TeamName

Michael Theisen, Jasharn Thiara, Patrick Tibbals, Trevor Tomlin

https://teamname-tcss360.github.io/

teamname.tcss360@gmail.com

Table of Contents

Introduction

The design we decided to go with so far is fairly straightforward and simple. After the program has started running it will open up the initial state of the program, which displays a window and gives the user the option to sign on. Behind the scenes, the code that executes this is our driver class which instantiates a fileGUI object and calls the method guiBuilder. Once the user clicks on the sign on button, the program’s state is updated to a login screen. Which again is done by our fileGUI which instantiates a new object of a login screen class and updates the window to represent this. At this point, if the user has logged in successfully they will be taken to the main state of the program.

So in conclusion we have a driver program that actually runs the program and calls on a method to build the GUI from the fileGUI class. Then the fileGUI acts as a controller to instantiate new objects of the different states we want to show in the program. Along with all of this, the fileGUI class will have access to the user class and the version control class to get the necessary information needed before arriving at a new state. Essentially, our program creates these more broad classes, like fileGUI, that encompass related classes.

Rationale Summary

The class design for our program is this way for two reasons: simplicity and intuitiveness. When considering creating a design for our program it made sense to have these broader classes, like fileGUI, to control and call upon sub-classes that are related to the object. This allows us to avoid having very large classes that would just be troublesome to locate a functionality that needs to be adjusted. So having these managing type classes not only provides a hierarchy to our structure, but in the case problems arise, the developers will know exactly where to investigate.

There are a few specific rationales behind each decision that was made in creating a design. Firstly, the reason we decided to create a general GUI class to manage other classes that interact with the GUI is that it allows us to organize all GUI components in a way that is easy to understand. If changes must be made specifically to the login screen, we are able to know exactly that whatever work that needs to be done will be done within that class. Also we believe that creating specific classes like these allow other developers who are new to the project to come in and understand what is going on. Our project also utilizes the FileTree class that will interact with a fileNode interface. We did this because the file aspect in our program is high priority. Also, assuming that the code for this will be extensive, since we have search, delete, and add file functions, the idea to create another class that acts as a manager will ultimately lead to more organization and a smoother visual of what is going on during these functions.

Another approach we took was to use a collection, HashMap, to store user information. This helps substantially because we no longer have to read through our CSV file every time someone logs in to ensure the username and password are registered. Instead, when we need a user's information, for example, whether or not they have editing permissions, we can turn to our hashmap and grab the users information. The pairing functionality is also very convenient because the username and password will be linked to each other and give easy access to check each of these during login checking.

Heuristic 3.1: Distribute system intelligence horizontally as uniformly as possible, that is, the top-level classes in a design should share the work uniformly.

The design will allow the program to have separate packages that will contain the graphical interface classes separate from the packages that will contain the classes responsible for managing the file system.

Heuristic 3.2: Do not create god classes/objects in your system. Be very suspicious of a class whose name contains Driver, Manager, System, or Subsystem.

God classes will be avoided by breaking down new panels into separate classes allowing a shared communication rather than returning all the data back to one main class.

Heuristic 3.3: Beware of classes that have many accessor methods defined in their public interface. Having many implies that related data and behavior are not being kept in one place.

Getters and setters will only be used for required information for functionality.

Heuristic 3.4: Beware of classes that have too much noncommunicating behavior, that is, methods that operate on a proper subset of the data members of a class. God classes often exhibit much noncommunicating behavior.

As the program is developed this heuristic will be revisited and necessary adjustments to the program's design and implementation will be made.

Heuristic 3.5: In applications that consist of an object-oriented model interacting with a user interface, the model should never be dependent on the interface. The interface should be dependent on the model.

The interface the user will interact with will be responsible for calling to the file system in order to generate the user's screen. This will keep the storage system separate from our interface unless the users are using functionality that will set or get information from the file system.

Heuristic 3.6: Model the real world whenever possible. (This heuristic is often violated for reasons of system intelligence distribution, avoidance of god classes, and the keeping of related data and behavior in one place.

Our design does try to follow this, as our current goal is to represent objects like users, floor models, warranties, etc. We believe that using this approach will keep our program intuitive for developers and someone who is new to the project.

