Final Design

3.3
The final system decided upon uses one database, but multiple different tables. There is one table that contains every user's username, hashed master password, and their unique salt. Every time a user creates an account, there is a new table made that includes their username (which is unique and cannot be used twice) in the name for identification. In the user's own private table, all of their login information is stored. Their "Labels" (What the account is for, eg: twitter account, xbox account, etc), their usernames, and their encrypted passwords.

This design also has a function to allow the user to generate a secure password to be used for their various accounts. For example, if I was creating a new facebook account and logging my information at the same time into this program, I could click a button to generate a secure password, then copy it to use as my new facebook password and securely store it in the database at the same time. This decreases the likelihood of a user getting their password brute-forced on whatever website they are signing up for.

The encryption system uses PBKDF2 to hash the user's master password. The master password itself is NEVER stored. This ensures that it cannot be retrieved by anybody, even in the event of a database breach.

The normal passwords the user stores (not to login to this program, but for things like twitter, facebook, etc) are encrypted using AES. This is an industry-standard encryption/decryption model, which makes it virtually impossible to decrypt the user's passwords without the Master Password, which again, is never stored.

Essentially, the user creates an account with a master password, which only they know. This master password is then used to encrypt/decrypt their normal passwords. Without the master password, nobody, not even the database administrator, could get their passwords in plain text. Even in the event of a database breach, all they would have is a useless mess of encrypted gibberish.

This model was chosen instead of the other two because it has a high scalability, as well as being easy to maintain. The security is also superior to what was previously proposed, preventing even the system administrator from having access to the passwords stored by the user. It was also much easier to test. Everything being unified into one database made it so that the process was much more streamlined and simple. The use of java's SQLite driver, allowing for creation and modification of SQL databases entirely within java without having to access an external MySQL database makes it incredibly easy to test with all the other components, and leaves issues about the java program interfacing with MySQL out of the picture.

Essentially, it allows all testing to be done in the same place at the same time since the database is embedded into the application itself. This means MySQL not initializing properly or any other number of potential issues will not be a problem. There will also be significantly less lag when running tests (and running the program in general), since the java program handling the logic will not need to interface with an external database. It also allows the program to be more portable, since the SQLite database can be easily taken with everything else and put onto a new machine as easily as any other java component, unlike the MySQL solution, which would require installing MySQL on the new machine (and doing who knows how many things to configure it properly).

This approach streamlines everything into a neat package that is much easier to test, can be transferred much more simply, and will have significantly less errors and lag than the alternative.

3.3.1

The “components” used in this solution can be broadly broken up into the java classes the functionality was separated into.

EncryptionUtil.java - This class is the core of the project. It ensures that all data is stored in a secure manner, and that all hashing, encryption, and decryption is handled properly. Without this class, all this program would be is a database storing plaintext passwords.

The security measures were touched on earlier, and I do not want to re-explain too much as this document is quite long. The PBKDF2 and AES algorithms are secure standards to ensure password encryption and hashing. I did not make them, and they are well-documented. To be brief, the algorithms allow for a master password (which is never stored unhashed and therefore cannot be stolen) to be the only way for the user to get their passwords decrypted. Functions like generateIv and generateSalt use SecureRandom to make sure that the Iv and Salt values being used for the hashing and encryption processes are secure.

The main tests done on this component were decision-table testing, and unit tests were the main ones performed. Things like ensuring the same salt and same password would generate the same hash every time, testing the encryption and decryption functions, testing the uniqueness of salts and Ivs, ensuring null values are not allowed to be used for encryption, etc. It was mostly done with a black box mindset. Do the correct inputs get the correct outputs? There isn’t much else you can do with the encryption testing. Does it work and is it consistent are basically all that matter.

DatabaseManager.java - The next main component was the database manager. This component creates and updates the database. It ensures that information is properly stored and retrieved when requested. Its tests largely were unit tests as well, but with a bit more integration. Things like ensuring null information couldn’t be stored were important, as well as being able to store and retrieve a piece of data and have it be the same coming back out as going in. However, since it does interact with the database a lot (since it handles SQL queries), it can also be seen as a sort of integration testing. Inserting users, credentials, retrieving them, ensuring they're valid, were all necessary to make sure the database aspect worked properly.

Equivalence class testing was the main approach. Trying to test valid vs invalid inputs, and making sure only ones that were acceptable went through. If a test case contains invalid inputs, it should fail, and if it contains valid ones, it should succeed.

Credentials.java - The class that contains the model of how most information is stored. Each entry a user makes for their label, username, and password is stored in a "Credential" object that is then inserted into the database. The Credential contains a label, username, encrypted password, iv, and id values. After a user enters information to be stored, the password is encrypted, and the iv associated with it is also inserted into the Credential for later decryption.

While testing directly on this component is mostly just ensuring that null data is not passed to it, it, like all the components, was used in later integration testing. However, since it is mostly just an object constructor and getters, testing on this was mainly ensuring that data was passed in and retrieved correctly.

MasterKeyCache.java - An object that contains a master key. It has only 2 functions, a setter and a getter.. Testing on this is simply ensuring no null data is passed to it and that the data is stored and retrieved correctly. Important for the program, but individually not much to be tested.

PasswordManagerController.java - This is the brain of the program. This contains the functions called by the User Interface to register a user, login, add credentials, and delete credentials. If the user is attempting to modify anything in the database, their request is going through here. The tests performed on this were mostly equivalence class testing again. Ensuring valid inputs are processed, and invalid inputs are rejected. This ensures that the database manager does not get requests for incoherent data storage or retrieval. It is largely protected from that due to its own unit tests, but a little redundancy never hurts. There is a decent amount of integration testing happening as well, since any tests performed with these functions usually call on encryption processes as well as the database manager to process the requests.

