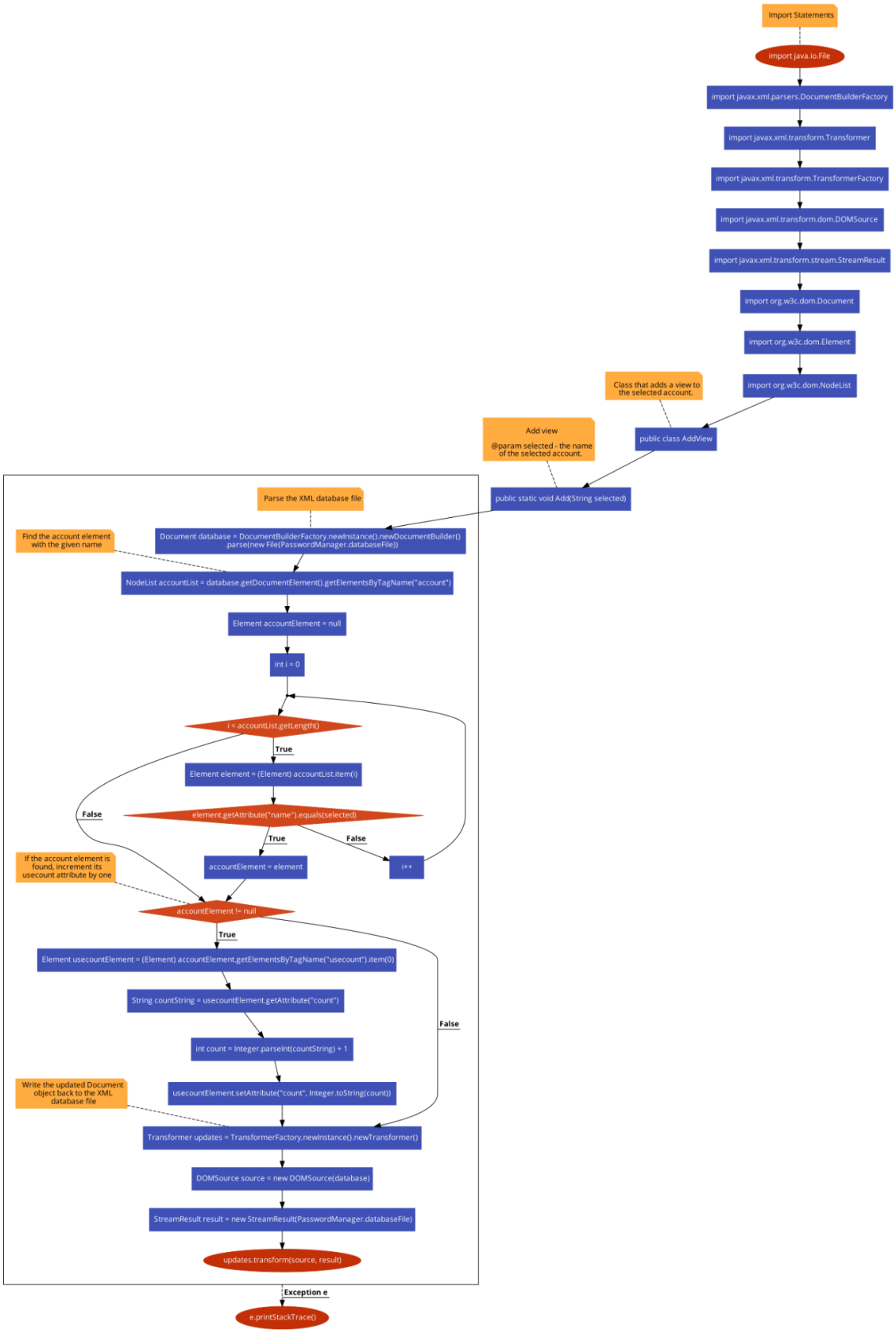
Below is the utility class used to add a view to the selected account when the user clicks the "View Details" button. The Add() method is used **polymorphically** because it takes a String parameter and can be called with different arguments, making the code more versatile.

