Finvest Holdings

Security Software & Analysis

SYSC 4810 Assignment

Nathan MacDiarmid

101098993

Passwords – IloveCats2!

December 4th, 2023

Table of Contents

[Problem 1 3](#_Toc151735868)

[Problem 2 8](#_Toc151735869)

[Problem 3 12](#_Toc151735870)

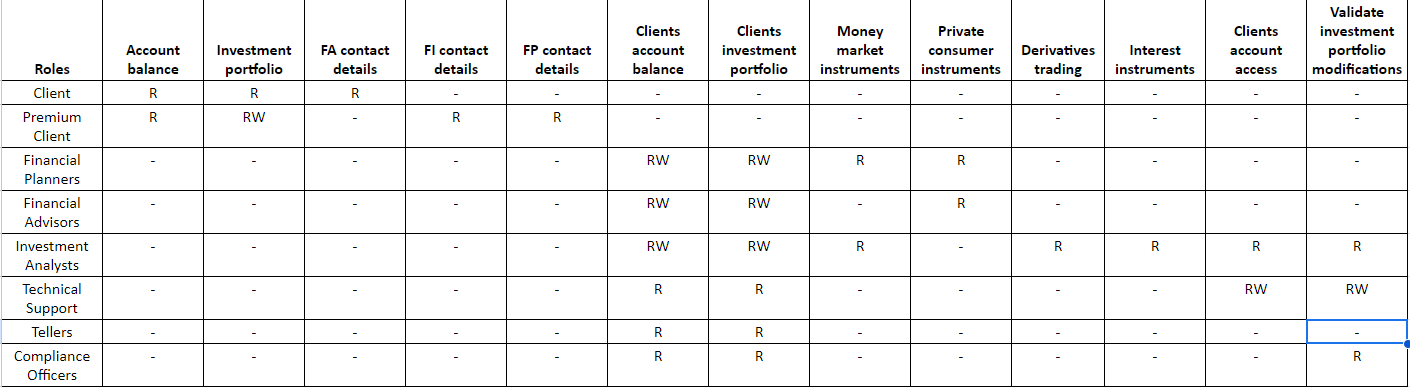
[Problem 4 19](#_Toc151735871)

[Problem 5 23](#_Toc151735872)

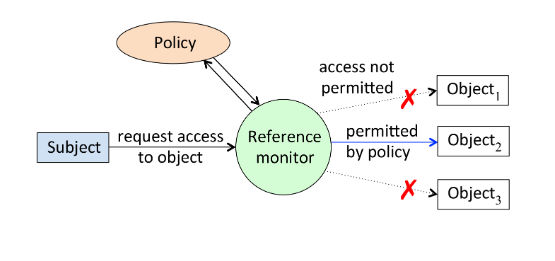
[References 24](#_Toc151735873)

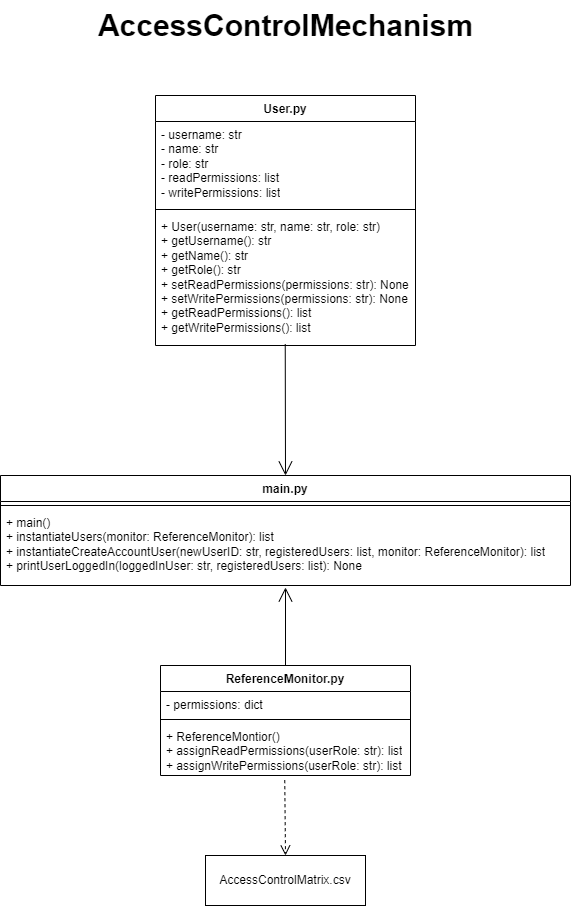
# Problem 1

1. The access control model that the Finvest Holdings security system will be the RBAC control model. This is because RBAC sets permissions based on roles which is similar to what is described in Part 1 Section 2 – Context. RBAC also allows for the expandability of permissions via additions of different roles, making it very scalable.
2. The access control representation that the Finvest Holdings security system will be is an access control matrix. This is because each of the roles have different access to various permissions and an access control matrix allows for the setting of permissions for each role separately. Furthermore, each row can be used as a capabilities list, depicting the permissions of each role, and each column can be used as an access control list, depicting how many roles have access to a selected permission.
3. In my access control matrix, a role can only have read, write, or no permissions at all. Since there didn’t seem to be any programs that required running in the description as a role would either view the information or edit that information, I chose to only allow for the options of reading and writing. Upon checking if the User is a Teller, the User will be notified the times they are allowed to access the program.

