ChatGPTatHome

Anthony Chapkin, Hai Duong, Jeremiah Brenio, Windie Le
https://github.com/ChatGPTatHome/Project
ChatGPTatHome@gmail.com

Table of Contents

Introduction

We decided to use the model/view/controller philosophy in our design. The Models class will provide data to the Screens, giving a screen a reference to the model they need by using a hashtable of all classes in the model package. Our design will utilize JSON files for persistent data, which will be accessed for the sake of retrievement and mutation by classes in our model package. Our controller package will only have a single class called ProjectHub, that will create and contain an instance of MainFrame, and will also direct this MainFrame to create and add certain GUI elements to itself, like a factory.

All of the names of the GUI classes in the view package will be appended with “Screen”, except for MainFrame. These classes will inherit from the abstract class Screen, which will define a shared behavior between these classes, and will also provide a static method for creating a Screen, which will be utilized by MainFrame. The Screen classes will also have a reference to a Models object created in MainFrame, that they will use to update or fetch data from their respective model classes. This also means their models will share states. For example, the AboutScreen will take in a Models object, and from the Models object, it will also obtain a reference to an About model. This About model is the same one any other Screen might need to access or mutate.

Rationale Summary

We decided to go with a model/view/controller philosophy because we wanted each GUI component to have its associated model, making the code less coupled and more cohesive. Since our design will utilize JSON files for persistent data, we will create our model classes around it so that they can give data appropriately to each Screen GUI that needs it.

We want to go with a Screen abstract class and also pass in a reference to a single Models class to our UI classes to implement similar behavior and a shared state between them. This would allow us to easily scale or add to our program and ensure our data stays consistent between all objects and files. We will use a naming convention for each GUI component presented to the user by appending “Screen” to the view classes for consistency. MainFrame will be an exception, as it will be a factory to create all the screens and add them to one unified window frame. We did this for the sake of scalability, as ProjectHub no longer has to worry about creating the Screen classes itself. The reason we want to split our view package into multiple Screen classes is for the sake of organization, modularity, and scalability. It makes it easy to add a new Screen, make a change, debug, as well as split workload and responsibility between developers.

As for creating a project, the division of tabs in the workspace was given positive feedback by users of our paper prototype. So following that, we will create four Screen classes, TaskTabScreen, MatTabScreen, ToolTabScreen, and CostTabScreen, that will act as separate tabs for our workspace, each with its own model, TaskTab, MatTab, ToolTab, and CostTab respectively.

Vis-à-Vis Riel's 10 Design Heuristics

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.

FOLLOWS: Our program has a highly modularized model and view packages. That is, the workload is split mostly evenly and in a way that makes sense from an organizational point of view, between our top-level classes.

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.

FOLLOWS: We don't have any classes with suspicious names. Our most suspicious class is MainFrame, which does a fair bit of work, but does not overstep its boundaries.

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.

DOESN'T FOLLOW: Our model classes have to contain accessor methods in their public interface for the sake of sticking to a model/view/controller design pattern. This data needs to be accessed by view classes for their function, but our shared Models class makes sure all this data is consistent and in one place.

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 a great deal of noncommunicating behavior.

DOESN'T FOLLOW: Most or all of our classes in view have access to the same Models object. From this Models object they are able to access models needed by them, meaning, they only have members they use themselves. However, some of our model package classes have data stored that they themselves will never use, though most will at some point use them to update persistent data.

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.
FOLLOWS: Our program follows the model/view/controller design pattern, which includes a one-way communication from our view to our model. Meaning, our model is not dependent on our user interface.

Heuristics 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.)
FOLLOWS: Our project is modeled after a real-life file cabinet, where we have different papers with tabs stuck on them that contain certain information useful for some sort of project.

Heuristics 3.7 Eliminate irrelevant classes from your design.
FOLLOWS: One could argue that Project and the different Tab model classes are irrelevant, and we should either just have the Project class or just the Tab classes. We split them up for the sake of organization and relation with their respective ScreenTabs, and for the sake of avoiding unnecessary duplication of code in each Tab model, that could simply be placed in Project. This is also the case for system calls to retrieve data from our JSON files, as it can now be done solely by Project. Not splitting them up in Tabs would also require us to potentially hard code the keys in the TabScreens.

Heuristics 3.8 Eliminate classes that are outside the system.
FOLLOWS: A Folder model might not be in the domain of our program, and everything it needs to do is covered by the File object. But, because we need to keep the current path between our classes consistent, and we need a way to change which directory we are in or what our topmost directory is, it is justified in our eyes.

