Deliverable 2

TCSS 360

Group Emerald

Alex Ewing
Owen Orlic
Lucas Perry
Daniel Alberto Sanchez Aguilar

Github:

Group Emerald

https://github.com/lperry2/Group-Emerald

Contact:

lperry2@uw.edu
aewing24@uw.edu
danisan@uw.edu
oorlic14@uw.edu

Introduction:

Team Emerald's design for project partners is a simple application that helps facilitate the organization of events and projects for its users. From being able to modify one's project to one's own heart's content, a user can add and remove files, budgets, and notes through a navigable GUI. For our user login we establish an input for a user to place their name and when submitting their name, is stored within the program and searches whether that user has ever logged in before, where they would be considered a new user if they have not. From that input, the program would shape itself to the user's account where one can observe through the about button who is the owner of the current account.

Table of Contents:

Rational Summary:

Heuristic 1: Distribute system intelligence horizontally as uniformly as possible, that is, the top level classes in a design should share the work uniformly.

We do distribute system intelligence horizontally as uniformly as possible.

We do this by having our classes having unique respectives tasks. There isn't any class that is doing a lot more work than any other classes. The work is distributed across classes pretty evenly.

Heuristic 2: Do not create god classes/objects in your system. Be very suspicious of a class whose name contains Driver, Manager, System, or Subsystem.

We do not have any god classes in out system.

In our project, there are no classes that are meant to represent many different types of "things" or any that have many methods that are not related.

Heuristic 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.

We do not have many accessor methods within our classes.

In our project we do not have any classes that have more than 2 access methods.

Heuristic 4: Beware of classes that have too much noncommunicating bahavior, that is, methods that operate on a proper subset of the data members of a class. God classes often exhibit a great deal of noncommunicating behavior.

We are aware of classes that have too much noncommunicating behavior.

We are aware by creating classes that have decent amount of communicating and also we are currently limited to small communications by our current advancements within the project. There is no god class that tends to operate completely inside itself.

Heuristic 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.

We did follow this.

We did this by using setters and getters in our UI classes to be able to modify and display the model classes.

Heuristic 6: Model the real world whenever possible.

We did do this.

All of our class names are things you would expect to be involved in project planning and the operations of each class relate to those classes specifically and are not seemingly random.

Heuristic 7: Eliminate irrelevant classes from your design.

We do not have irrelevant classes in our design.

We feel that all of our classes are a representation of the important components needed to create an application for planning and designing projects, specifically DIY projects or renovation projects.

Heuristic 8: Eliminate classes that are outside the system.

We eliminated classes that are outside of the system.

We do this by only utilizing classes where we know we are going to use their functionality within our system.

Heuristic 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. Ask if that pieve of meaningful behavior needs to be migrated to some existing or undiscovered class.

We did do this.

