Mao and The Zedongs (毛泽东)

Design Deliverable

Samuel Adams, Mike Fly,

Dylan Hill, James McHugh

https://github.com/SlowNibbler/1989-Tiananmen-Square

Contact:

sdadams@uw.edu

flym@uw.edu

dhill30@uw.edu

jamesm47@uw.edu

Introduction

Mao & The Zedongs is a team comprised of Dylan Hill, Samuel Adams, Mike Fly, and James McHugh, who are working on a solution for a problem all D.I.Y. enthusiasts are sure to face. When working on a project, it's hard not to wind up with seemingly endless instruction manuals, rough sketches, and every other kind of paper imaginable. Often times it's even harder to keep all of that paper from getting lost or being misplaced. That is the kind of problem that we aim to solve. Our application will allow users to create digital project folders where they can upload any documents they have pertaining to a certain project and have them all stored in a single organized place, allowing for easy access.

Table of Contents

Rationale Summary

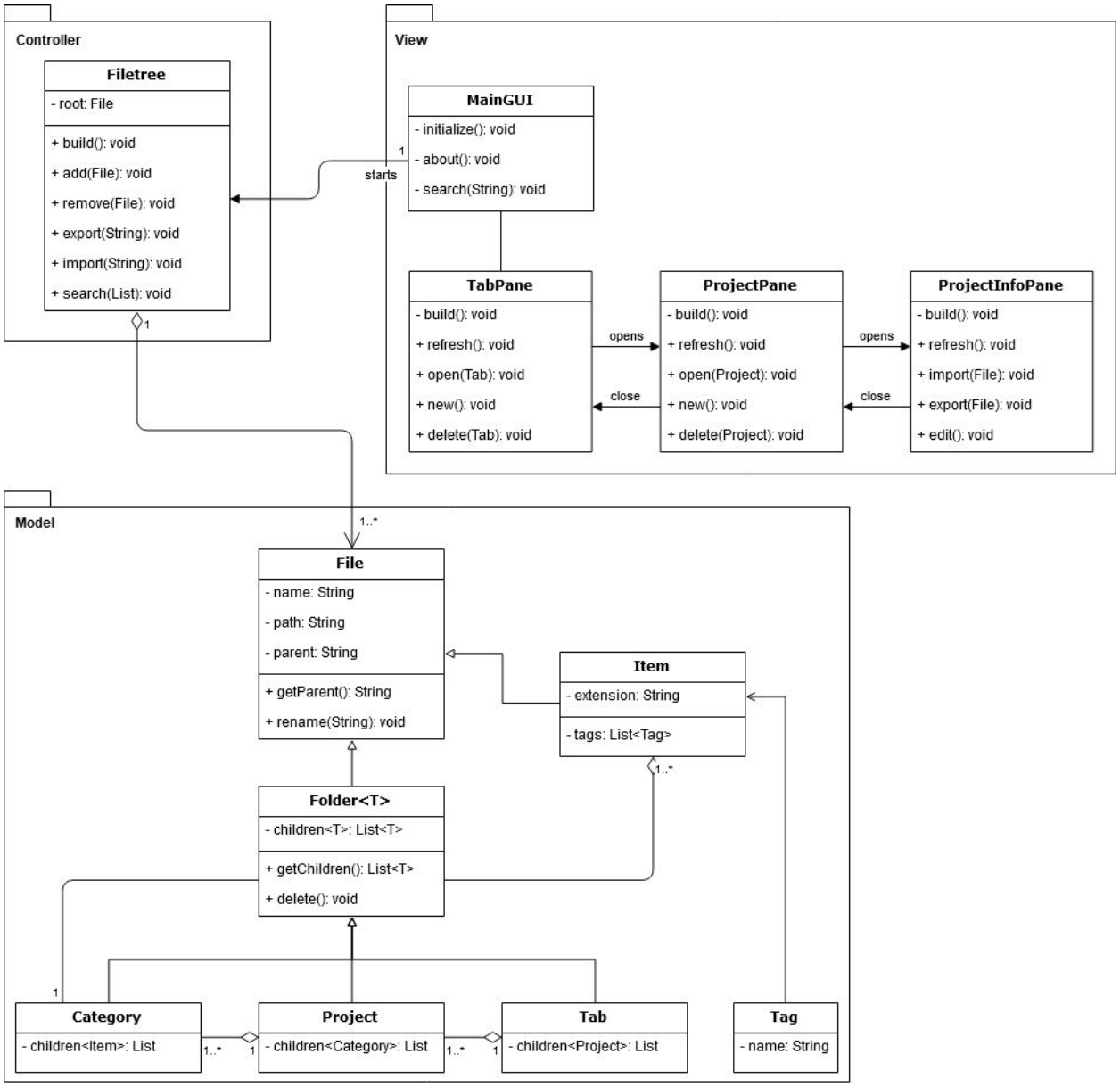
When we first approached this project, we had a pretty good initial idea of how to put everything together. The basic functionality of our program is essentially the same as any other file browser so we have been building our classes in a similar structure. At the root of our project, we have the two most important classes; File and Folder. File is the highest level parent class for all of our backend file representation and is extended by most all other classes which make up our file tree system. The reasoning we used to reach this conclusion was that most every element in our backend will need to be tied to an actual file path somewhere, so it would be best to have everything be considered as a "file" with its own file path. Even the folder class is a child of File since even folders have file paths. The primary difference between the two classes is that folder has an array list of "contents". We decided that using an array list to hold the files of each project would be best given the need to dynamically add and remove items from any position in the list when a user adds or removes files on their end.