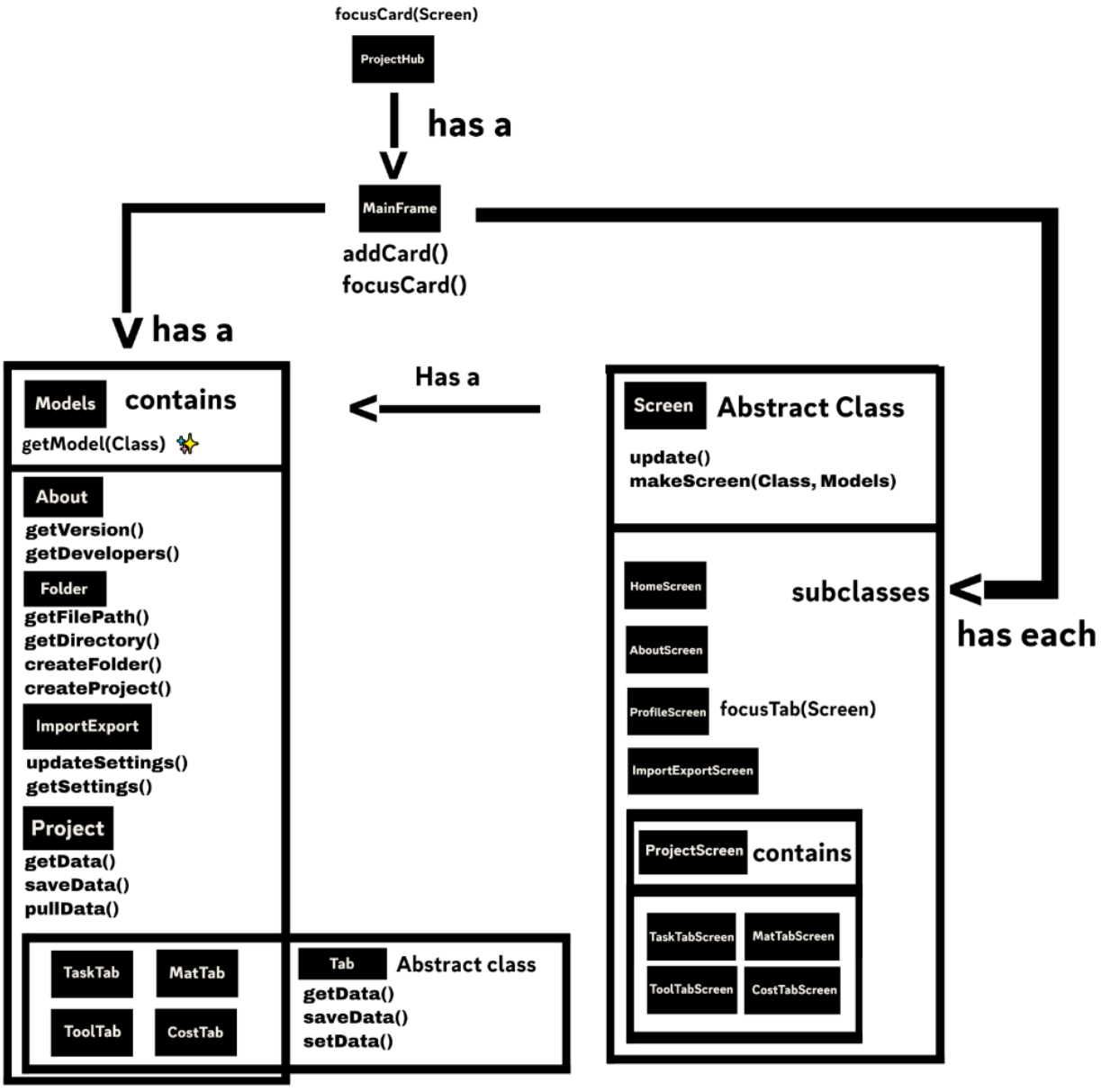
Heuristics 3.9 Do not turn an operation into a class. Be suspicious of any class whose name is a verb or is derived from a verb, especially those that have only one piece of meaningful behavior (i.e., do not count sets, gets, and prints). Ask if that piece of meaningful behavior needs to be migrated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

DOESN'T FOLLOW: Our About model class only really fetches data from our persistent data storage, and then allows other objects to access that data. We decided to have it anyway just because it makes sense from a grouping or functionality standpoint (i.e. We need this functionality but we can't really combine it with another class in a way that would make sense to us).

Heuristics 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.

FOLLOWS: One could say that Models is an agent class, but you could say for any factory class. The reason we want Models to exist is to have a neat way to store all our models, and also create them if they have not already been created, as well as an easy way to share state. Project might be an agent that fetches data from the persistent data for our TabScreen classes, but the reason we included it is to limit system calls, and not have to implement a way to access the persistent data in each Tab model.