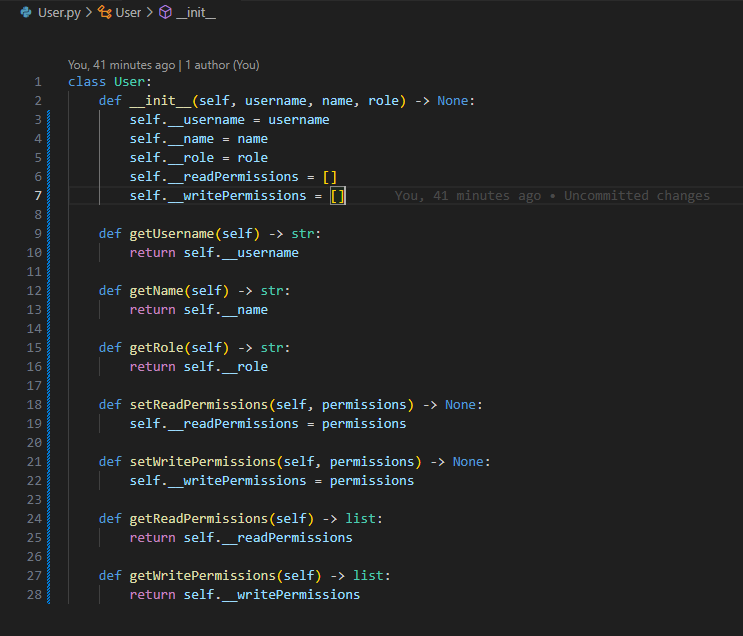


1. The access control mechanism was implemented in an object-oriented way such that it models the example of a reference model implementation that was given in class in slides Ch5\_2[3]. The example given in class and a UML diagram of the access control mechanism is depicted below.





In the example, the Subject requests permission access from the Reference Monitor and the Reference Monitor assigns some or no permissions based on a Policy. In my access control mechanism, the User is a Subject requesting permissions from the ReferenceMonitor that is determined by the AccesesControlMatrix. Each User contains a username, a name, a role, readPermissions and writePermissions which are all private attributes and can only be accessed via getter and setter methods. The username is a unique name only associated to that User and treated like a user ID. The name is the Users personal name, and the Users role is the permissions role such as Client, Premium Client, etc. The readPermissions and writePermissions contained the appropriate permissions that were sent back from the ReferenceMontior.



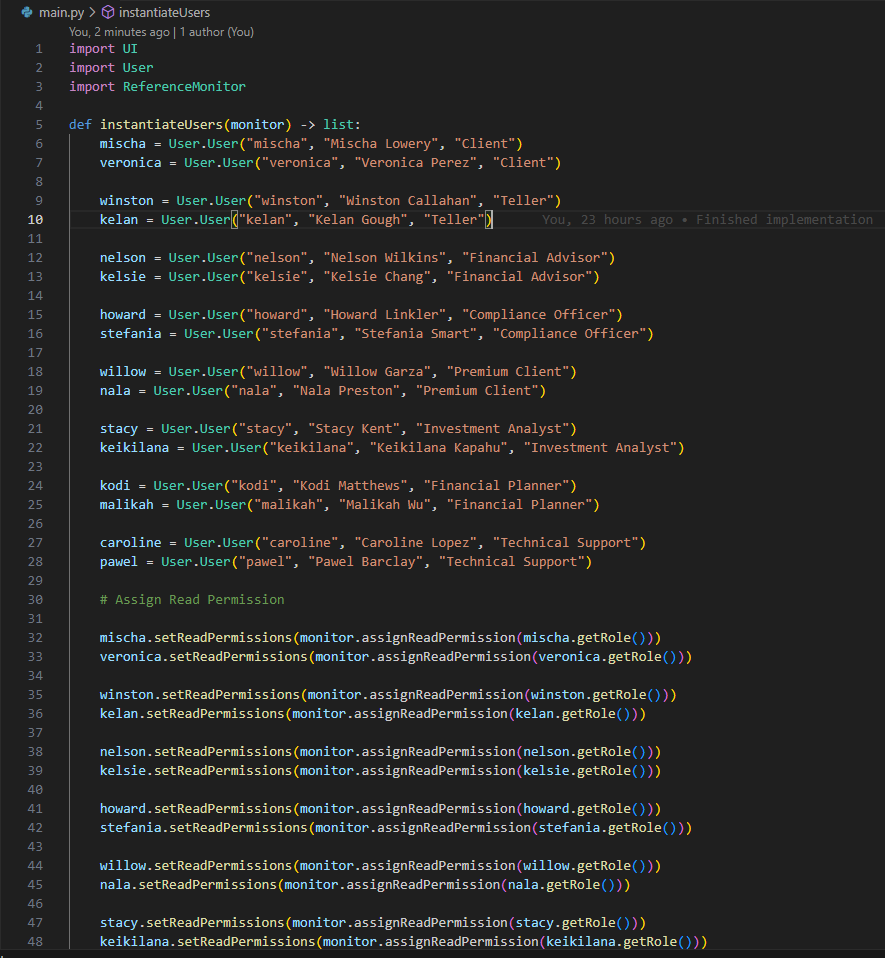
The Reference Monitor in the example controls what Subjects get access to what permissions based on a Policy. In my access control mechanism, the ReferenceMonitor reads from the AccessControlMatrix file and stores all the permissions in a private dictionary attribute called permissions. The ReferenceMonitor is the only file that can access the AccessControlMatrix and has two methods that control what permissions a User receives called assignReadPermissions() and assignWritePermissions(). With the ReferenceMonitor being the only file that can access the AccessControlMatrix, it adds an extra layer of security to the system by extrapolating the matrix to only one entry point; and since no other files have direct access, the risk of another file changing or modifying permissions decreases dramatically.

Like their names suggest, the methods in the ReferenceMonitor assign a User their read and write permissions respectfully. They each do this by receiving a User’s role that’s passed into the method and returning a list of permissions based on that specific role. If the User role that’s passed does not exist, or has no permissions, these methods return an empty list, ensuring that User’s are only passed the proper rules.



The main file is where everything is instantiated, and different User’s get their defined roles based on the ReferenceMonitor’s AccessControlMatrix. One instance of the ReferenceMonitor is instantiated as well as the list of User’s that were provided in the assignment description. These User’s are only used as an example. The main file contains four methods, instantiateUsers(), instantiateCreateAccountUser(), printUserLoggedIn(), and main().

The instantiateUsers() method takes in a reference to a ReferenceMonitor instance, creates instances of all the Users, and sets the Users’ permissions based on the AccessControlMatrix from the ReferenceMonitor instance. The instantiateCreateAccountUser() takes in a newUserID which is a string representing the ID of a new User that isn’t registered in the system yet. It also takes in a list of registeredUsers, and a reference to a ReferenceMonitor instance. The monitor instance is used to set this new User’s permissions. Every new User is automatically assigned the ‘Client’ role as it has the lowest level of permissions given in the assignment description. The printUserLoggedIn() method takes in a loggedInUser which is the username of the User that successfully logged in, and a list of registeredUsers. The printUserLoggedIn() method prints out all the appropriate information that a User should receive upon logging in given the assignment description. This includes the User’s username, role, readPermissions and writePermissions. It also includes special permissions if the User’s role is a ‘Teller’. Lastly, the main() method takes no inputs and calls all three of these previously defined methods. The main() method also instantiates a ReferenceMonitor instance and keeps a list of registeredUsers in the system. Furthermore, it is the only method that is called globally in the entire program and renders the entire system.