In regard to Riel's 10 design heuristics:

1. We have made an effort to balance out the work of our top-level classes to follow the first design heuristic. Although it could change as we add more and more implementation and functionality, our current model has most of the work spread out across our collection of backend classes which represent files, folders, tags, and more.
2. The only thing close to a "God Class" in our program currently is our GUI main. Since we have done most of the GUI building using the Eclipse window builder plugin, most everything about the GUI has been automatically built in the single class. We will keep an eye on this during further iterations and split it up if we deem it to be beneficial.
3. Our individual classes have a pretty limited accessor methods in order to be in line with the third design heuristic. There are cases in which a class will need an accessor method in order to get the name of a project or item to be displayed in the GUI, or when the user wishes to edit the name or other properties of an item, but that has been the major extent of our accessor methods.

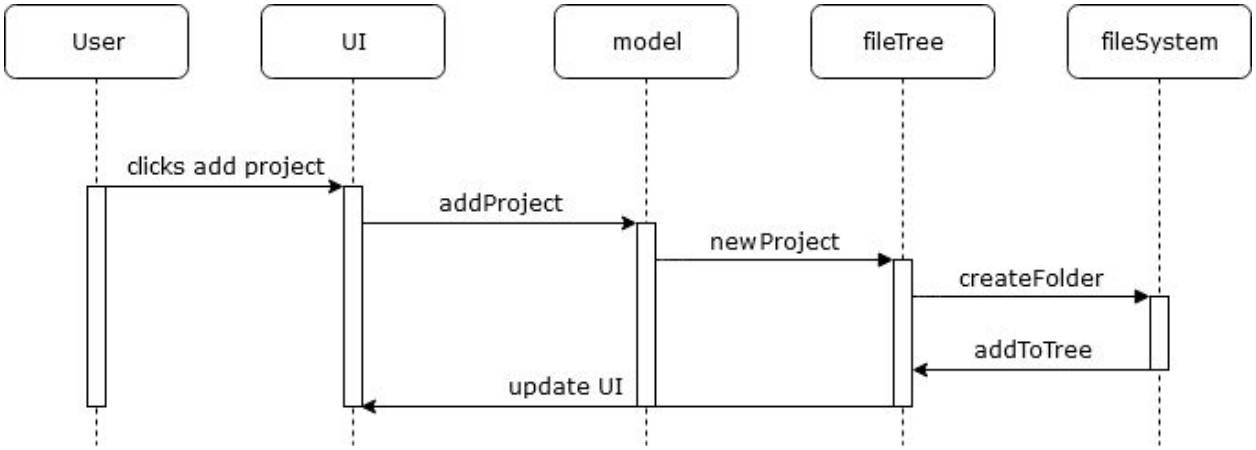
4. The majority of our classes communicate with other classes in some way or another. This is mostly due to the hierarchical structure that we are building our program around. We tried to create most classes to have a simple parent/child relationship to ensure plenty of interaction between the two.
5. In developing our user interface, we have made sure not to have any instances of the model being dependent on the interface. The interface is dependent on the model, as the list of projects is generated based on the actual backend list of project objects.
6. Since files and folders actually do exist in the real world, we have done what we can do to make our digital representations make sense as representations of the real deal. Files and folders only contain information that would make logical sense for them to contain.
7. All of the classes that we have made serve a purpose to the program as a whole so we have no irrelevant classes. We strived to have a clean, easy to follow, design structure.
8. We have only designed the classes that we feel are necessary for our project so there are no classes that we feel are "outside" of our system. Again, we tried to create code that is efficient and to the point.
9. Each of our classes is named after a noun that is an essential aspect of our program. We have made sure to make as much usage of the object classes we have built as possible in order to deter any single-operation type classes from popping up.
10. We set up the plans for this project with the hope of avoiding too many useless agent classes and so far we have not had to remove any. Although still subject to change, we are confident that the layout we currently have will be able to carry us through to the end.