Heuristic 3.7: Eliminate irrelevant classes from your design.

Although we only have so many classes at this point, as a team we are agreeing that any classes that are not necessary or at least as stand alone classes will be eliminated.

Heuristic 3.8: Eliminate classes that are outside the system.

Once the rough draft implementation is completed unneeded classes will be condensed or removed as the situation allows.

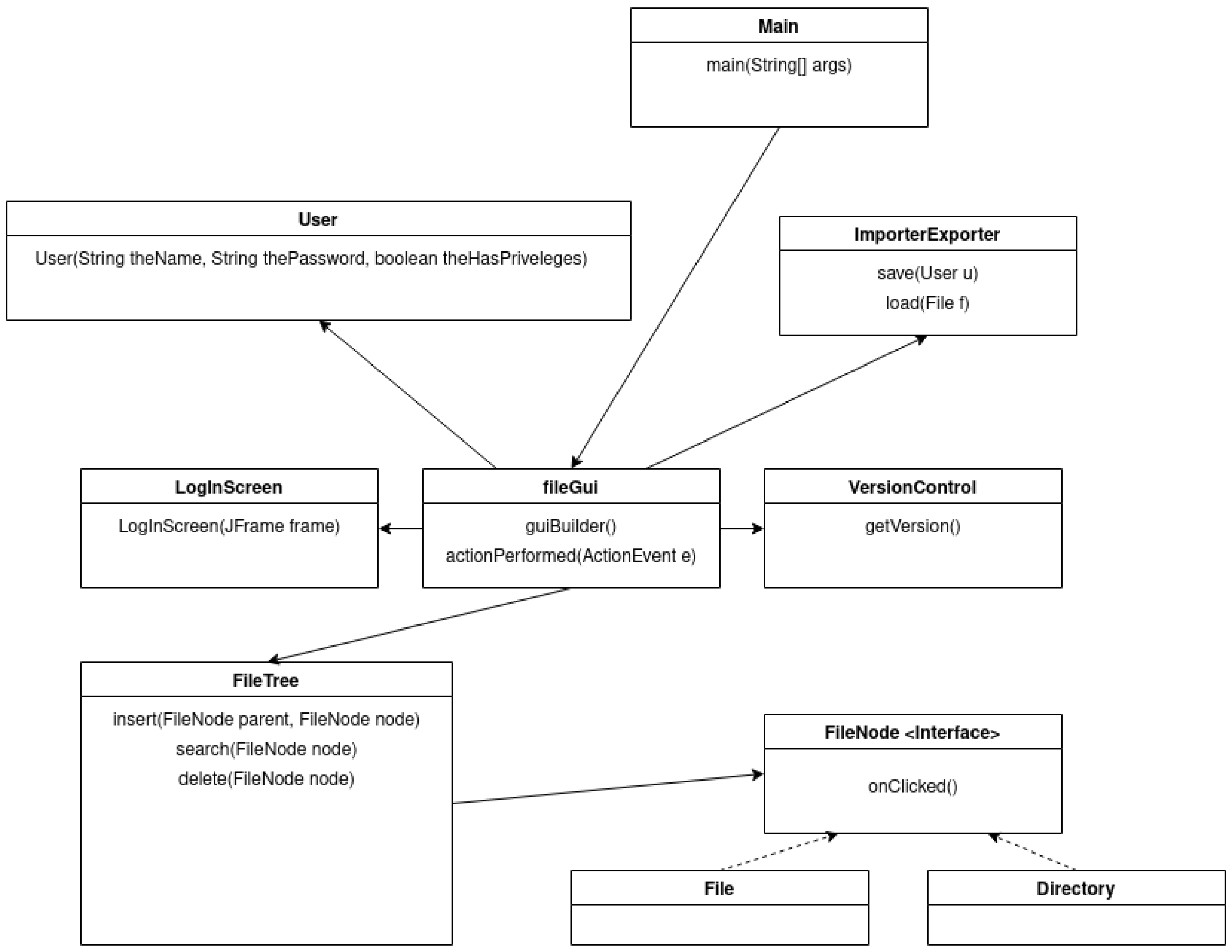
Heuristic 3.9: Do not turn an operation into a class. Be suspicious of any class whose name is a verb or is derived from a verb, especially those that have only one piece of meaningful behavior (i.e., do not count sets, gets, and prints). Ask if that piece of meaningful behavior needs to be migrated to some existing or undiscovered class

Our design does follow this Heuristic because rather than creating operations into classes, we are using an object-oriented design. Within these classes, methods may exist to represent these operations and interact with the object (Example File class -> Save() method).

