Project Title: Password Protector
Team Members: Lei Liu, Christopher Gang, Christopher Sickler, Andrew Guilbeau
UT EIDs: ll28379, cg37877, cbs2468, abg926
Repository URL: https://github.com/chrisliu1234/461L-Project

Homework 2: Information Hiding and Design Patterns
EE461L Group 11

Part A: Information Hiding

In our project, we will implement information hiding in several sections of our project. We use this method in order to separate the client from the implementation of our project.

First, we will create a client that will handle interactions with the database and application, which makes a simpler interaction for the main program to update information in the project. These methods will allow the user to edit the data without knowing how it is accessed or changed. By utilizing this database client, we can decrease interdependence of modules. Information hiding allows us to change our design decisions inside the client without having to change much of the code outside the client.

The advantage of using information hiding is that it utilizes low coupling and high cohesion, and it can make sensitive data hidden to users.

One disadvantage from the user's perspective is that the user is restricted to the methods that are provided to them when creating and deleting passwords. The user has no access to change the database itself. The user also does not have the privilege of changing how the client interacts with the database. Furthermore, the user is restricted from accessing certain information. If the users demand more functionality, future revisions will be made to the application.

```
public class Account{
        private static Account instance;
        private ArrayList<password> accountList;
        // password is an object we will create that will have all the different information (username,
        // passwords, etc.)
        // password will be set on creation and the user will have the option to later edit the info
        Object obj = new Object();
        private Account(void){
                //instance variables
                passwordList= new ArrayList<>();
        }
        public static Account getInstance(){
                //code written later in Part B in singleton pattern, purpose is to only keep one instance of account
        }
        /** returns an arraylist of the passwords or names to read */
        public ArrayList returnPasswords(void){
                //returns the arraylist of passworrds
        }
        /**
        * returns the arraylist to caller
        * @param none
        * @return arraylist from Passwords class
        */
```

```
	/** creates or deletes a password from the arraylist depending on passed values */
	public void edit(String name, String edit_Type){
		//stores or deletes user’s password into the private password arraylist
		//depending on the value of edit_Type, edit_Type will be something like “store” or “delete”
	}
	/**
	* stores or removes passwords from the arraylist depending on value edit_Type
	* @param name is the passwords name to edit, edit_Type is a value that adds or removes the password
	* @return void
	*/

	//other functions
}
```

Part B: Design Patterns

Singleton:

This project will implement the singleton design pattern, which utilizes memory more efficiently by keeping only one instance of the ArrayList of passwords. This will allow the user and client to edit the single ArrayList created by the program so that all data is edited on a single instance instead of multiple instances.

One advantage of implementing this method is the simplification it provides. Only one ArrayList is produced and accessed as opposed to multiple ArrayLists. Simplicity allows for less margin of error. Another advantage of the singleton design pattern is the ease of use it provides for the user. The user only has access to the getInstance() method, which simplifies the use of the application.

One disadvantage from this method is that the class that uses this method cannot be extended. Another disadvantage is that two users cannot utilize the same class simultaneously.

```
public class Account{
	private static Account instance;
	private ArrayList<password> accountList;
	// password is an object we will create that will have all the different information (username,
	// passwords, etc.)
	// password will be set on creation and the user will have the option to later edit the info
	Object obj = new Object();
	private Account(){
		//instance variables
		accountList = new ArrayList<>();
	}
	public static Account getInstance(){
		if(instance == null){
			synchronized(obj){
				if(instance == null){
					instance = new Account();
				}
			}
		}
		return instance;
	}
}
```

<u>Factory:</u>
This PasswordGenerator abstract class will be implemented for all the different types of passwords (uppercase passwords, lowercase passwords, passwords with special characters, and numbers). We will have classes that extend the PasswordGenerator class and override the createPassword method that will then follow an algorithm to generate the password with each subclass' parameters for the password. This will allow for multiple different implementations of creating the passwords.

The advantage of doing the factory method for the password generation is that we will not always know what the different parameters for the password will be. Giving users the option to input parameters selects the correct factory extended class and simplifies the algorithm for password generation. It also decouples the algorithm for generating the password from the actual factory of the password. Also for debugging of our code, it allows us to know which methods for password creation need to be fixed.

```
public abstract class PasswordGenerator{
        public Password addPassword(String label){
                Password password;
                int params = selectChars(); // method that will allow the user to select the types of
                                             //characters their password will contain
                int size = chooseSize();// method that specifies length of password
                password = createPassword(label, params, size);
                Account.addPassword(); //temporary method meant to highlight how
                                          //we will uniformly store our passwords
                return password;
        }
        abstract Password createPassword(String label, Int params, Int size); //this is the factory
method
}
```

Then we can implement different password generators based on whether the user would like to include security questions and their answers with the associated account.

```
public class passwordOnly extends PasswordGenerator{
        Password createPassword(String label, Int params, Int size){
                if(params == 0){ //only lower-case characters are to be used
                        … //will select only the lower-case range of characters from our list
                }
                elseif(params == 1){
                        … //upper and lower-case characters can be used
                }
                ……              //remaining combination cases, ex: numbers, special characters, etc
                …...
        }
}
```

But thanks to the factory method, we can now make a class that still creates the password, but also now stores security questions and their answers with your password for the account.

```
public class fullPassword(String label, Int params, Int size){
        …. // same implementation as above in terms of password customization
        ….
        securityQuestion();
        securityAnswer();
}
```

The disadvantage to using this design pattern is that it makes the code more complex, which means it will be more difficult to revise if we devise a better method of password generation later on.