The black-box testing methodology is largely used because the program is working with data input/output and storage. The internal logic is clear, and should be tested accordingly.

Overall, the testing done on the logic components of the program were quite thorough, and demonstrated, in my opinion, a very robust and rigorously tested system that is unlikely to be given any inputs it will struggle with.

PasswordManagerUI.java - The user interface.

This part was the hardest to test, since it was mostly a GUI. Directly testing the GUI proved to be a very difficult and unstable process. However, it is also the best way to perform system testing, since every element of the program is utilized at some point (even if very indirectly) when interacting with the UI. As a result, a combination of mockups, mirroring, and every other technique I could figure out to call functions from the UI without simulating inputs as if I were a user (like setting up an external program to simulate button presses on my keyboard and clicks from my mouse) was used. Unfortunately, due to the difficulty of writing these tests, they are split up into multiple test case files (unlike the other components which were all done in a single test file). Many of these tests integrate portions from practically the entire system, like testing the process of a user registering, then logging in, then adding a valid credential to their table.

The tests on this component show that the program works quite well. If the integration of these components failed, this portion would not work as well as it did.

3.3.2

The environmental considerations for this project are somewhat minimal ,but they are there.
A “digital notebook” to store your login information reduces the amount of paper that would be otherwise used to write things down on. Also, while indirect, hackers often target critical infrastructure in order to demand some sort of ransom, and this can be something like a chemical plant, an oil refinery, a water treatment facility, etc. Lowering the possibility of getting important login information stolen allows for less of these sorts of attacks to happen.

The societal impact of a program like this is very positive. It allows people to have a greater peace of mind knowing they are significantly less likely to be hacked and have their lives compromised. With the amount of critical information that we rely on private accounts for (banking, emails, health information ,etc), it has never been more important to be as secure as possible. This program can greatly improve that, and offer security to many people.

The safety of this program is very important. The encryption methods used ensure that the user’s data, and by extension the user, will remain safe. Their data will be safe, their passwords will be safe. They will face significantly less danger having their accounts secured. The reliability and security of the program (and the information of the users) was ensured by using industry-standard hashing and encryption methods, which have been tested and proven to be effective.

The economic considerations of a program like this are very simple. The program uses freely available tools, and anyone could use it. Also, having secure passwords significantly decreases the likelihood that you will be faced with something like your bank account being hacked, which can cause considerable financial damage and stress. Other types of ransom threats involving leaking sensitive information will be made much more difficult if the would-be victim has securely-stored passwords that meet proper security standards.

Overall, this program has a positive impact on all of these areas.

3.3.3

The test cases are too numerous to be listed here (and quite a few were explained earlier), and they will be explained more thoroughly in the later report document and presentation. However, the majority of the testing was ensuring that the encryption and hashing processes were safe, secure, and consistent, as well as the data storage and retrieval being reliable. This was done to ensure that no password would be incorrectly encrypted (and thus lost forever), stored non-securely (defeating the entire purpose of the program), and that nobody but the user, possessing the master password, could ever gain access to their data.

The rest of the testing was mainly ensuring that the GUI worked properly. While this was difficult to do with simulated systematic test cases, direct user interaction demonstrates that every aspect of this has been worked out. This will be displayed more fully during the presentation (since it's a GUI, systematic testing is likely not the best way to see if something is rendered correctly anyways).

Each class has its own test suite (and the GUI class has a few), which were all done using JUnit test cases. The tests were mostly done using maven, which allowed certain functionalities to be easily imported, and the tests to be easily executed. However, VSCode's built-in test functions were also utilized, and "testing with coverage" was done to see how much of the code was actually covered by the test cases. The overall result was 79%, just shy of the 80% goal that was set. However, since almost every part of the untested code was graphical elements of the GUI, I consider this a great success. The logical components of the system were tested very thoroughly, and I believe it has been proven that they are quite robust.

3.3.4 Limitations

Some of the limitations of this program are as follows. To start, this program is locally run only. There is no networking. While potentially good to reduce the likelihood of a database breach, this is also a problem because it prevents the user from accessing their information unless they are at their computer. This is partially remedied by the program (and database) being easy to transfer, but it is not a complete solution. From the beginning it was recognized that any sort of networking would put this project beyond the scope of a realistic 2-month deadline, so features like that had to be axed from the beginning. It would also make it significantly more difficult to test, which would be an issue given the purpose of this project.

There is also the issue of potential database corruption. While it may not be likely that someone with malicious intent could get access to the securely stored passwords in the database, it would be very easy for someone to erase them. And with no server containing a backup, you would lose access to all of your information. Even something like spilling a drink on your computer could lead to critical account information being lost. The creation of database backups is something that would improve the quality of this program, and give greater peace of mind knowing that your data does not rest on a single point of failure.

Finally, there is the issue of cross-device utilization. While I may be able to take the database & program on my computer at home and put it on another computer I use often to access whenever I need to, this solution does not extend to, say, a smartphone. One of the most likely times I will need login information will be when I'm on my phone, yet this program has no compatibility with one. Ideally, a sort of twin program could be developed for a mobile device that could interact with the database the same as this one can, thus extending the utility of the program. While java generally runs on almost any device, the GUI elements may not transfer well to a phone. Ideally though, this is a solvable problem that could be solved, and functionality could, in the future, be extended to smartphones as well.