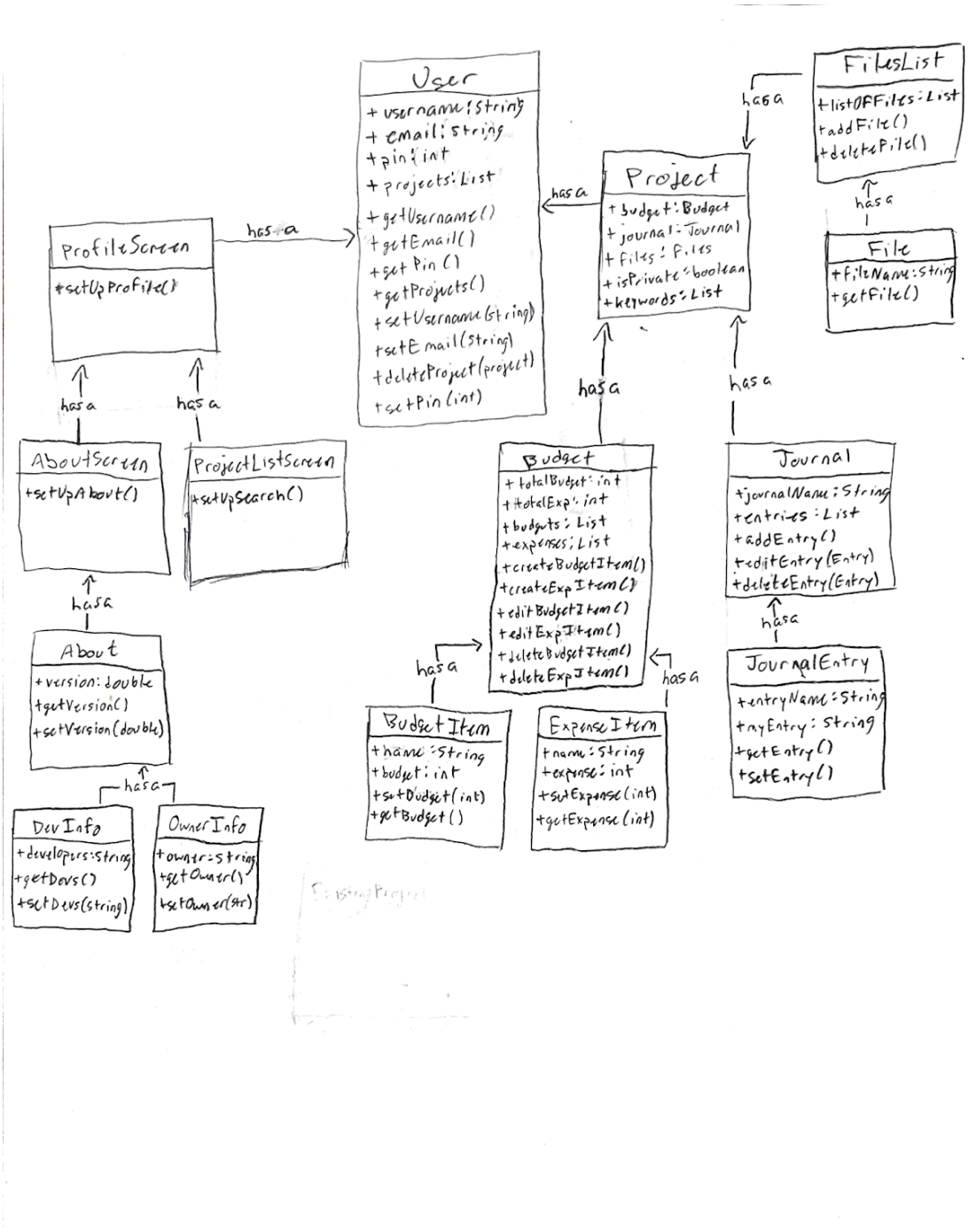
We had some classes with names such as AddUser and CreateProject which may have lead us to have the class really just be an operation. These classes need to be written in such a way that in the User class, a user can be created and in the NewProject class a project can be created.

Heuristic 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.

We did not do this.