The reason that I chose to create a User class and a ReferenceMonitor class separately was because it keeps the AccessControlMatrix away from the User. By not giving the User direct access to the AccessControlMatrix, I added another layer of security to the system by only providing the User with what it needs which is the given permissions.

1. Testing here

# Problem 2

1. Since I used Python, there is a library called hashlib [2] that can be imported from the Python API. The hashlib library contains various hash functions that are one way, contain second-preimage resistance, and collision resistance which are all three properties that a proper hash function should have. The Secure Hash Algorithm (SHA) hash functions vary from 224 bits to 512 bits. Although SHA-1 has known security vulnerabilities, SHA-2 does not. SHA-2 are the updated hash functions that eliminate these vulnerabilities once again create an un-hashable function. I used the SHA512 hash function as it uses the most bits to hash a given string, resulting in the most possible number of combinations of bits to hash that string. This makes it the most difficult hash function to decrypt as it contains the highest number of bits.
2. The password record structure of the information that is stored in the passwd.txt file follows the format “username salt hashed salt and password combination”. The password file does not contain permissions of a User, nor the User’s role itself. This is because passwd.txt is simply used as a verification tool to verify that User’s username and password match the given login information. I did this because it adds another additional layer of security to the system specifically regarding offline attacks. Since the passwd.txt file only contains a username, salt, and salt and password combination, an attacker looking for a specific account with specific permissions will be unable to know which ones are which. This deters the attacker more because if they were somehow able to figure out the hashed password, they wouldn’t know what type of permissions they would be getting, especially if they’re looking for a ‘Premium Client’ for example that is able to edit a User’s portfolio.

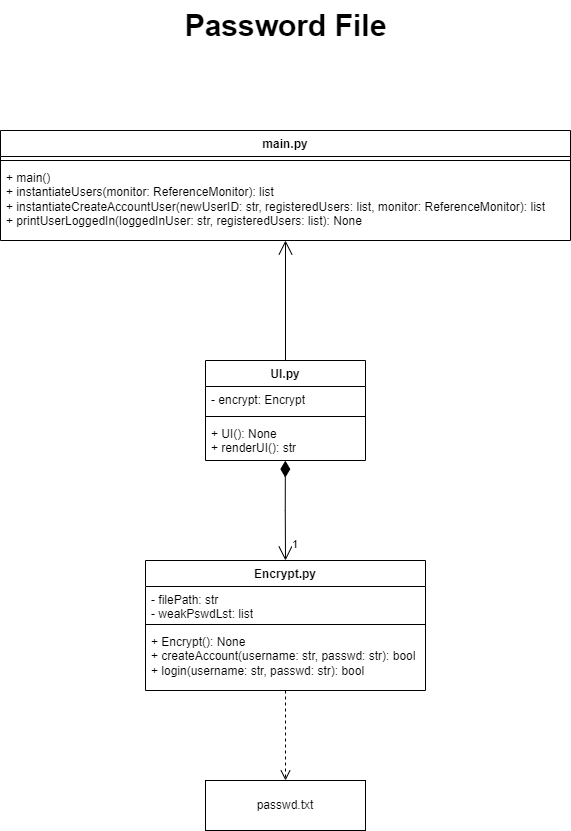
Below is a screenshot of the passwd.txt file with all the Users from the assignment description loaded in. For testing purposes, all of the usernames are the names listed and all of their passwords are ‘IloveCats2!’. As you can see, all of the password hashes are different, even with the password being the same. This proves that the SHA512 hash function that was chosen contains all three properties that are needed in a hash function.



1. The password file is where a generic user interface began. Since Problem 4 contains instructions on designing a user interface to log Users in and this problem needed abilities to get and store passwords with other information, I decided to implement a basic UI so that I could pass my own information into passwd.txt. This allowed for manual testing during the development of the password file, ensuring that what I stored in the file could be properly retrieved later. The basic UI that I used and the UML diagram of password file portion of this assignment is depicted below.