Class Diagram



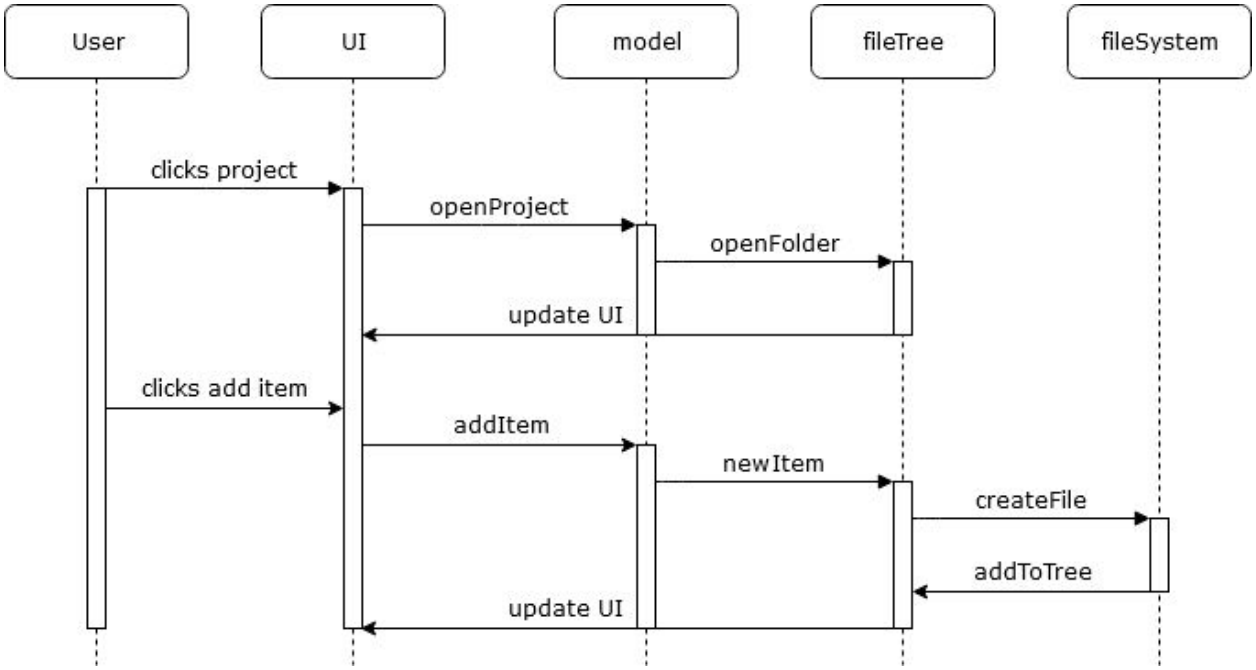
User Story Sequence Diagrams

US01 As a user, I want to add a project so that I can keep track of it easily.