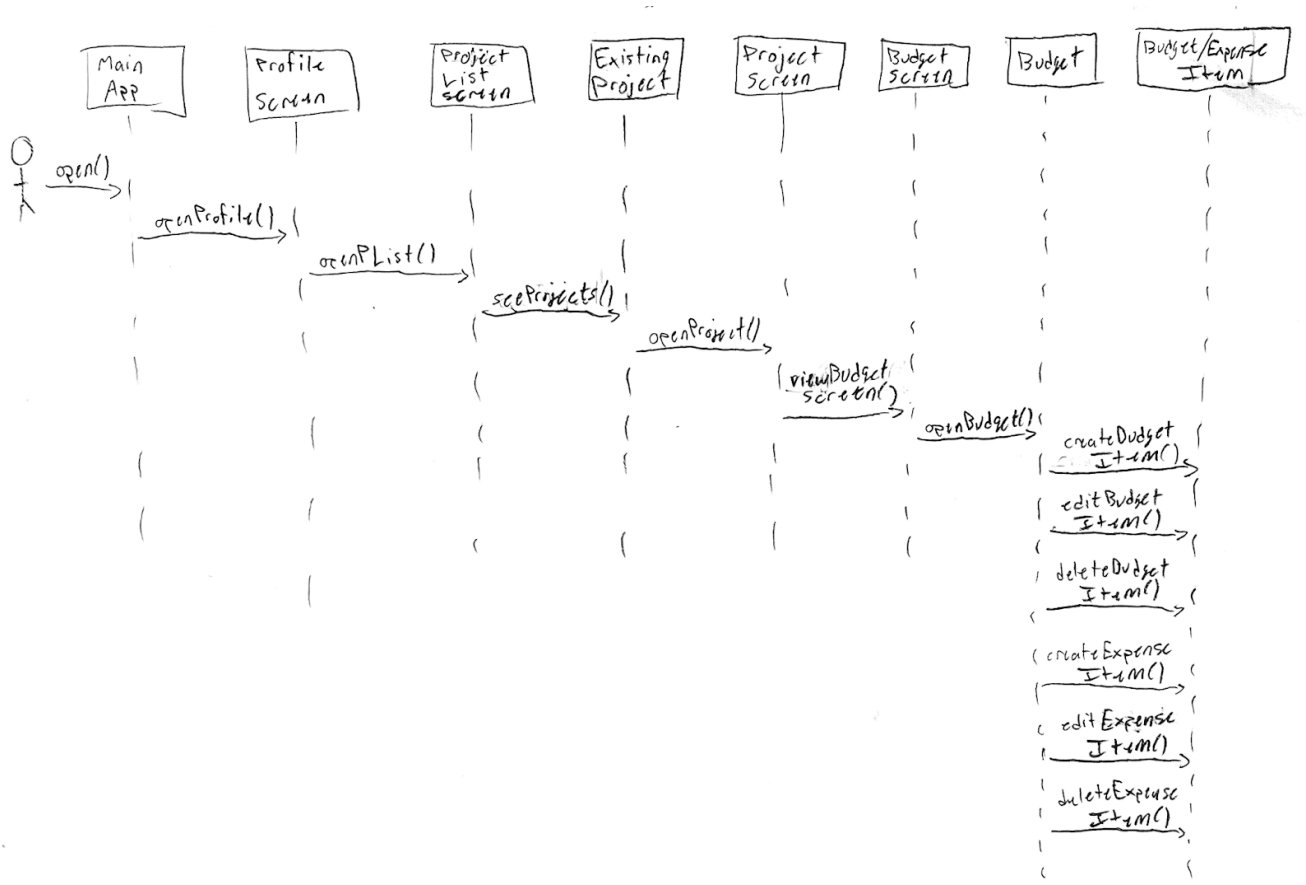
We may not be far enough into our project to find that agent classes such as these are irrelevant but as of now we don't have any classes that are completing behavior that could and should easily be completed by the another class (e.g. we don't have a writer class that adds our journal entries, the entry class just does it itself).

Class Diagrams:



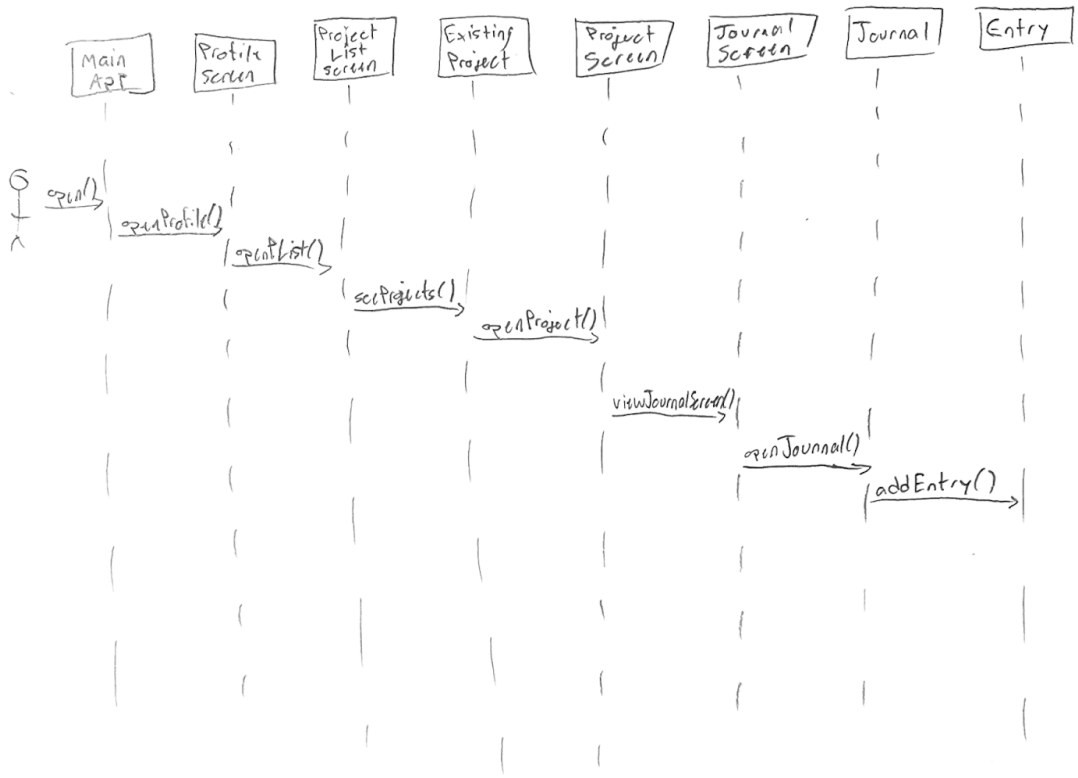
User Story Sequence Diagrams:

US01: “As a DIY enthusiast I want to be able to change my budget as my project plans change”.



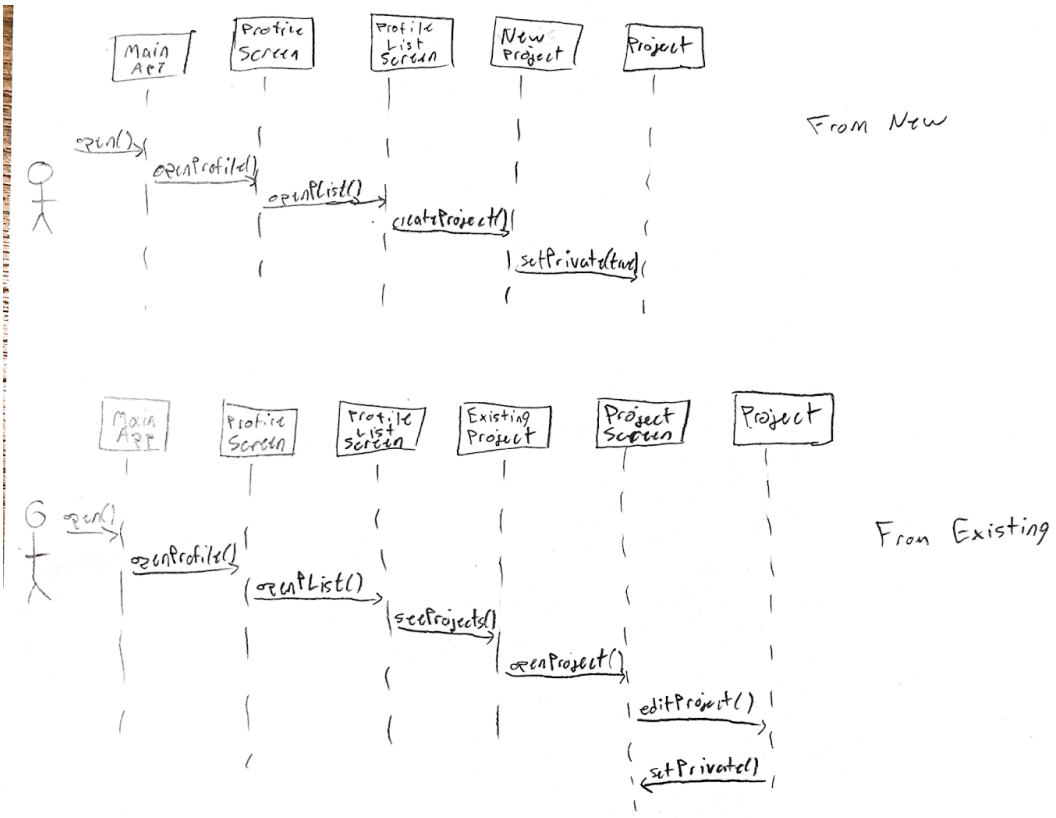
To add, change, or delete a budget, one must access the Budget class by navigating to the Budget Screen. From here, the user can edit budget or expense items via public setters and getters in the budgetItem and expenseItem classes, create a new budget/expense item with the constructors which only have one parameter which is the monetary value assigned to that item. Additionally, the budget class holds all the items in lists and can delete a budget or expense item from here.

