GRIME README

Minor Changes

- Methods Relocation
  1. We relocate the `transform` method from `OperationUtil.java` to `AColorTransformOperation.java`
  2. We relocate the `filtering` method from `` `OperationUtil.java `` to `AFilterOperation.java`
  - Justification: From the feedback of the previous assignment, one TA told us these two methods should not live together, especially considering we have multiple levels of abstractions, we relocate these methods to those abstract classes and limit their accessibility, so only those operations under the same classification can use the method in their super classes.
- According to the assignment instruction, we change the arguments expected by the program, currently the program can run with `-file [filePath]`, or `-text` or no arguments.

Model

Library

This program uses a (hash)map for its upper-level model, representing an image library that allows multiple images being loaded and processed during a single run. By taking the benefits of the map structure, all images stored can be identified and can be retrieved by a key (in this case, image names user assigned).

The values of the map are objects (lower-level model) that directly or indirectly implement the `ImageFile` interface.

ImageFile

The current implementation of `ImageFile`, `ImageFileImpl`, uses a 2-D array of Color to store the color information from an image file, filed `alpha` in `ImageFileImpl` represents whether the alpha channel is enabled for this object. We do this differentiation because we aim to set a restriction that an image with alpha channel disabled should not apply an alpha related operation (See § operation). `ImageFileImpl` will also record the possible maximum value for Color.

Operation

Currently, `IImageOperation` is an interface for operations that can be preformed on an `ImageFile`. ImageFile can access and apply these operations by calling its own `applyOperation` method, as explained above in § Major Changes. All modification is performed on a **deep copy** of the original `ImageFile`, so the original `ImageFile` and the information stored in it will **NOT** be mutated by all means. If users want to replace the value corresponding to the identifiable key, however, they can simply reassign the value of which the key they used to retrieve that image.

Operators

For some operations, we do expect accepting an operator when that operation class getting instantiated, so that we can find the correct function apply to the given image. The benefits of using such a key-value mapping instead of creating several classes is obvious, that it has higher integration level and avoids code duplication. For current implementation, `IColorTransOperator` and `IFilterOperator` are two examples of how this operator should look like.

- These two (and probably all such) interfaces are especially for enum objects that lists the name of the operators. Current implementation also requires operable matrices, along with an appropriate getter, being built/implemented during instantiation. Note, however, this is not a necessary step for building an operator, an **operator** can work without a matrix, but only if the **operation** can work without it.
  1. `IColorTransOperator`

     Every enum member should be the name of a particular operation to convert a Color to another, by some defined rules, which may have a modified RGB value.

     Each `IColorTransOperator` contains a matrix that records the color transform for a particular pixels. The matrix is defined in the enum creation and can be retrieved by calling `getMatrix()`.

     Currently, `GreyscaleOperation` is able to use enum members (operators) that `SimpleArithmeticGreyscaleOperator` and `SingleChannelGreyscaleOperator` declared. `TingingOperation` is able to use enum members that `TiltingOperator` declared.
  2. `IFilterOperator`

     Every enum member should be the name of a particular operation to filter an image, by performing convolution using the provided kernel.

     Each `IFilterOperator` contains an odd matrix that represents a kernel.

     The kernel is defined in the enum instantiation and can be retrieved by calling `getMatrix()`.

     Currently, `FilterOperation` is able to use enum members that `IFilterOperator` declared.

R/W Capability

Both `ImageLibModel` and `ImageFile` are designed in a way that can limit (specifically in view package) the capability of performing malicious/mistaken mutation or overwritten. Specifically, we make these two interfaces implement the read-only interfaces `ImageLibModelState` and `ReadOnlyImageFile` that only have methods for inspecting information about models. On the other hand, given the ability to retrieve the information/status of a library/image, the model doesn't tell any details about the concrete implementation as all methods return abstraction and can only use public methods.

Extensibility

- Adding new operation is easy given the extensibility of the model.
    1. Create a new class that implements `IImageOperation`, or extends an existing abstract one if this operation land under that particular classification (existing abstract posts several restrictions for its subclasses, and we assume the future implementors will abide by those restrictions), overwrite the required methods based on the mechanism and properties(e.g. is it alpha related) of the operation.
        - Some operation probably needs an `operator` and its corresponding `function`, if so, implementor may need to create new enum class to specify the operators and implement the function / call the existing one when building the operation class.