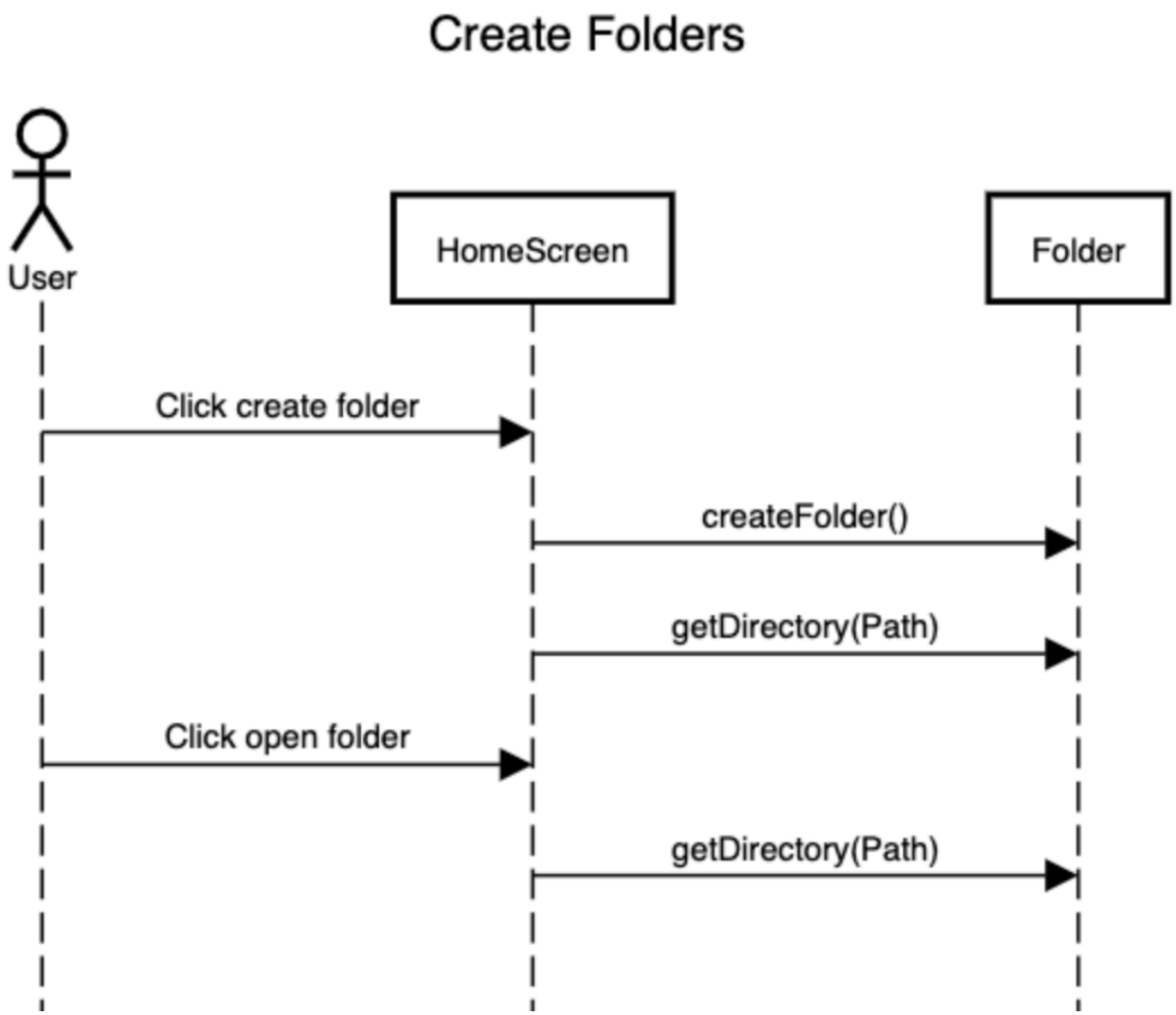
Class Diagrams



User Story Sequence Diagrams:

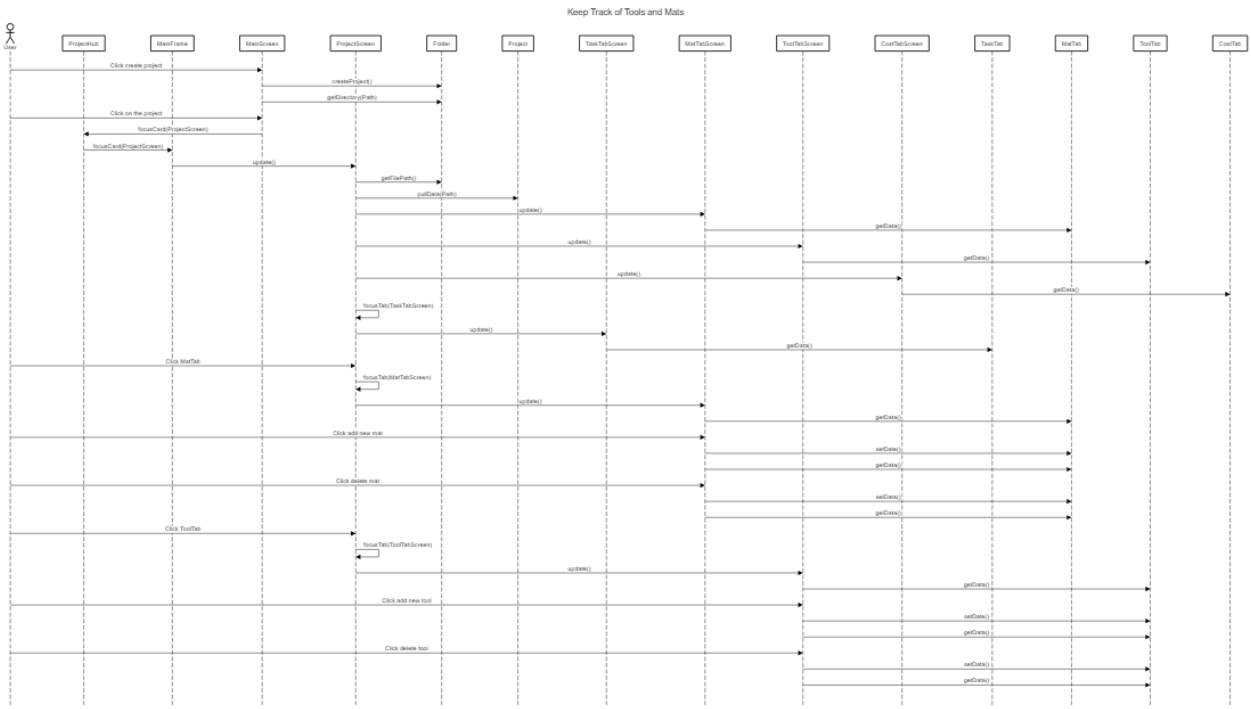
US01: As an organized user, I want to be able to easily create folders.

sequencediagram.org Link: Create Folders



The sequence diagram represents the process of creating and accessing folders. Initially, the user interacts with the home screen by clicking on an option to create a folder, triggering the createFolder() function. This function presumably involves creating a new folder in the system's directory. Once the folder is created, the user can then click to open this or any other folder, which leads to the execution of getDirectory(Path). This function retrieves and opens the directory specified by the path, allowing the user to access its contents.

US02: As a DIY enthusiast, I want to keep track of tools and materials used for a project.

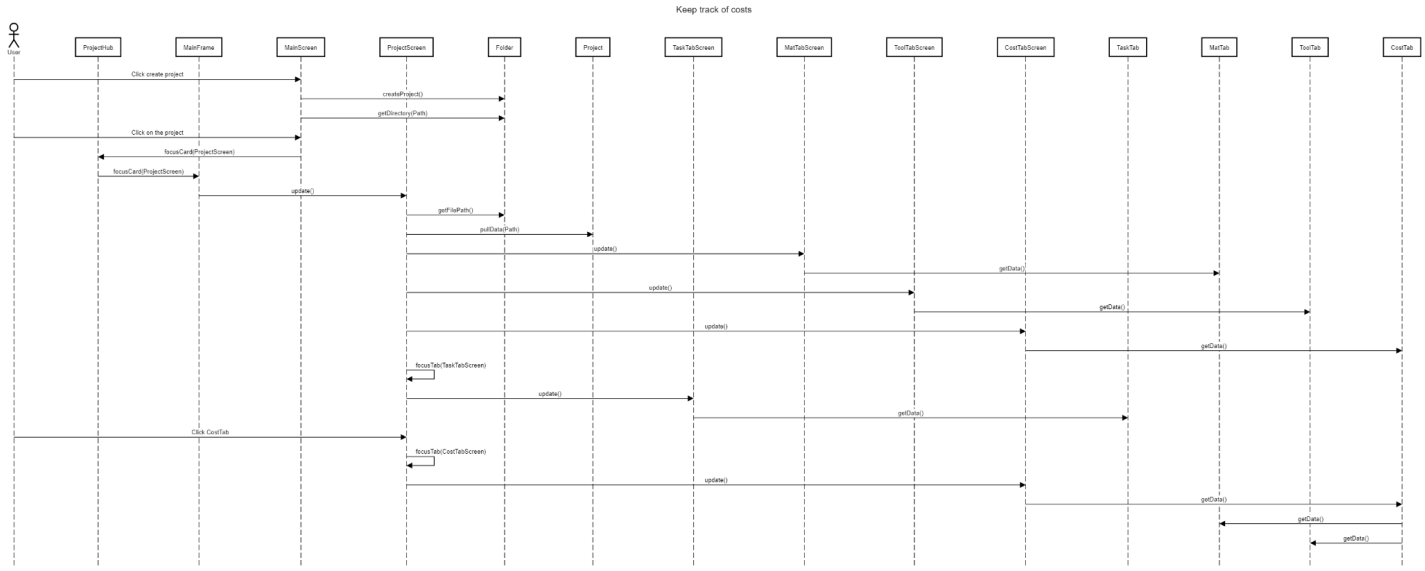


sequencediagram.org Link: [Keep Track of Tools and Mats]