US02: As a DIY enthusiast, I'd like a way to journal my plans, activities, and purchases so I can keep track of how my estimates compare to my actual costs and guide later financial decisions.



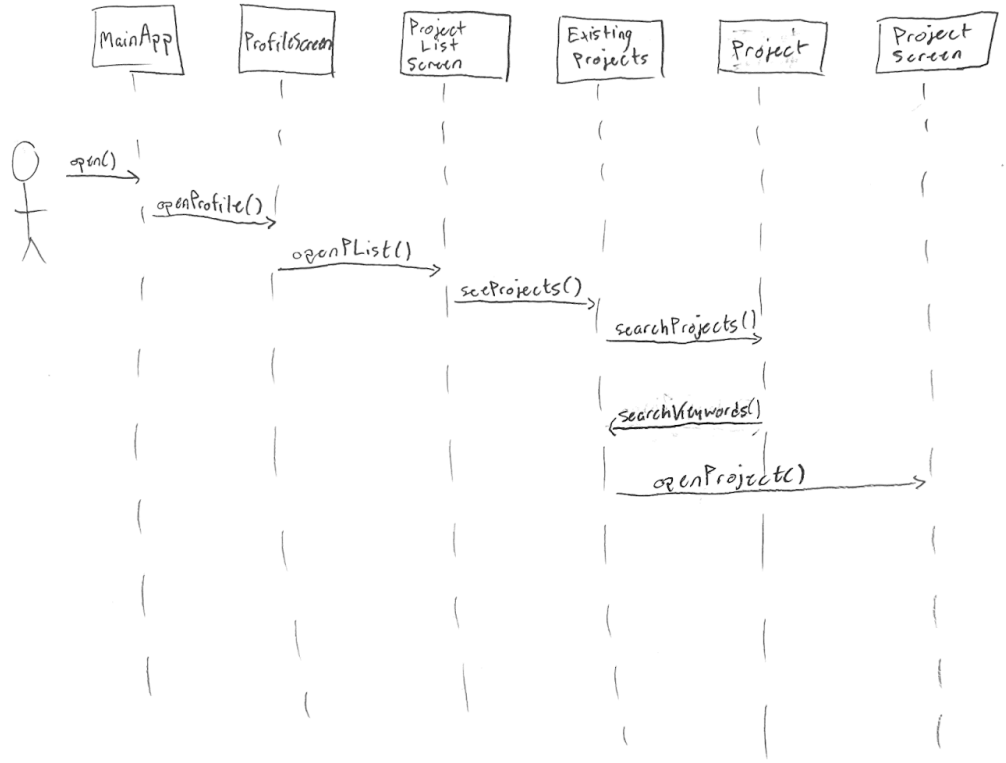
To create a journal entry, one must access the journal class by navigating to the journal screen. From here the user can create a new entry which will be added to a list, most likely a linked list, stored in the Journal class.

US03: As a family party planner I want to be able to keep specific projects private so I can plan surprise parties.