A black screen with white text

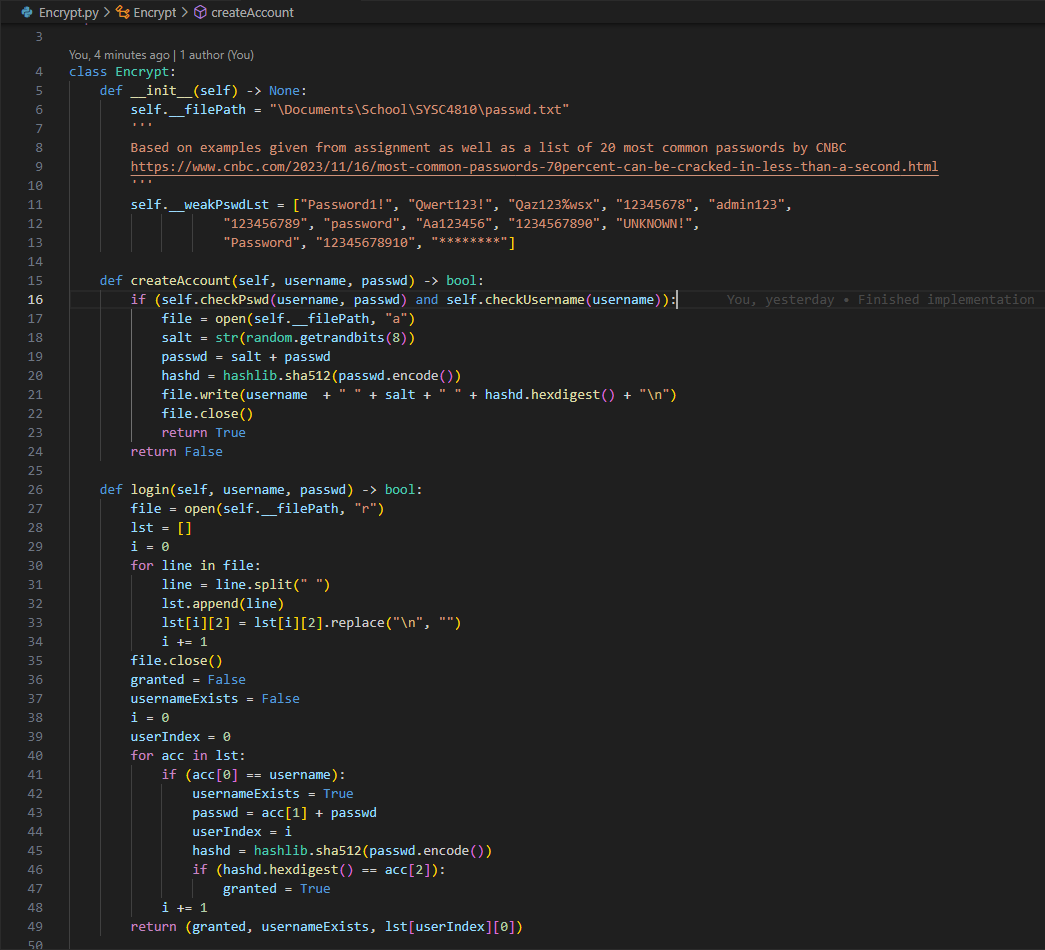
Description automatically generated



The passwd.txt file is only accessible by an Encrypt class. This is similar to the ReferenceMonitor and AccessControlMatrix scenario from Problem 1, by extrapolating the passwd.txt to only one access point, the Encrypt class, it dramatically reduces the possibility of other files changing or incorrectly accessing parts of the passwd.txt file. Additionally, the Encrypt class contains two private attributes called filePath and weakPswdLst. The filePath is the string of the path to the passwd.txt file, there are no getters or setters for this attribute so that it is more difficult for an attacker to change the file path to a possible bad passwd.txt file. The weakPswdLst is a list of commonly used passwords that should be avoided, which will be touched on more in Problem 3 c.

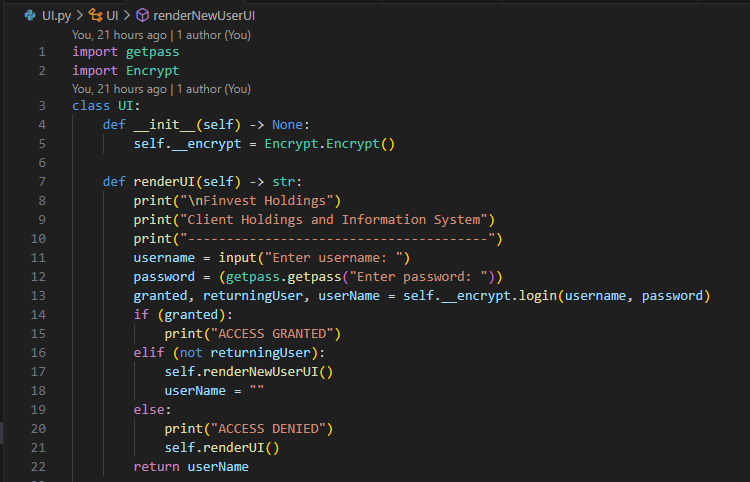
The Encrypt class also contains a createAccount() method as well as a login() method that allows for new information to be passed to the passwd.txt file as well as previously stored information to be retrieved. The createAccount() stores information in the passwd.txt file by taking in a username and password, generating a random salt value from 0 -> , concatenating the salt with the given password, hashing the concatenated password with SHA512 and storing the hashed password with the given username and the generated salt value in the ‘user salt concatenated password’ pattern.

The login() method allows for the retrieval of information from the passwd.txt file. It does this by taking in a username and password, similar to createAccount(), but instead it reads the file line by line, looking for a match on the given username. If it finds a username that matches the given username, login() concatenates the stored salt attached to that username to the given password, hashes that with SHA512 and compares it to the hashed password attached to the username that is stored in passwd.txt. If the password hashes match, the username is then retrieved from passwd.txt and returned. There will be more explanation on this in Problem 4 c.



Any of the inputs that are going into the createAccount() or login() methods come from the UI class. This was my manual testing entry point for this problem as it allowed me to pass and retrieve Users from the passwd.txt file. The UI class contains one attribute called encrypt which is an instance of the Encrypt class. It is a private attribute with no getters and setters. It also contains one method called renderUI() that takes no inputs.

The renderUI() method uses the built in input() functionality from Python and requests input from the user via terminal. A username and password variable are used locally to store this input and then passed to the login() method from the Encrypt class. Depending on the return value of login(), the user can then call the createAccount() method from the Encrypt class, or retry the login() method. If the username and password combination are successful, the username is then passed to main to handle assigning the proper permissions to that User. More information on this will be discussed in Problem 4 c.



1. During the development process, I tested the password file by initially passing in different username and password pairs and saving them to the passwd.txt file. I then implemented the ability to hash the password, including the hexdigest() function from sha512 and hashlib so that it wasn’t saved as a hashed object but instead a string of letters and numbers. After, I implemented a random bit generator to generate a random salt value to be attached to the password before its hashed. I then tested that this salt and password combo would save correctly. I then made sure that I could retrieve the salt password combo from passwd.txt. Lastly, I made sure that when I appended the plaintext salt from passwd.txt to the same plaintext password that I stored, they matched. Since I developed the test cases after the entire implementation was finished, the test cases from Problem 3 d and Problem 4 d will subsume these test cases.

# Problem 3

1. The simple user interface is depicted identically to what was asked in the assignment description. The first and only text that the user sees upon trying to login to the system is ‘Finvest Holdings Client Holdings and Information System’. It also prompts the user to enter a username followed by a password. If the username and password pair match a user, ‘ACCESS GRANTED’ is printed, otherwise, if the username exists and the password doesn’t match, ‘ACCESS DENIED’ is printed. This will differ slightly in Problem 4 c when implementing the actual login capabilities. The password is not shown to protect one’s input which is done using the getpass Python library. The Python library getpass contains no known security flaws and does not store input anywhere. All of this is rendered and printed out by the UI class.