2. Now the new operation can be applied to a `ImageFile` but we still need to let controller know **when** (but not how) this operation should be evoked (detailed described in § Controller : Extensibility).

View

Our GUI view is consist of three function sectors - info panel, preview panel, and control panel. Currently, the info panel shows the histogram graph of the image that is currently worked on, and the preview panel present this image. Histogram panel is fixed in size, while the preview panel is (indirectly) resizable as it can be resized by adjusting the size of the frame window. If the image is too large for preview panel to present, the scroll bar will show up and the user can still see the whole image by scrolling the mouse wheel. The controller panel is made up by two area: selection area and button area. When the program start, the library is empty. No other except `laod` can be performed, so other buttons are grey out, they will be enabled when an image is imported. Whenever an image is loaded or created by an image processing operation, the selection list will auto select that newly imported/created image, and the two left panel will refresh according to the information (pixels) of the selecting image. The refreshment will be performed also if only another existing item in the list is selected. Buttons are linked to all commands supported by the controller, pop-up window will show up when trying to perform an operation, user need to input (name/brightness value/save directory) however they want as long as those inputs are meaningful. Bad input will be thrown away and the view will pop up alert window to tell what's wrong with the input, letting them try again. We try our best to design it user-friendly, such as assigning the default input so almost all operation can be performed simply by pressing "enter" without specifying (predictable) information everytime.

Controller

The Controller is in charge of commands user inputs in and interacts with the view so that the view can correctly render feedback messages.

ControllerImpl

`ControllerImpl` stores all supported command/operations in its cmdMap, which be expanded by extending this class for supporting future operations. The `run` method pass the command sequence to an appropriate `ICommand` object and delegate tasks to it.

Command

`ICommand` defines an interface for future commands to implement, this follows the command line design pattern and aims to make the program more extensible. The current `ACommand` abstract common part of parsing command line arguments and only delegate the essential part of a command operation to its subclasses. This aims to reduce code duplication

I/O

We designed and implemented `IWriter` to manage output format, by the formatting rule required by a particular file format; same strategy was used for `ILoader`, which is used to analyze a given image file and convert it into a 2-D Color Array. Considered the large amount of extension/file suffix used over different platforms/machines, we designed `SuffixManager` in order to supply appropriate `ILoader` / `IWriter` based on the provided file extension (as a String). The goal of this design is trying to give the program as much extensibility as possible, when it's dealing file format.

GUIController

The GUI controller `ImageProcessControllerGUIImpl` implements a new interface `ImageProcessControllerGUI` that enables itself to receive a stream of arguments (mostly from the `ActionListener` and `ListSelectionListener`), refill its queue that stores command/arguments, and perform(/run) the operation. This controller still extends `ImageProcessControllerImplV2` to inherit the latest command set. The `run()` method is overridden to adapt the new interactive interface.

Two private inner classes are `ImageProcessActionListener implements ActionListener` and `ImageLibrarySelectionListener implements ListSelectionListener`. Two classes are instantiated in `ImageProcessControllerGUIImpl`'s constructor at the point of initializing the linked `IGUIView`, and are dispatched to hear events from the view panel.

Extensibility

- Any additional command can be added by
    1. Implementing the `ICommand` interface (or extending the current `ACommand` abstract class), specifying how `execute` should work, and
        - either execute or the helper for execute need to call the corresponding `IImageOperation` to perform the operation.
    2. Putting the new command into the supported map by extending the current `ImageProcessControllerImpl` class.
- The controller is also designed to be open for more file types. To support a new image type, simply
    1. Implements `ILoader` (how to load an image, i.e., how to convert a particular image file to a 2-D color array)
    2. Implements `IWriter` (how to save an image, i.e., how to convert back given a 2-D Color array to the target file format).
    3. Add the two implementations to `availableSuffix` map in the `ASuffixManager<T>` by directly or indirectly extends it. The `provide` method will supply the controller with a suitable rule to convert back and forth from the human-readable file format to machine-processable data type, and finally
    4. Update the value for `save` and `load` of `cmdMap` in `ImageProcessControllerImpl` with the new `WriteSuffixManager<IWriter>` and `LoadSuffixManager<ILoader>`

Resource Image Citation

Image "elephant" citation: Photo by https://unsplash.com/photos/qiPTr8GmhM0

Image "painting" citation: Photo by https://unsplash.com/photos/JLfem8ViKVA

Image in `sample.png` "Ama Dablam mountain" citation: Photo by https://unsplash.com/photos/9wg5jCEPBsw