The diagram outlines a sequence in a software system designed to help a DIY enthusiast track tools and materials for projects. For tracking projects, the user navigates to the ProjectScreen using focusCard(ProjectScreen) and then focuses on the materials tab (focusTab(MatTabScreen)), where they can retrieve (getData()), modify (setData()) material details. Similarly, for tools, the user switches to the tools tab (focusTab(ToolTabScreen)), where they can access (getData()), update (setData())

US03: As a DIY enthusiast, I need a way to keep track of project costs.

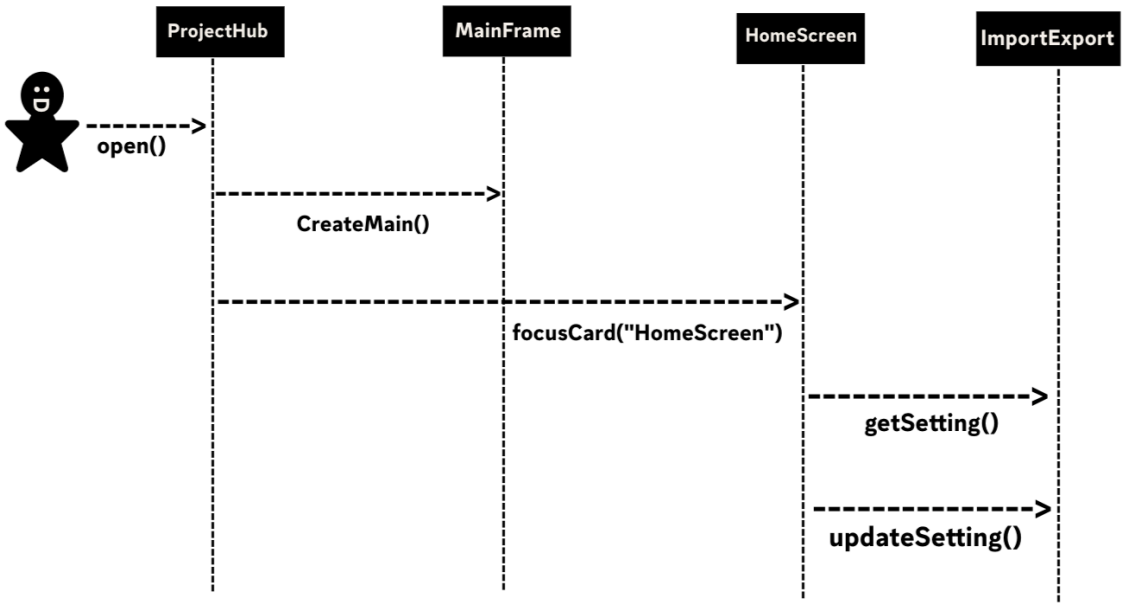
sequencediagram.org Link: [Keep track of costs]



The user starts at HomeScreen. They click to create a new project. Then they open the project. Next, they click the CostTabScreen. Now they see the costs of the project.