Figure 20 - AddView.java code

```java
// Import Statements
import java.io.File;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;

/**
 * Class that adds a view to the selected account.
 */

public class AddView {

    /**
     * Add view
     *
     * @param selected - the name of the selected account.
     */
    public static void Add(String selected) {
        try {
            // Parse the XML database file
            Document database = DocumentBuilderFactory.newInstance().newDocumentBuilder()
                    .parse(new File(PasswordManager.databaseFile));

            // Find the account element with the given name
            NodeList accountList = database.getDocumentElement().getElementsByTagName("account");
            Element accountElement = null;
            for (int i = 0; i < accountList.getLength(); i++) {
                Element element = (Element) accountList.item(i);
                if (element.getAttribute("name").equals(selected)) {
                    accountElement = element;
                    break;
                }
            }

            // If the account element is found, increment its usecount attribute by one
            if (accountElement != null) {
                Element usecountElement = (Element) accountElement.getElementsByTagName("usecount").item(0);
                String countString = usecountElement.getAttribute("count");
                int count = Integer.parseInt(countString) + 1;
                usecountElement.setAttribute("count", Integer.toString(count));
            }

            // Write the updated Document object back to the XML database file
            Transformer updates = TransformerFactory.newInstance().newTransformer();
            DOMSource source = new DOMSource(database);
            StreamResult result = new StreamResult(PasswordManager.databaseFile);
            updates.transform(source, result);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

It **parses the XML file**, iterates through the XML file to find the account element with the selected name, and then increments the use count attribute of the account element by one.

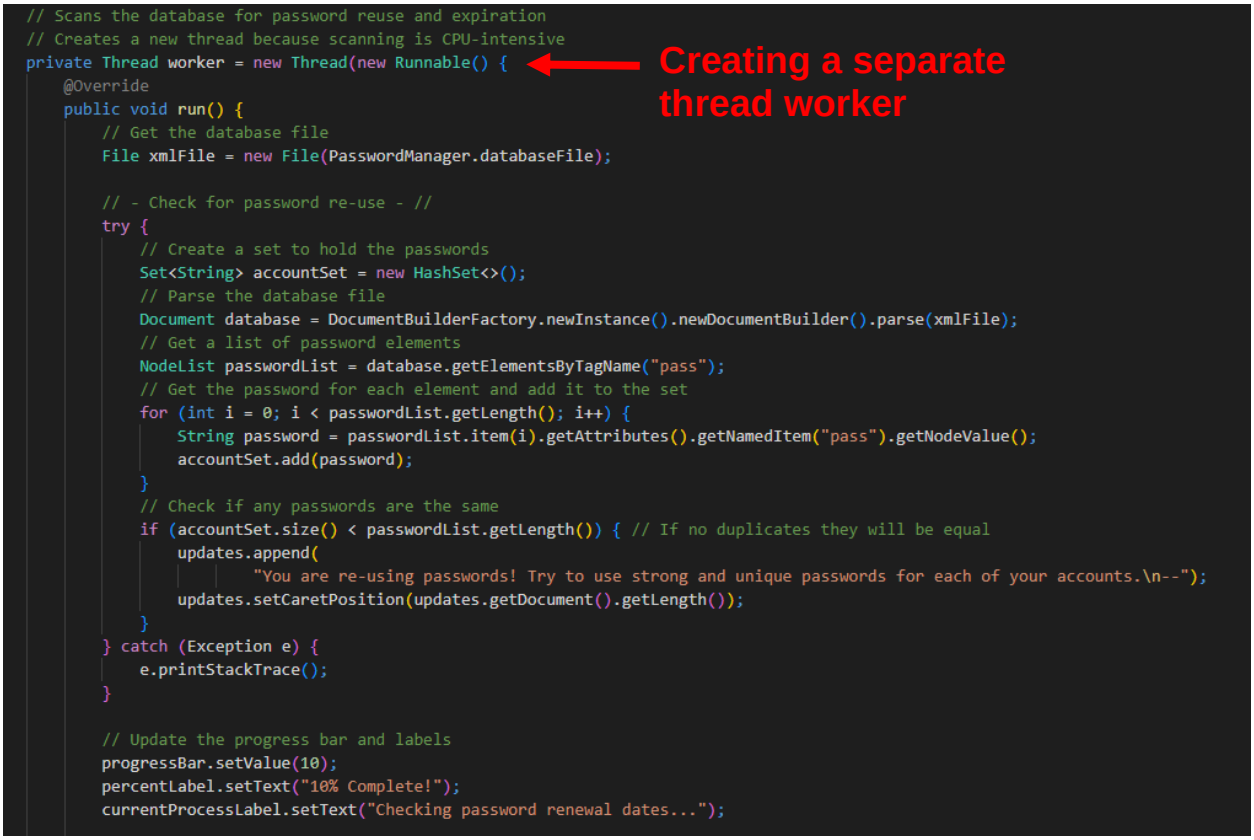
Figure 21 - Flowchart of AddView.java



Scan Password Feature

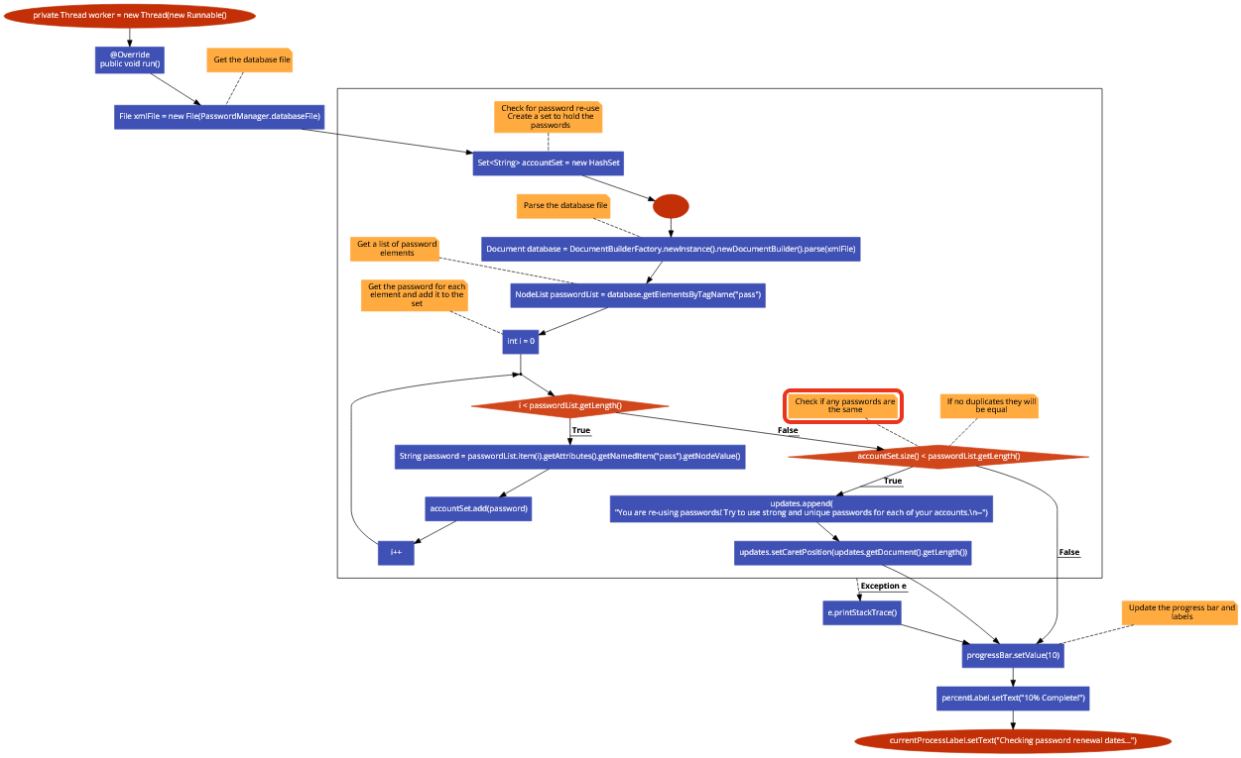
A separate CPU thread allows the application to perform long-running operations in the background without freezing the GUI. If the below code executes on the main thread, the UI will freeze until the code finishes running, preventing the user from interacting with the application.

Figure 22 - Code to check for reused passwords




