Nick Erokhin
11/28/2018

ImageSearch

A simple CLI app to manage and search through a photo library.

ImageSearch is a search engine for searching through image tags. Once a user adds a photo, the system fetches the image tags from the Google Cloud Vision API and stores the tags in a document matrix. The user is then able to use specific keywords to search through all indexed images to find the most relevant images to their search. Once a search is made, the user is even able to preview the images received in their most recent query. The shell interface is powerful, enabling the user to stack multiple commands at once such as adding images, saving the index to disk, running a search, and displaying all search results. All within one query

Requirements

Key:

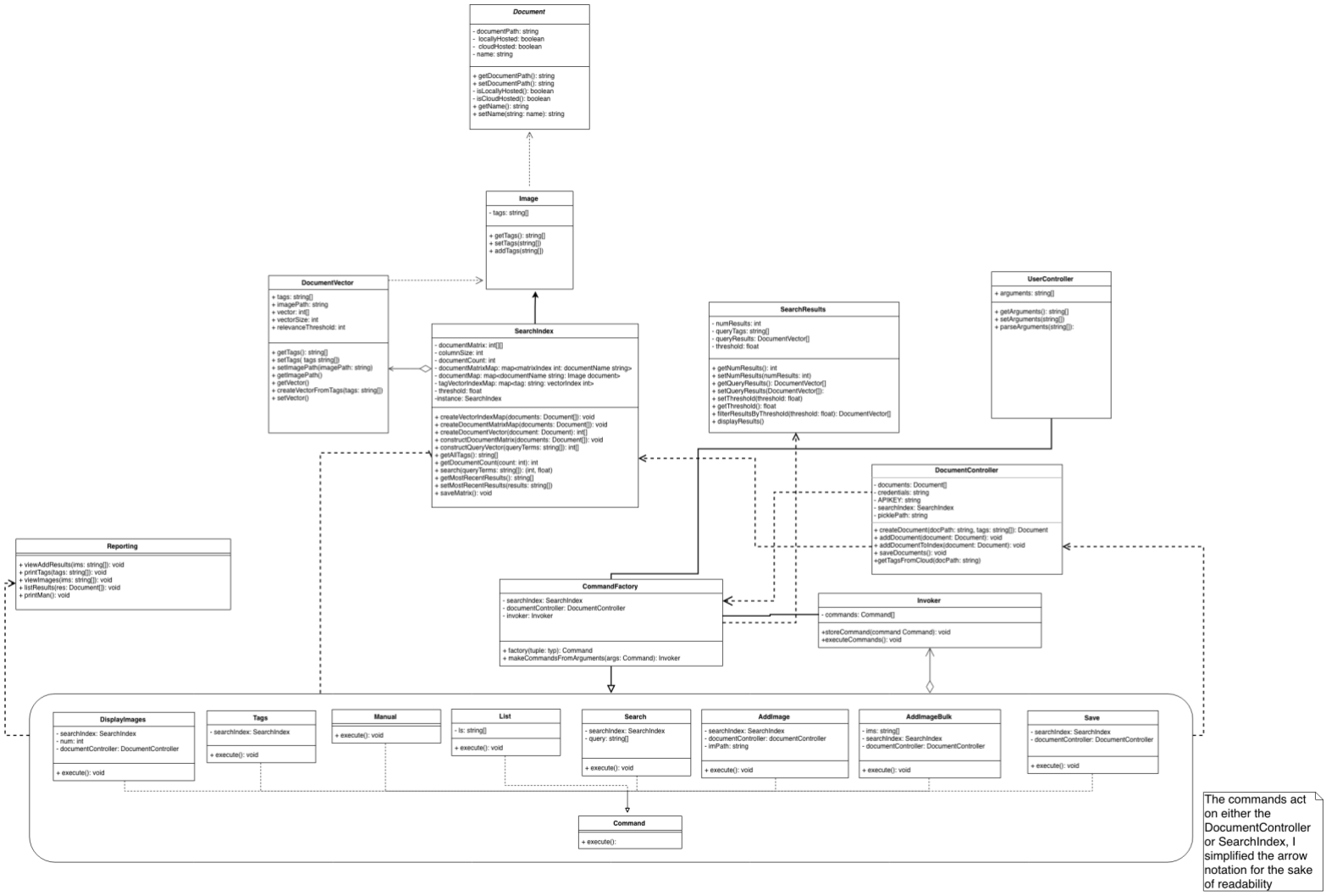
Complete
Partially Implemented
Incomplete

ID	Requirement Text
1.1	The system shall accept user uploads
1.2	The System shall allow users to download one or more images
1.3	The system shall provide the ability for a user to add and remove tags from an image
1.4	The system shall provide the ability for the user to list images in the system
1.5	The system shall provide the ability for the user to search for image tags and image text
1.6	The system shall provide the ability to the user to view the image without downloading to disk
1.7	The system shall allow the user to provide tags when specifying the image to upload
1.8	The system shall allow the user to delete images
1.9	The system shall allow the ability for the user to index every image
1.10	The System shall allow the ability for the user to batch-upload image files with a csv file

ID	Requirement Text

2.1	The system shall index every new upload automatically unless specified otherwise
2.2	The system shall provide an index on image text and image labels
2.3	The system shall persist the image index on disk
2.4	The system shall load the image index on program launch
2.5	The system shall rank the results based on query relevance
2.6	The system shall return search results with the image name and image tags side-by-side
2.7	The system shall return results in a timely manner

Final Class Diagram



The class diagram changed significantly since the 2nd progress report. The implementation of the design patterns caused the biggest change, followed by realizing inconsistencies and redundancies in design upon implementation.