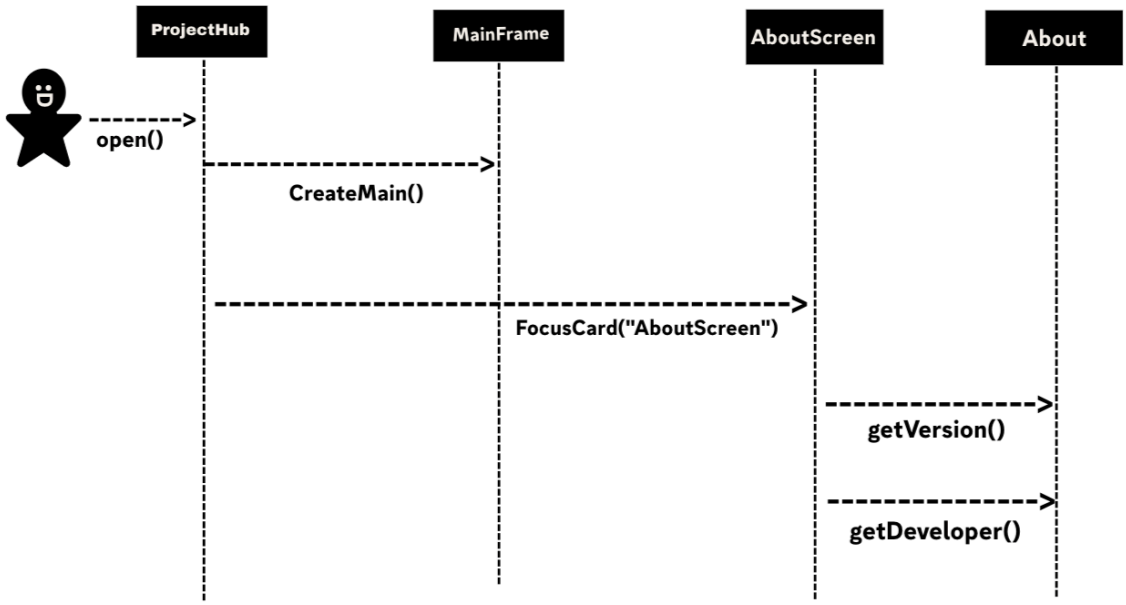
The sequence diagram describes a software application process that enables a DIY enthusiast to manage and track project costs. The process begins when the user creates a new project via the MainScreen, which involves creating a project in a Folder and retrieving the directory path. Upon selecting a specific project, the application focuses on that project within ProjectScreen by updating various components associated with the project's structure, including tasks, materials, tools, and costs. As part of managing project details, data is pulled and updated for materials, tools, and tasks from their respective tabs. The user can specifically view and update cost-related data by clicking on the CostTab, which triggers the CostTabScreen to fetch and aggregate cost data from both the MatTab and ToolTab. This setup facilitates detailed tracking and updating of project costs, essential for budget management in DIY projects.

USX01: As a user, I want to enter settings such as my first name and email address.



The diagram illustrates the sequence of actions a user follows to enter and update settings such as their first name and email address. Initially, the user opens the application, as depicted by the open() action. This triggers the creation of the main interface (CreateMain()) of the application, shown as the MainFrame. The user then navigates to the home screen via the focusCard("HomeScreen") action, suggesting a change in the user interface to display the home screen. Within this screen, the user accesses existing settings through the getSetting() method and subsequently updates these settings with the updateSetting() method, likely through a settings form on the HomeScreen. These actions allow the user to modify personal details stored in the application.

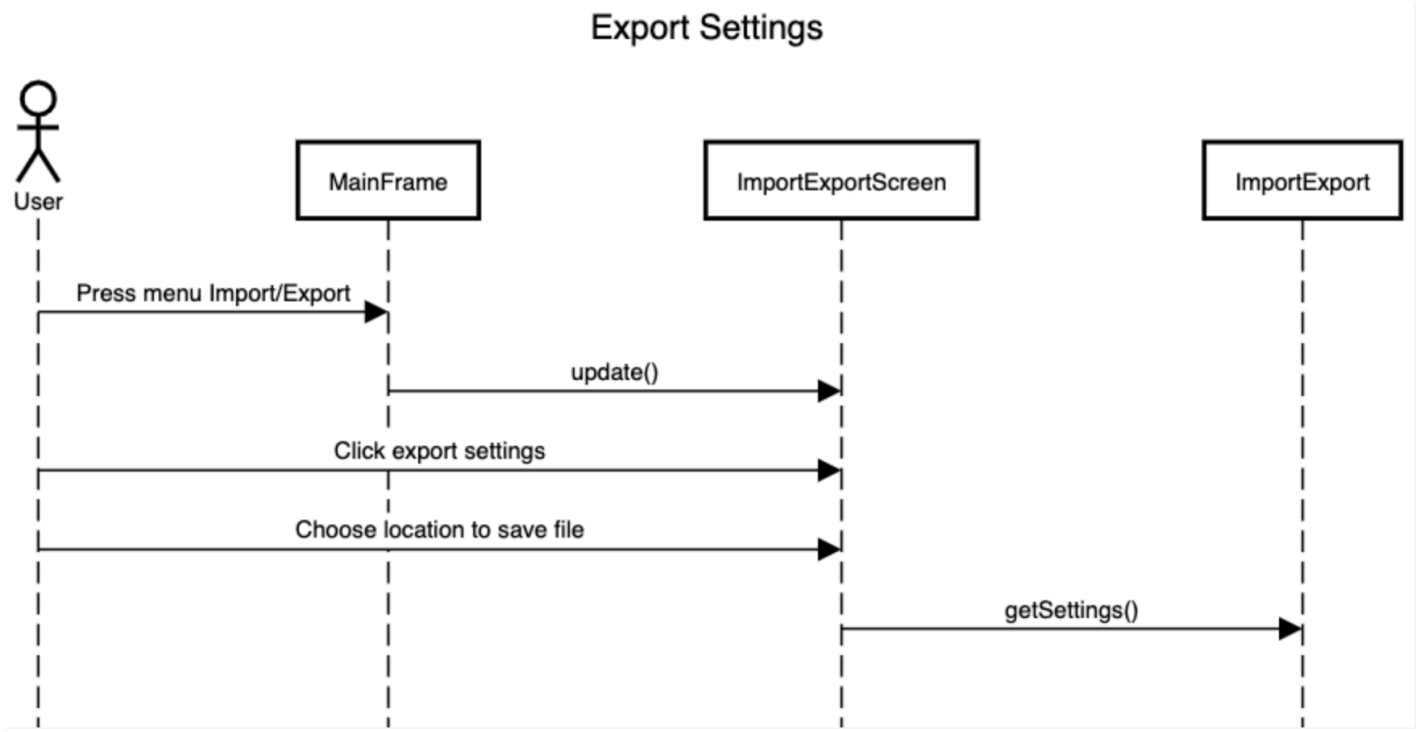
USX02: As a user, I want to see the version of the software and other information such as the names of the developers.