A black screen with white text

Description automatically generated

If the given username doesn’t exist, the enroll user interface is displayed. It contains the ‘Finvest Holdings’ and ‘Client Holdings and Information System’ text similar to the login page, with the addition of the ‘CREATE ACCOUNT’ text. This again prompts the User for a username and password pair.

A screenshot of a computer

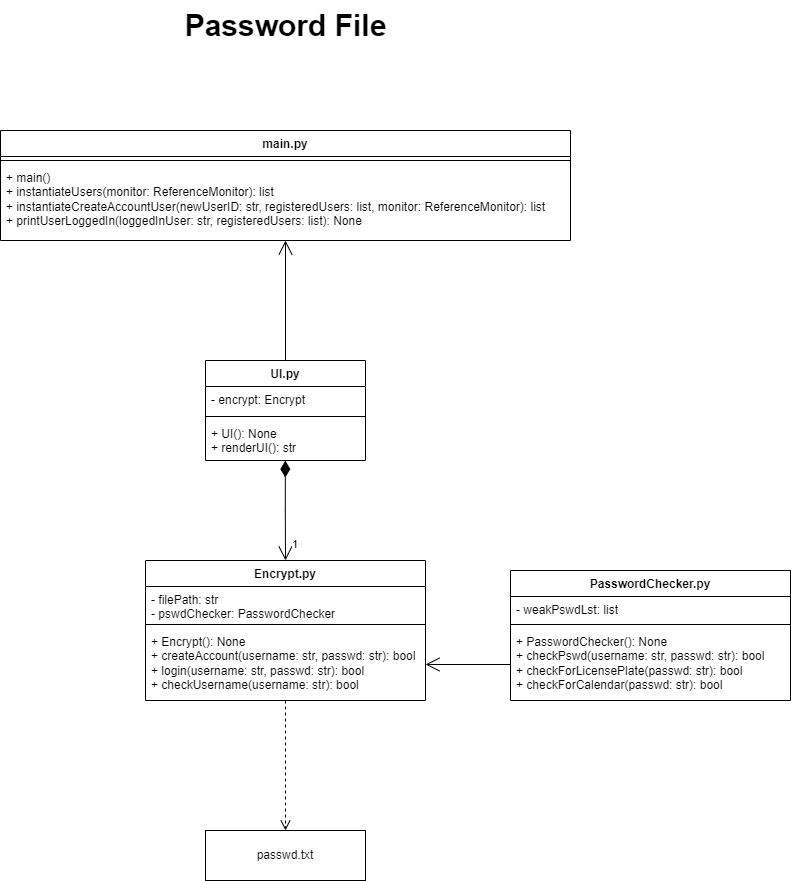
Description automatically generated

If the username already exists or the password does not meet the requirements, it will print out ‘INVALID INPUT’, as well as ‘Username must be unique’ and ‘Password must include:’ followed by all of the password requirements as depicted in the assignment description and seen in Problem 3 b. Once the user account has been created, the program ends and must be restarted for the User to be able to login with the username and password pair they provided. This is illustrated in Problem 4 c.

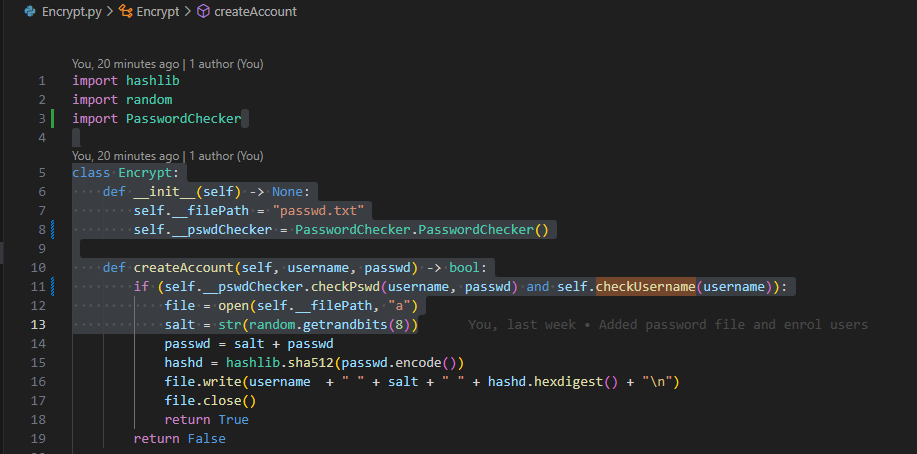
A screen shot of a computer

Description automatically generated

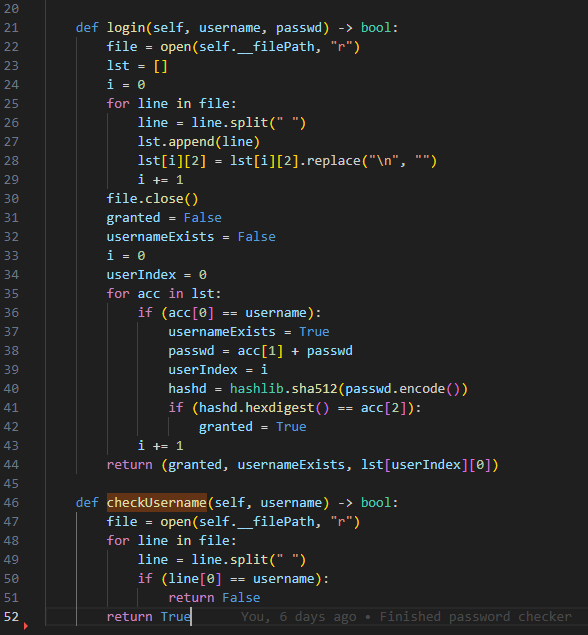
1. The proactive password checker has a few different parts to it, making sure that each of the required portions for creating the password are met. The password checker requires passwords to be 8-12 characters in length and must include at least one upper-case letter, one lower-case letter, one numerical digit, and one special character from the set {!, @, #, $, %, \*}. The password checker also contains a blacklist of commonly used weak passwords that was found on an article written by CNBC, a news platform. The password checker also protects against calendar dates, phone numbers, and license plate formats. A password contains a calendar date if it contains six numerical digits one after another. This also inadvertently protects against phone numbers since a phone number is nine numerical digits and a calendar format will already be detected and not allowed upon the sixth digit. The password checkers protects against Ontario specific license plate numbers in the format WXYZ123 where WXYZ are any letter, upper-case or lower-case and 123 are any numerical digit. If any of these formats are detected anywhere within the provided potential password, including beginning, middle, or end, the password is not allowed. Lastly, the password cannot be the same as the username.
2. The proactive password checker is implemented as its own PasswordChecker class. I decided to do this as it separates the concerns keeping the single responsibility principle in mind. The PasswordChecker contains all of the methods necessary to check the given password format so that all of the requirements in Problem 3 b are met. Below is the updated UML diagram on the passwd.txt side of main.py that depicts all of the methods inside PasswordChecker and how it relates to the rest of the program.