This program is designed so that the Commands (Bottom of the diagram in the box) are the only objects interacting with every other object beyond the UserController. These commands are generated by the CommandFactory from the input received by the UserController. An argument string is parsed into n different commands, each of which is made into an object. The command pattern was especially useful here because the commands were executed the same way regardless of which classes they were acting upon.

Additionally, factory allowed for the creation of vastly different objects through one class. Some commands acted on the SearchIndex and depended on data from the DocumentController, or maybe only relied on SearchIndex; either way, factory made managing this significantly easier.

Additional changes were made when redundancies were found during implementation. For example, both DocumentController and SearchIndex had an "addDocument" method. This didn't make for the SearchIndex because it serves as a model, so I altered the design such that the SearchIndex only contained the necessary data and relationships for search, while DocumentController handles the documents. I have also made many of the previously public instance variables, private.

Implementing an MVC also forced a change in the class diagram. A more robust View was needed in order to provide the user with feedback other than search results, so the Reporting class was made. Additionally, separating the classes in their respective model, view, and controller compartments altered the design such that each class had methods consistent with what it was responsible for.

Design Patterns

Command:

Command was the perfect design pattern for handling CLI argument commands. My system was designed to allow the user to use multiple commands at once, for example

add 2 pic1.png pic2.png save search q/ term1, term2, term3 /q

“add 2 pic1.png pic2.png” adds the two specified images to the index. Adding to the index is completed as follows

1. Obtaining tags for each image
2. Creating Document objects for each image
3. Adding them to the DocumentController
4. Recomputing the index
5. Adding results to the view

“save” simply saves the index to disk. This is done by dumping the object into a pickle file

“search q/ term1, term2, term3 /q” runs a query with the specified terms. Performing a search is completed as follows

1. Creating the query vector using the mappings in the SearchIndex
2. Comparing the query vector with document vectors
3. Mapping the results back to the file names
4. Passing the results to the view

Each of these commands is vastly different and uses the classes in the system in completely different ways. Using command has completely abstracted this complexity from the UserController, where each command has an execute() method which handles the logic appropriately for each unique command.

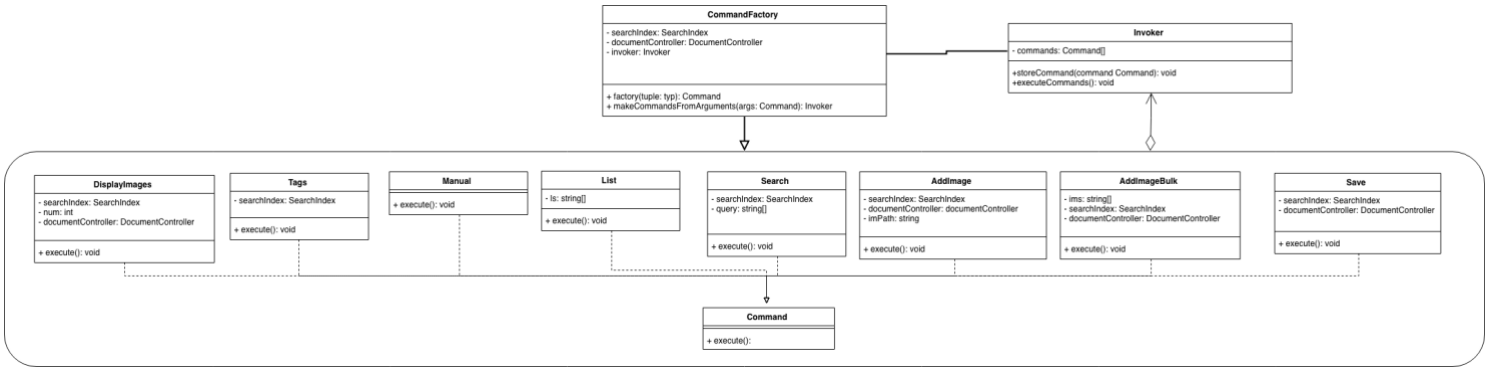
Factory:

After considering command, the system now had many separate command objects to be made. Each one requiring specific objects for its respective initialization. As more commands are added, the system will only grow in complexity. Factory is the perfect solution to manage this.

CommandFactory is initialized with all dependencies for the commands, both SearchIndex and DocumentController. Some commands may need one, both, or neither. However, the logic in factory() handles this very well, creating a simple command and providing each command with exactly the objects required.

Factory completely abstracted the complexity of managing vastly different commands from the UserController

The following diagram shows the combination of command and factory



The CommandFactory and Invoker are both controller by a client. CommandFactory is used to both generate command objects and populate the Invoker. The client then runs executeCommands() on the Invoker.

Design

Halfway through the project, I noticed that I wasn't getting stuck as often as I would with my normal spaghetti code practices. Prior to this, I'd often get stuck in a mess of my own code. What I had thought was a complex system ended up being very manageable once I completed my designs. Instead of spending hours debugging a confusing system, ImageSearch was built one step at a time, with easy debugging.

Having a structured diagram has helped tremendously. In particular, when one knows what each class/module is responsible for, the source of a bug is apparent much quicker than it would be otherwise.

Design patterns exist for a reason, they provide a robust framework for programmers to solve problems that makes sense to everyone involved. Not only do design patterns have the benefit of being well-known among programmers, but they also make working on large-scale projects substantially easier. Design patterns are the difference between learning a codebase over a span of months versus weeks.

The analysis and design of a system is just as important if not more important than the implementation itself. In a large scale project, a higher priority on planning will undoubtedly save countless developer hours.