Heuristic 3.10: Agent classes are often placed in the analysis model of an application. During design time, many agents are found to be irrelevant and should be removed.

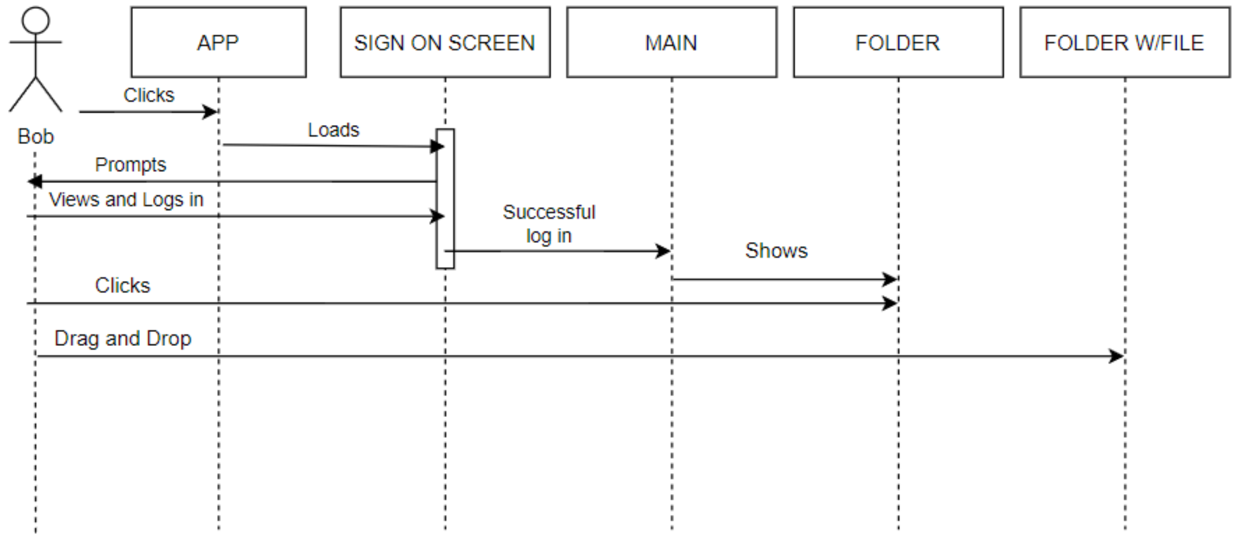
As the classes are established the design will avoid implementing classes that are solely responsible for communication between other classes.

Class Diagram



User Story Sequence Diagrams

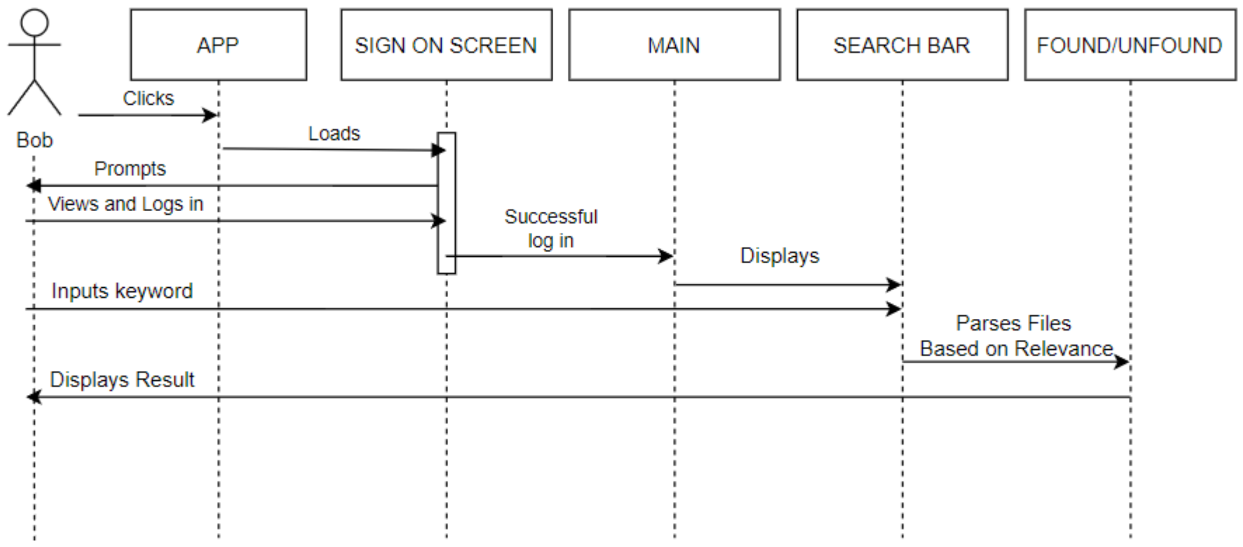
US01: As a user, I want to be able to store documents.