The Encrypt class now has a PasswordChecker instance variable. This is used to call the methods from PasswordChecker directly in the Encrypt class. As described in Problem 2 c, the username and password are already being passed from the UI to the Encrypt class. With the PasswordChecker addition, the Encrypt class simply passes the username and password one step further to the checkPswd() method in PasswordChecker to run the password checks. The username and password are not saved in the PasswordChecker and are only passed as attributes, avoiding any possible security risks. A Boolean is returned by checkPswd() to the Encrypt class, returning true if the password can be used and false otherwise. This can be seen on line 11 in the code snippet below.

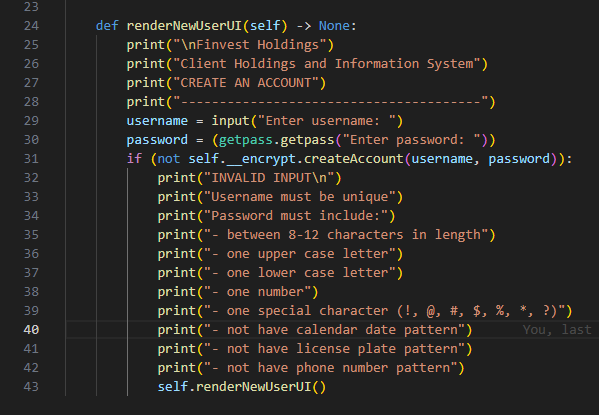


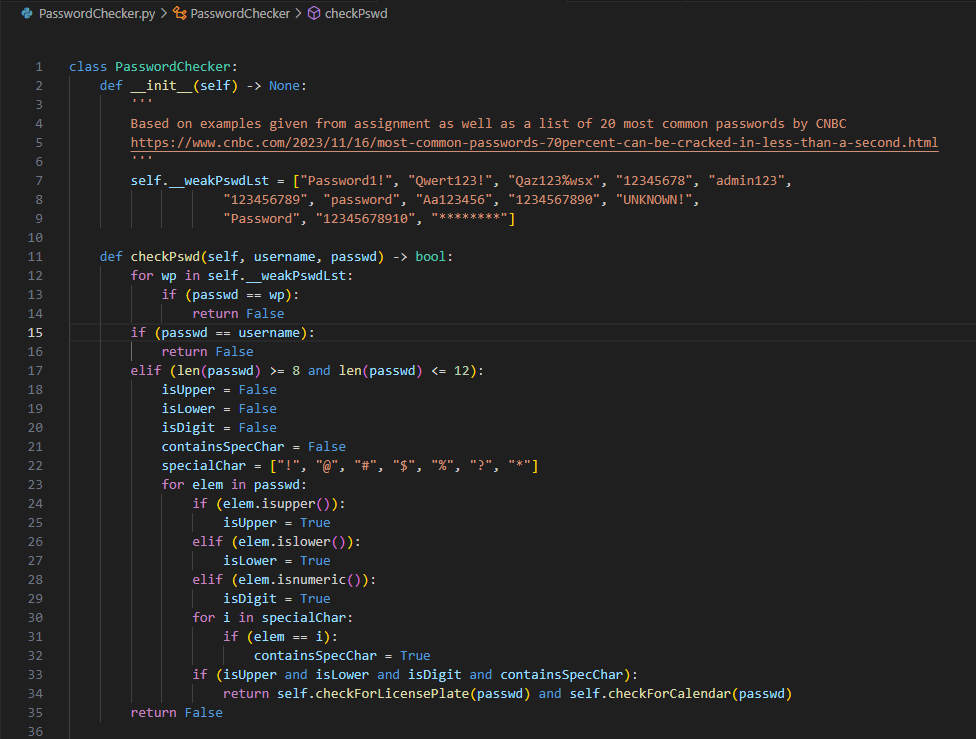
This next photo inside the Encrypt class depicts the functionality of checkUsername() method that verifies that the given username is unique. This method is in the Encrypt class and not the PasswordChecker as Encrypt should be the only one to be able to access the passwd.txt file. This limits the number of potential security risks.



The PasswordChecker class contains one private attribute called weakPswdLst which is the list of weak passwords from CNBC described in Problem 3 b. PasswordChecker has three methods called checkPswd(), checkForLicencePlate(), and checkForCalendar(). All these methods are part of the password checker implementation. The createAccount() method now passes the username and password to checkPswd() to perform the password checks described in Problem 3 b. The reason both the username and password are passed to the checkPswd() method is to verify that the username and password are not the same.

The checkPswd() method is the main method in PasswordChecker that returns the final Boolean to the createAccount() method in Encrypt, being true if the password is usable and false otherwise. It begins by parsing the weakPswdLst and checking if the given password matches any of those, if it does, checkPswd() returns false and the remainder of the method does not run. The checkPswd() method then checks if the password and username are the same, if they are, checkPswd() returns false. If the username and password are not the same, checkPswd() checks if the password length is between 8 and 12 inclusively. If that’s true, checkPswd() loops through each individual character of the password and checks if there’s an upper-case and lower-case letter, a number, and a special character from the set {!, @, #, $, %, \*}. The checkPswd() method stores if the character has each of these in Boolean expressions for each requirement, ie isUpper is changed to true if an upper-case character is found. If all three of these Booleans are true, checkPswd() returns the Boolean result of a call of checkLicensePlate() and checkForCalender(). Otherwise, checkPswd() always returns false, indicating that the password is not usable.