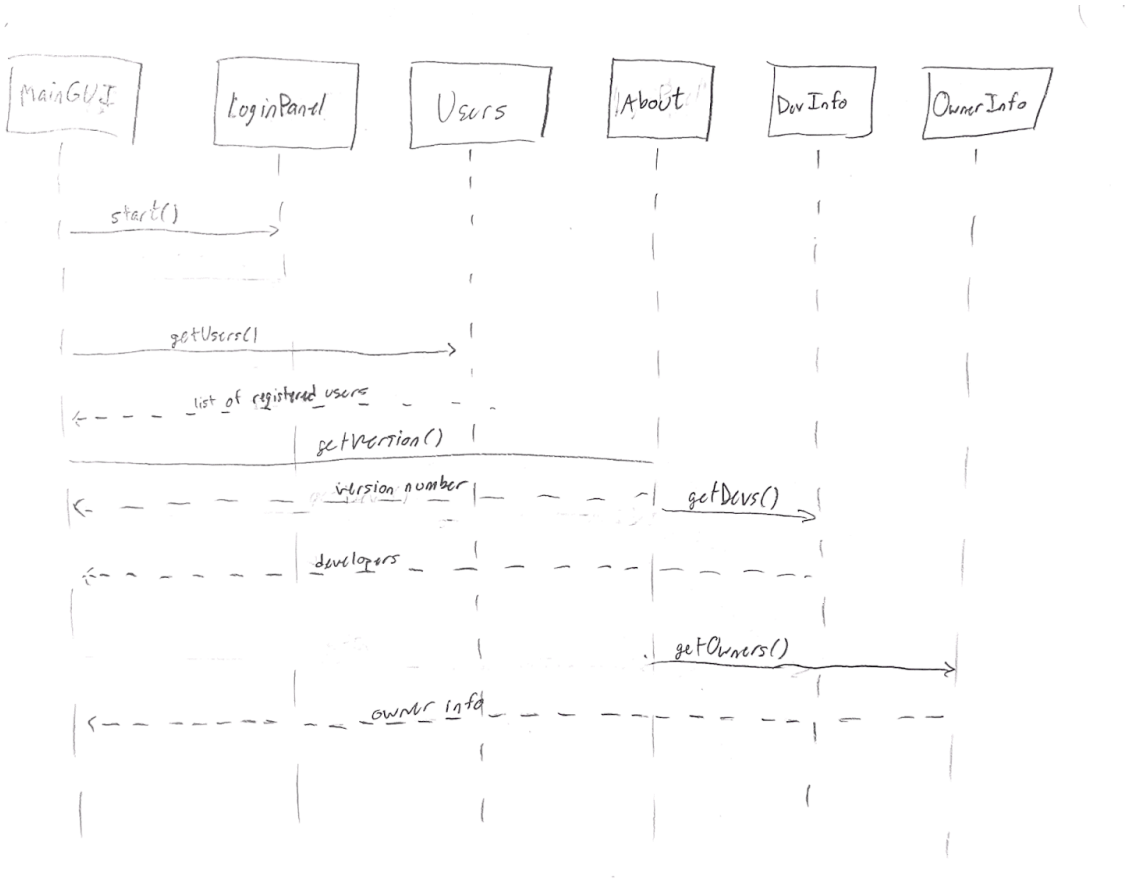
A project can be made private in two ways. If the user is creating a new project, the newProject class has a method that can set the project to private in construction. If a project is already made and is currently public, one has to navigate the project stored in the existing project class. Once the user accesses the project screen, they can edit the project and alert the project class to change its isPrivate field to true instead of false.

US04: As a DIY enthusiast, I'd like to be able to use tag words on my projects so I don't need to remember the exact name of my files.



Projects can be made with keywords so that the user can search terms related to a project to search for it instead of searching for the exact name. To do this one must navigate to the project list screen where a search bar is provided for the user. When given the option of creating a new project or opening an existing project, choose existing. Now the user can type in keywords for the project they've created. These keywords are saved in a list in each Project, so the search method will look through this sorted list of keywords and pull up only the projects that have these keywords. Then the user can open their desired project.

System Startup Sequence Diagram:



Once the program begins, Project Partner needs to prepare its login screen. MainGUI calls start which prompts the login panel to create the two text boxes, the about button, and the login button. We also have a file that holds all the users who login so our program can remember if someone has registered an account before. The only other thing that needs set up immediately is the about button so we need the data to use if it is selected. We get the Project Partner version number from the about class, the developer info from the DevInfo class and the owner info in the OwnerInfo class, even though at this point the owner is no one because no one is signed in.