This sequence diagram explains the process of storing documents into the user interface. We have Bob clicking on the actual Application from the desktop that then loads the Sign On Screen. The Sign on Screen then prompts Bob to input his credentials. He sees this and enters his password that is then accepted by the Sign On Screen which then loads the Main page.

Folders are shown within the Main page. Bob clicks on one of these Folders, which ever he wants, and then Drag-and-Drops a document from outside the Application on the Desktop into the File where he wishes to store the document. Finally we have the Folder location that holds the document stored.

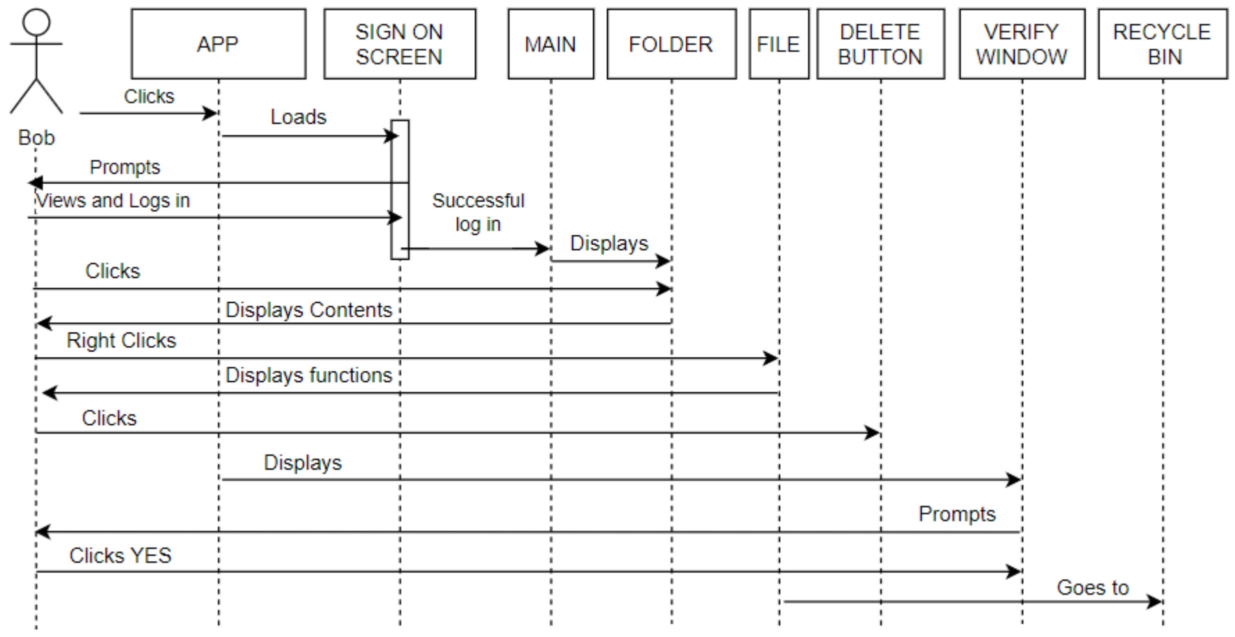
US02: As a user, I want to be able to use a keyword search.