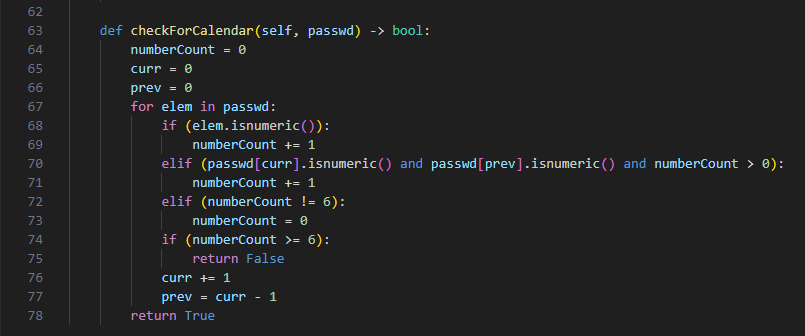




The checkForLicensePlate() method checks for an Ontario license plate format, WXYZ123, anywhere within the password, this could be the beginning the middle or the end. The method returns true if the password does not contain a direct license plate format and false if it does contain the license plate format, indicating that the password is not usable. It first initializes four variables to 0 called letterCount, numberCount, curr, and prev. The letterCount is to keep track of how many letters in a row there are in the password. The numberCount is for how many numbers in a row there are, but this only starts counting after letterCount has reached four, ie, there are four letters in a row. The curr and prev are simply to keep track of which character index the loop is on while looping through the password with curr being the current character index and prev being the previous character index. Every loop through, checkForLicensePlate() checks if the letterCount is four. If it is, it then checks if the current element is a number, if so, it increments numberCount. Otherwise, numberCount is reset to zero. If letterCount is not four, checkForLicensePlate() then checks if the current element is a letter, if so, it increments letterCount. Lastly, the loop checks if letterCount and numberCount equal four and three respectfully, if they do, we have a license plate pattern and false is returned, indicating that the password can’t be used. Otherwise, curr and prev are incremented and the loop continues. If checkForLicensePlate() doesn’t return prematurely, it returns true, indicating that the password is valid and useable.



The checkForCalendar() method is much less complex than the checkForLicensePlate() method. Following the format for a date being 23/12/06, it is a sequence of six consecutive numbers. So checkForCalendar() simply iterates through the password and checks if the character in the password is a number. If so, it increments a numberCount similar to checkForLicensePlate(), otherwise it resets to zero. If the numberCount ever reaches six consecutive numbers, checkForCalendar() returns false, indicating that the password is unusable. Otherwise, the curr and prev variables are incremented like checkForLicensePlate() and the loop rolls over. If checkForCalendar() doesn’t find six consecutive numbers, it returns true, indicating that the password is usable. Like I mentioned previously, this method also implicitly detects phone numbers and the other popular date format which is 123-456-7890 and 2023/12/06 respectfully. This is because both have more consecutive numbers than six, so the checkForCalendar() method already handles it. I did this to reduce redundant code.



If all of these conditions are met, the createAccount() method in the Encrypt class is allowed to save the username and encrypted password in the passwd.txt file.

1. Testing here

# Problem 4

1. The user interface that was described in Problem 2 c and Problem 3 a remains the same. The only thing that changes is after a User has been verified, it prints out the read and write permissions that a user has. This is delved into deeper in Problem 4 c. Below is what is printed out upon User validation.

A screenshot of a computer

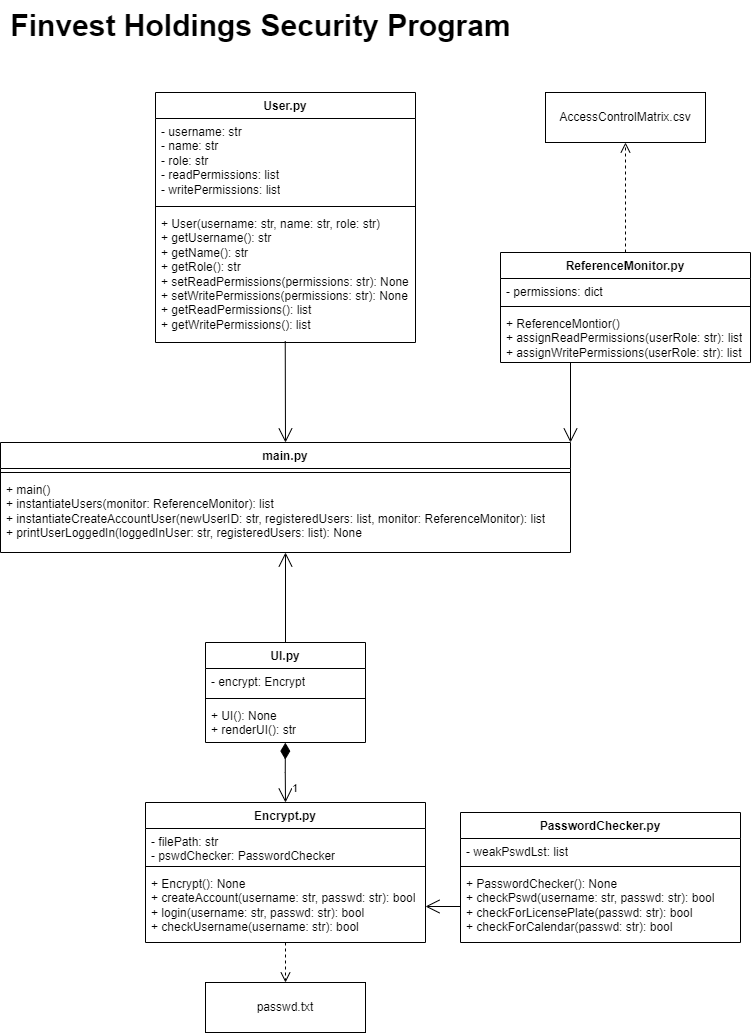
Description automatically generated

1. The password verification mechanism was developed as part of the Encrypt class. Since the Encrypt class is the only class that has access to the passwd.txt file, I left it that way, keeping only one entry point, diminishing any security vulnerabilities coming from multiple access points on the passwd.txt file. It is implemented in one method in Encrypt that is called by the UI class, passing the username and password pair. The method is called login(), which is very fitting for what is happening. Firstly, it opens the passwd.txt file and reads it line by line. On each line, it splits what is related to that account into three different elements that are stored in a local variable. That variable is then only referenced once in that iteration of the loop, checking that the username exists in the file. If the username exists, the plain text salt from that account is appended to the password that was passed from the User through the UI to the login method. That new password is then hashed and compared to the hash value that is stored in the passwd.txt file. If they match, the User is logged in, otherwise the ‘ACCESS DENIED’ message mentioned in Problem 3 a is displayed. The login method returns a three-digit tuple including a Boolean, true if the User was authenticated and false otherwise. It contains another Boolean indicating if the username exists. This is what’s used for displaying the ‘ACCESS DENIED’ message instead of the ‘CREATE ACCOUNT’ message. Lastly, it contains the username that was passed in by the UI. If the username password doesn’t match, it returns an empty string, indicating the User must create an account. The reason it returns the username that was passed and not the username that is contained in the file is to protect against injection attacks that attempt to get a valid username from the passwd.txt file.