The diagram represents the flow of interactions in a software application from a user's perspective, aiming to access the version and developer information. When the user opens the application (signified by the open() action), the application's main interface, represented by MainFrame, is created through the CreateMain() method. Subsequently, the user navigates to an "About Screen" via the FocusCard("AboutScreen") action, which likely shifts the focus to this specific part of the interface. Once on the "About Screen," two operations are performed: getVersion() and getDeveloper(). These methods are responsible for retrieving the software's version and the developers' names, respectively, providing the user with the requested information.

USX03: As a user, I want to export settings (name and email) for synchronization to other devices

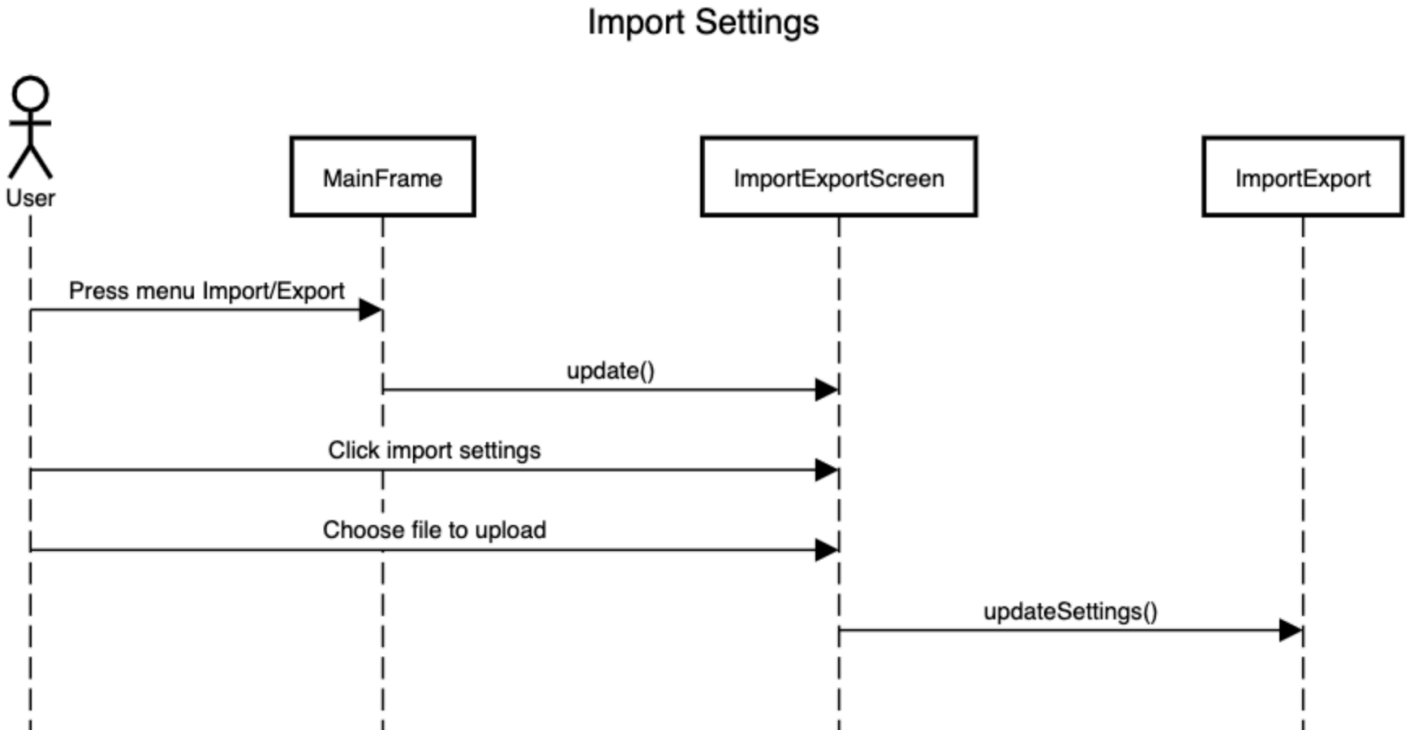
sequencediagram.org Link: [Export Settings]