This sequence diagram begins similar to the first for the sake of encapsulating each user story differently in its entirety. Here, Bob wants to be able to search for a particular file via a keyword. We have Bob clicking on the actual Application from the desktop that then loads the Sign On Screen. The Sign on Screen then prompts Bob to input his credentials. He sees this and enters his password that is then accepted by the Sign On Screen which then loads the Main page.

Inside of the Main screen, there is a Search bar that has a text box Bob can click on. When he types in the keyword he wishes to Search, the Application will parse through the files based on relevance to his Search keyword. Finally, the display will acknowledge that either the File was unfound or it will display the file for Bob to see.

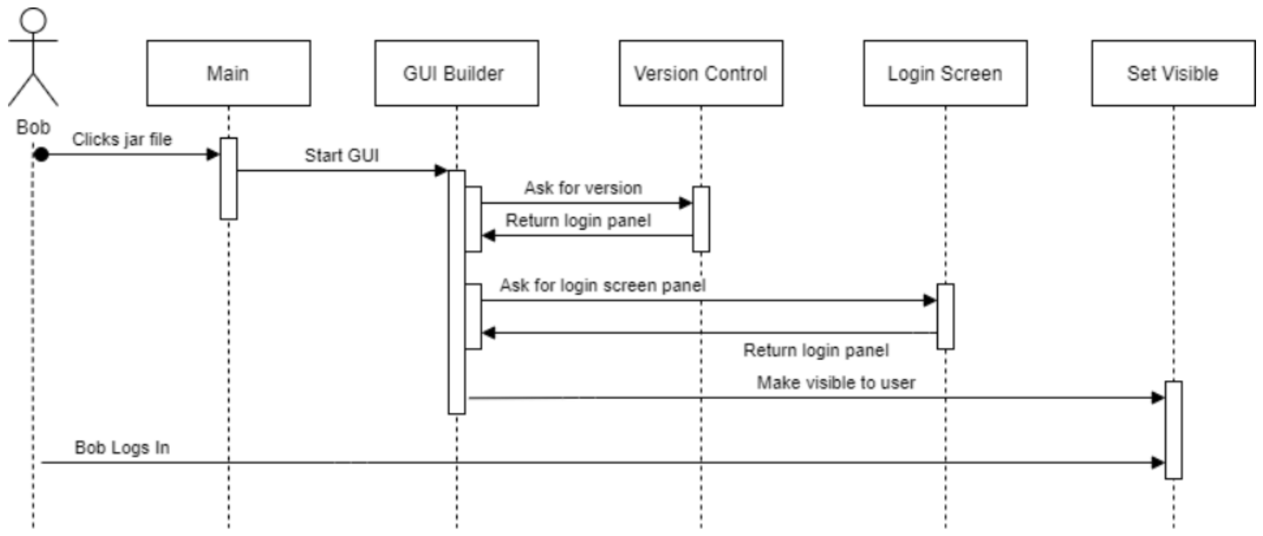
US03: As a user, I want to be able to delete old info.



This sequence diagram is similar to the first two. It explains the process of Bob wanting to delete old info. This will be done in the form of deleting a File. We have Bob clicking on the actual Application from the desktop that then loads the Sign On Screen. The Sign on Screen then prompts Bob to input his credentials. He sees this and enters his password that is then accepted by the Sign On Screen which then loads the Main page. This Main page will display to Bob all the available Folders.

Bob clicks on his preferred Folder and the Folder shows it's contents to Bob. He then right clicks on the File he wishes to Delete. A function bar is shown to Bob where he then chooses the Delete Button from the functions. The Application will then prompt Bob with a Verification window that Bob will see and click Yes to delete. The File itself is then sent to the Recycle Bin within the Application.

System Startup Sequence Diagram:



Bob clicks on the .jar file to open the program. This launches the main class which initializes the GUI Builder class. The GUI Builder will contact the Version control class to obtain the version number for the program. Then the builder will call out to the Login Screen class to return a login screen panel then the GUI Builder will set the panel to visible. Now the user Bob can begin to sign in.