A computer screen with text

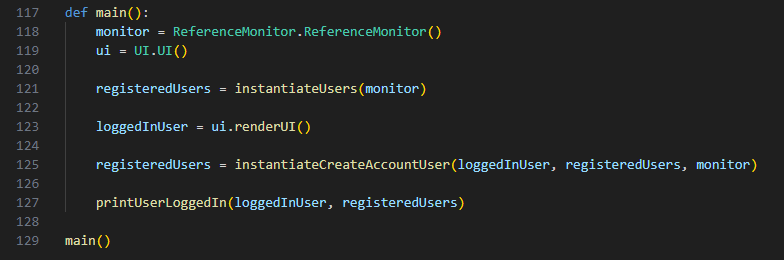
Description automatically generated

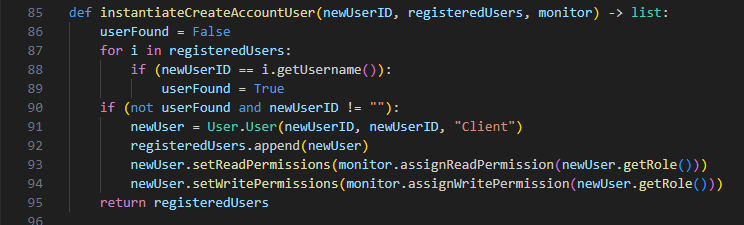
1. To enforce the access control method, we need to show a depiction of the entire program. Throughout this report I have been showing different parts as they have been mentioned, but in this part, I will bring together all the parts to display the entire UML diagram of the system.



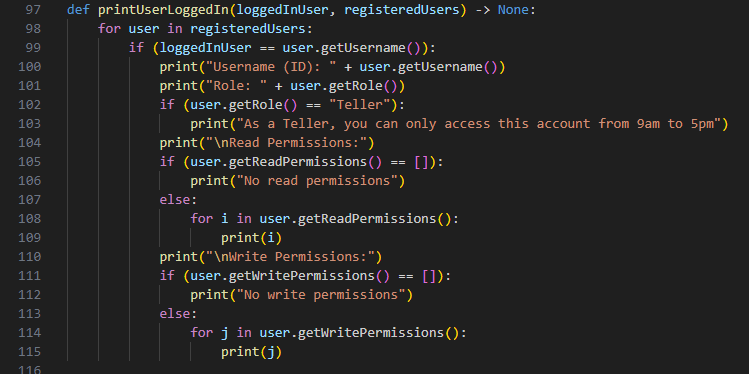
To enforce the access control mechanism that was described in Problem 1, the bottom half of the UML diagram and the top half of the UML diagram must work together. The bottom half enforces all password related issues such as password verification and encryption. It also handles what the User sees via the UI. The top half of the UML diagram handles anything regarding the access control mechanism such as User accounts and permissions. Both halves are tightly coupled to one another, keeping similar responsibilities relatively accessible to either side relative to the other side. Furthermore, it keeps the responsibilities contained in the AccessControlMatrix and the passwords contained in the passwd.txt file far away from each other so that an attacker must go farther into the program to get both files. Additionally, if an attacker gets both files, it will be much harder to identify how they connect to each other via a single User.

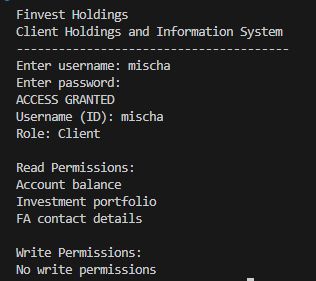
As previously discussed in Problem 4 b, the login() method returns a three-digit tuple containing if the username and password match, if the username alone exists, and the username that was given by the User to the UI. The UI then returns the username or empty string that was passed back from the login() method in the Encrypt class to a variable stored in main.py called loggedInUser. This is when both sides of the UML connect. The loggedInUser variable is then passed to a method called instantiateCreateAccountUser() along with a reference to the ReferenceMonitor that was described in Problem 1 d. This method handles assigning a new User its permissions since permissions are stored within the User instance itself. If a User does not exist in the list of preloaded Users called registeredUsers, a new User is instantiated with the username returned by the UI and the ‘Client’ role permissions. This is because the ‘Client’ role has the lowest number of permissions in the AccessControlMatrix. If a User already exists, this method is simply skipped. This enforces the use of the access control mechanism onto the User, setting its read and writer permissions based on what is stored in the AccessControlMatrix.





Since every User has been instantiated either manually or by creating an account, I needed to display User information to that User once they have been logged in and authenticated. Up until this point, the User could not login and see what their specific read and write permissions are. The printUserLoggedIn() method in main.py prints out User information such as username, role, read, and write permissions. This is where the UI changes slightly for this implementation, resulting in the final version of the UI. The only information that is displayed differently based on role is that of a ‘Teller’. If the role of a User is ‘Teller’, the system notifies them that they can only have access to the system from 9am to 5pm.





1. Testing here

# Problem 5

# References

1. <https://www.linkedin.com/pulse/comprehensive-overview-access-control-models-rbac-abac-jay-/>
2. <https://docs.python.org/3/library/hashlib.html>
3. <https://docs.python.org/3/library/getpass.html>
4. https://www.cnbc.com/2023/11/16/most-common-passwords-70percent-can-be-cracked-in-less-than-a-second.html