```
// Scans the database for password reuse and expiration
// Creates a new thread because scanning is CPU-intensive
private Thread worker = new Thread(new Runnable() {
    @Override
    public void run() {
        // Get the database file
        File xmlFile = new File(PasswordManager.databaseFile);

        // - Check for password re-use - //
        try {
            // Create a set to hold the passwords
            Set<String> accountSet = new HashSet<>();
            // Parse the database file
            Document database = DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(xmlFile);
            // Get a list of password elements
            NodeList passwordList = database.getElementsByTagName("pass");
            // Get the password for each element and add it to the set
            for (int i = 0; i < passwordList.getLength(); i++) {
                String password = passwordList.item(i).getAttributes().getNamedItem("pass").getNodeValue();
                accountSet.add(password);
            }
            // Check if any passwords are the same
            if (accountSet.size() < passwordList.getLength()) { // If no duplicates they will be equal
                updates.append(
                        "You are re-using passwords! Try to use strong and unique passwords for each of your accounts.\n--");
                updates.setCaretPosition(updates.getDocument().getLength());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

        // Update the progress bar and labels
        progressBar.setValue(10);
        percentLabel.setText("10% Complete!");
        currentProcessLabel.setText("Checking password renewal dates...");
```

Figure 23 - Flowchart of reused passwords checks

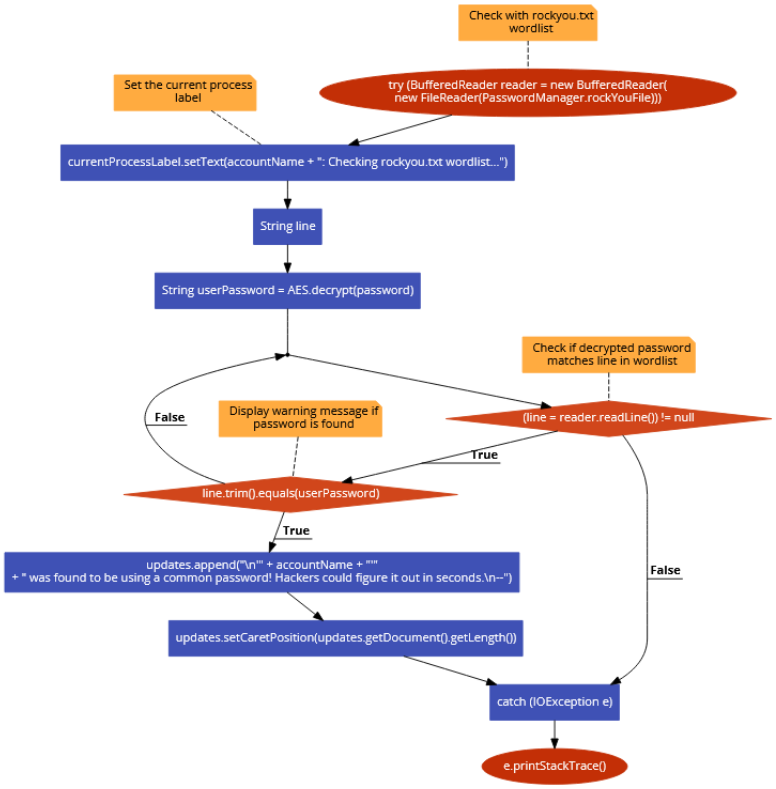


Additionally, to make the program run even smoother, I used a buffered reader when checking each password against the rockyou.txt wordlist, a list of millions of common passwords. A buffered reader breaks down the file into small chunks without loading the entire file into memory at once. That way, as soon as the password gets found, it does not have to load the rest of the file - significantly improving performance and reducing memory usage.

Figure 24 - Code to check with rockyou.txt wordlist

```
// Check with rockyou.txt wordlist
try (BufferedReader reader = new BufferedReader(
        new FileReader(PasswordManager.rockYouFile))) {
    // Set the current process label
    currentProcessLabel.setText(accountName + ": Checking rockyou.txt wordlist...");
    String line;
    String userPassword = AES.decrypt(password);
    // Check if decrypted password matches line in wordlist
    while ((line = reader.readLine()) != null) {
        // Display warning message if password is found
        if (line.trim().equals(userPassword)) {
            updates.append("\n'" + accountName + "'"
                    + " was found to be using a common password! Hackers could figure it out in seconds.\n-
            updates.setCaretPosition(updates.getDocument().getLength());
            break;
        }
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Figure 25 - Flowchart of rockyou.txt check



Additional Information

- Appendix III – Bibliography
- Appendix IV – Extensibility List

Word Count: 910