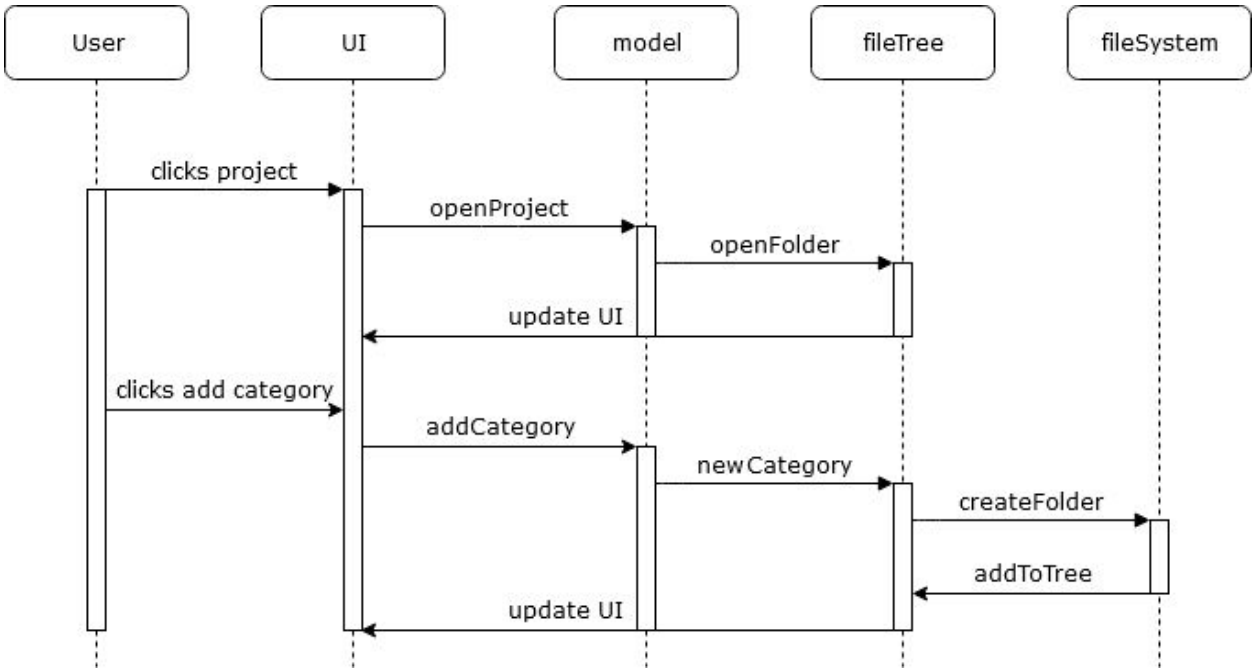
User story starts with the user clicking the “Add Project” button. This causes the UI to call the model with addProject. The model then calls Filetree with newProject to generate a new File. This new File is then added to the tree (our method of data representation). After adding to the tree the UI is updated for the user.

US02 As a user, I want to add an item to a project so that I can associate the item with the project.



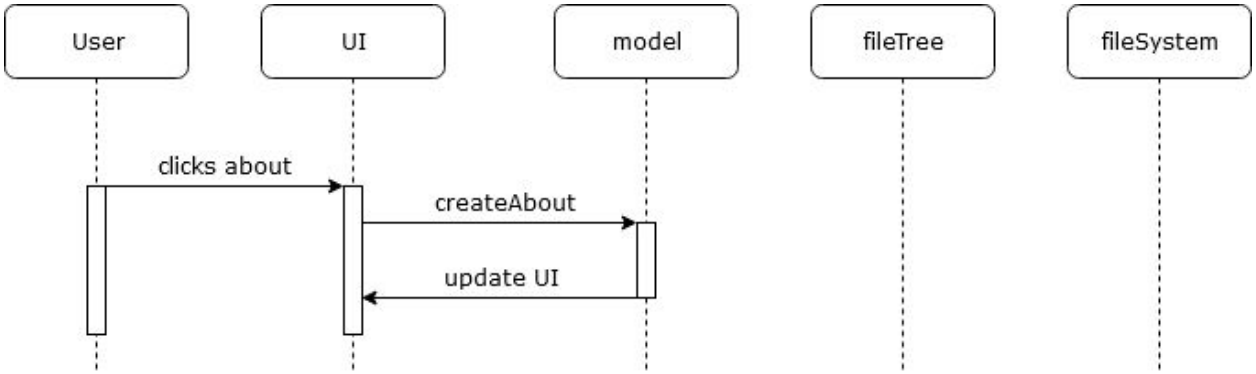
User story starts with the user clicking the Project they want to add an Item to. This causes the UI to call the model which calls the Filetree to open the Project and then update the UI. At this point the user clicks the “Add Item” button. The operation is similar to adding a Project at this point, only an Item is added to a Category while a Project is added to a Tab.

US03 As a user, I want to add subcategories to a project so that I can organize its items.