The sequence diagram outlines the process for exporting settings. Initially, the user accesses the export function by selecting the "Import/Export" menu in the main application frame. This triggers an update in the ImportExportScreen, preparing it for the next actions. The user then initiates the export operation by clicking the "export settings" button on this screen. Following this, the user selects a location to save the export file. Once the location is confirmed, the ImportExportScreen requests the current settings from the ImportExport component through the getSettings() function. This function is responsible for retrieving the user's settings (such as name and email) for export.

USX04: As a user, I want to import settings to synchronize with other devices.

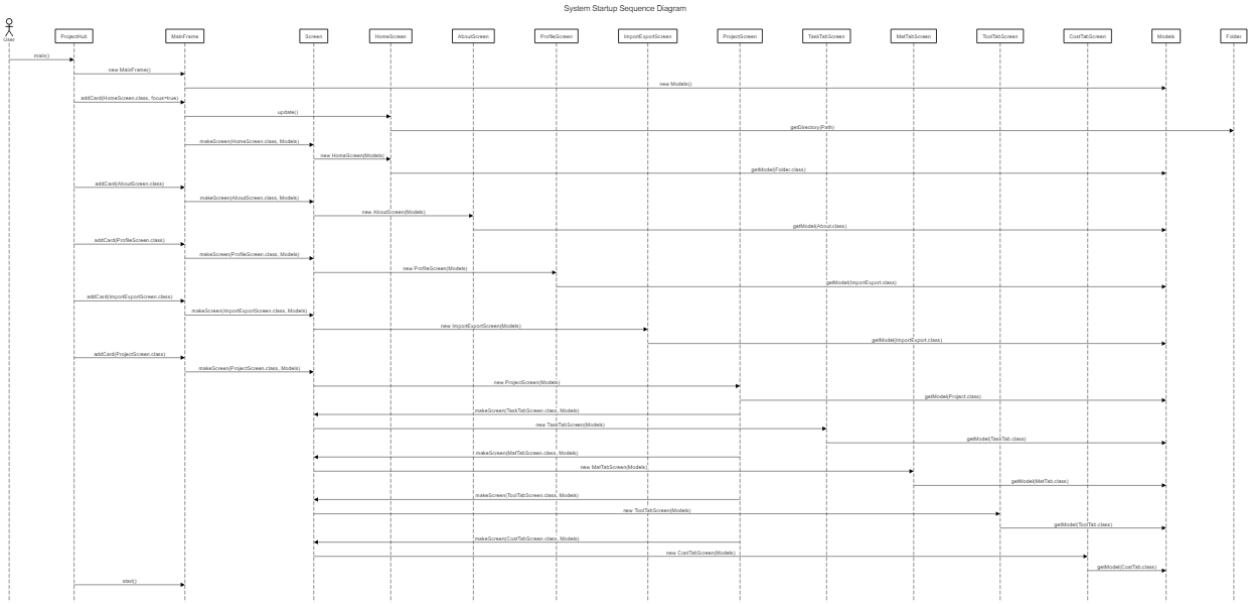
sequencediagram.org Link: [Import Settings]



The sequence diagram illustrates the process for importing settings. Initially, the user navigates to the Import/Export menu in the application's main frame. This action prompts an update of the ImportExportScreen. From this screen, the user initiates the import process by clicking the "import settings" button, followed by choosing a specific file to upload. Once the file is selected, the ImportExportScreen communicates with the ImportExport component to invoke the updateSettings() function. This function processes the uploaded file, updating the application's settings with the data extracted from it. This allows the user to synchronize their settings with other devices.

System Startup Sequence Diagram

sequencediagram.org Link: [System Startup Sequence Diagram]
Google Drive link: [System Startup Sequence Diagram]



The diagram represents a system startup sequence for a software application where a user initiates the process. The sequence starts when the user triggers the main() function, which in turn creates the main interface of the application (new MainFrame()). What follows is that each Screen is created and added into MainFrame. Each time a Screen is created by MainFrame, it is passed a Models reference that resides in MainFrame. Through Models, each Screen accesses and also indirectly creates their respective models. This is also true for Screens not found in MainFrame such as the TabScreen series of Screens.