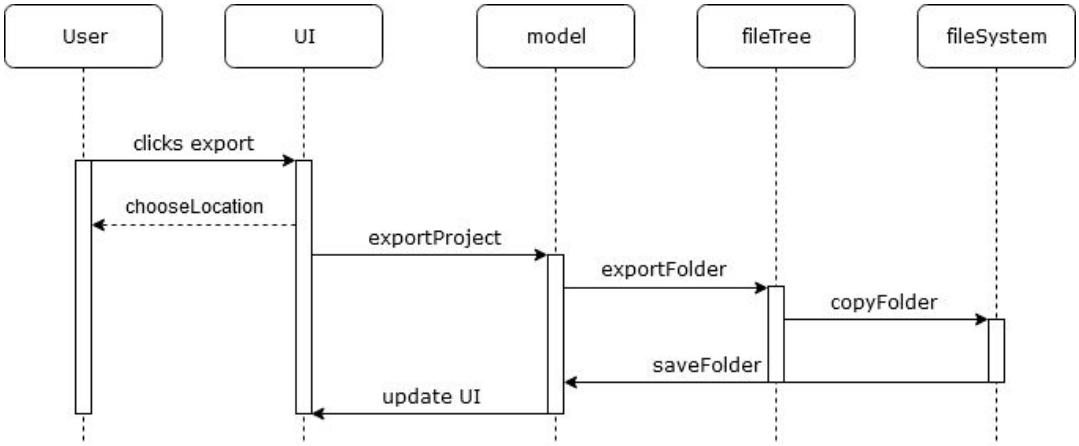
User story starts with the user clicking the Project they want to add a Category to. This causes the UI to call the model which calls the Filetree to open the Project and then update the UI. At this point the user clicks the “Add Category” button. The operation is similar to adding an Item at this point, only a Category is added to a Project while a Project is added to a Tab.

US04 As a user, I want to view the version number and developer names.



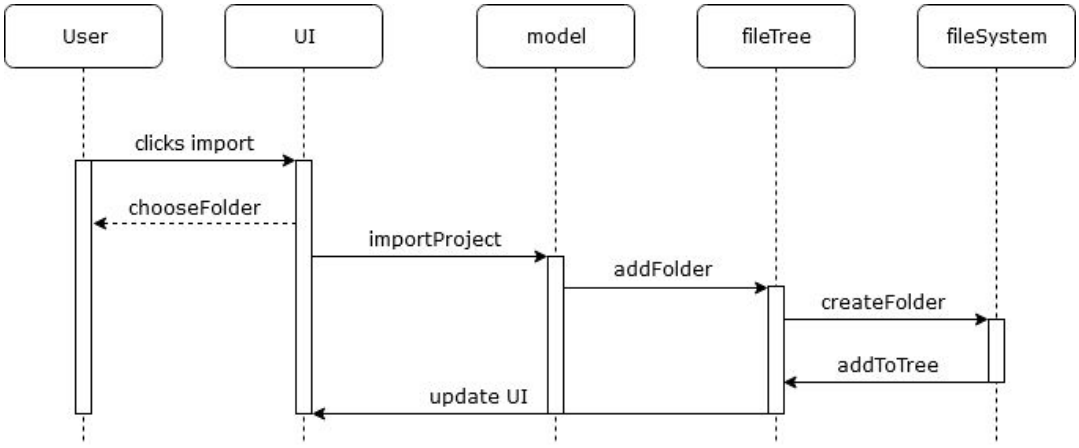
User story starts with the user clicking the “About” button. This calls the model to update the UI with a new pane containing the version number and developer names.

US05 As a user, I want to export projects for use on other devices.



User story starts with user wanting to export a project. User clicks on the “export” button on the home screen. A window is opened from chooseLocation, where the user is allowed to navigate their file system to choose a project to export. Once chosen, this causes the UI to call the model with exportProject. The model calls the fileTree with exportFolder. The fileTree calls the fileSystem with copyFolder. The project is added to the fileTree and the UI is updated.

US06 As a user, I want to import projects for use on the device.



User story starts with user wanting to import a project. User clicks on the “import” button on the home screen. A window is opened from chooseFolder, where the user is allowed to navigate their file system to choose a project to import. Once chosen, this causes the UI to call the model with importProject. The model calls the fileTree with addFolder. The fileTree calls the fileSystem with createFolder. The project is added to the fileTree and the UI is updated.

System Startup Sequence Diagram

Our system sequence starts with the user launching the program. Main calls build() from the Filetree class. This constructs the internal representation of the subset of the file system which the program will operate in, then calls the GUI. The GUI then constructs and displays to the user the splash screen for the program, which will consist of the tab pane. The UI elements from this point will be dynamically constructed based on the internal representation of the file system. The user is then free to add projects, search through the file tree, click on common tags, import files, or export their files all while having the changes they make reflected in the UI.