2017

Object Oriented

Assignment 2

ADAM HARRIS & DUNCAN BALDWIN
14008368 & 14002425

Contents

Client brief

3.1. Client Brief You are required to design and produce a GUI-based image manipulation program that uses the FreeImagePlus and Fltk libraries, according to the following client brief.

The program should:

- Contain (in memory) a collection of images, that is initially empty.
- Allow the user to load one or more images (with the use of a wildcard) into the container.
- Display one of the images in the container, scaled to fit the display, and initially showing the first image in the container – for any image that is displayed in the GUI, we shall refer to the corresponding image in the container as the 'current image'.
- Allow the user to select the current image, via a 'forwards/backwards' button interface, as used for the first milestone.
- Allow the user to rotate, flip, and scale the current image.
- Allow the user to save the current image to a file, with the user-supplied path/filename.

Although not essential, you may wish to extend the program further so that it:

- Displays thumbnails of all the images in the container (ideally in a separate 'View').
- Allows the user to create new images that are composed of other images in the container.

Problem analysis

Based on the above requirements the client has asked for an image manipulation tool using the two libraries FreeImagePlus and FLTK. The software should allow the user to contain a collection of images, with the ability to load more than one image at any given time and not break if the user has loaded a non-image file type into the software. The software should display the first image loaded into the program with the ability to move forwards/backwards between the loaded images. In terms of image manipulation, the user should be able to rotate, flip, scale and save the current image with a user specified path and filename. Additional functionality of thumbnails and create new images by composing two different images together, although these features are not essential.

Software design

S.O.L.I.D

Single responsibility

- An object or a class should have only one responsibility.

To create the graphical user interface we had an IView class that held all of the FLTK buttons/inputs the user could perform operations with. This meant that the view had two responsibilities, be a graphical user interface and perform as the view in the MVC architecture, this also meant that it was confined to be a GUI based off the current visual design. To negate this problem, we created an ImageGUI class and imported all of the FLTK functionality into it using the fluid program. This way we have a GUI class in charge of the FLTK functions and a view class in charge of updating that data on the GUI.

In theory, this also means that the view class could be turned into a template class and become which ever GUI type is used to create it. For instance, if the software is an image manipulation tool then the view would use an ImageGUI, likewise meaning that if the user created a SoundGUI then we could create another view that uses a SoundGUI. This would make the MVC design more generic and interchangeable.

Open/ closed

- Software should be open for extension but closed for modification.

The IFactory interface demonstrates the open/closed principle. The class is defined as a template. The template allows the class to be an instantiated with a key type which then set the value of the function in the interface. This means the class could be used to create an ImageFactory but also for a SoundFactory.

Liskov's substitution

- Objects should be able to replace with instances of their subtypes and still behave as expected.

EventArgs uses the IMedia interface for the data storage. Model and View only need to know about the EventArgs class but the specific GUI, in this case ImageGUI, can choose to cast the data to the relevant interface, in this case IImage in order to access the fipImage functionality.

Interface segregation

- Programming to many client specific interfaces rather than one general purpose interface

Separate IClick and IController, not all controller functionality could be for clicking, if we wanted to use key presses, create an IKeyPress and have the controller inherit from that interface. Then perform the same pattern as the IClick interface.

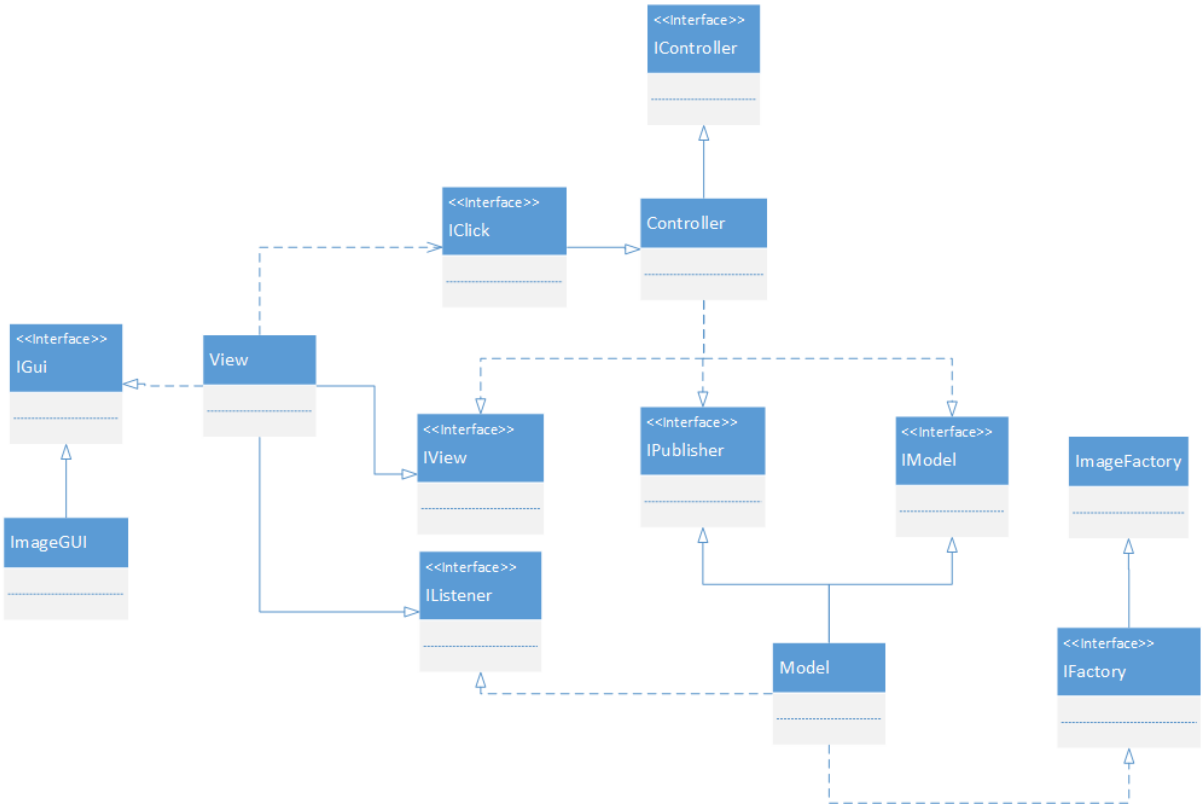
Dependency inversion

- Should depend on abstractions not concretions.

Use pure virtual functions, in C++ this makes the Interface an abstract class as well so it is required to be implemented for the program to build. It means that all concrete methods must override the abstract methods in the interfaces.

Diagrams

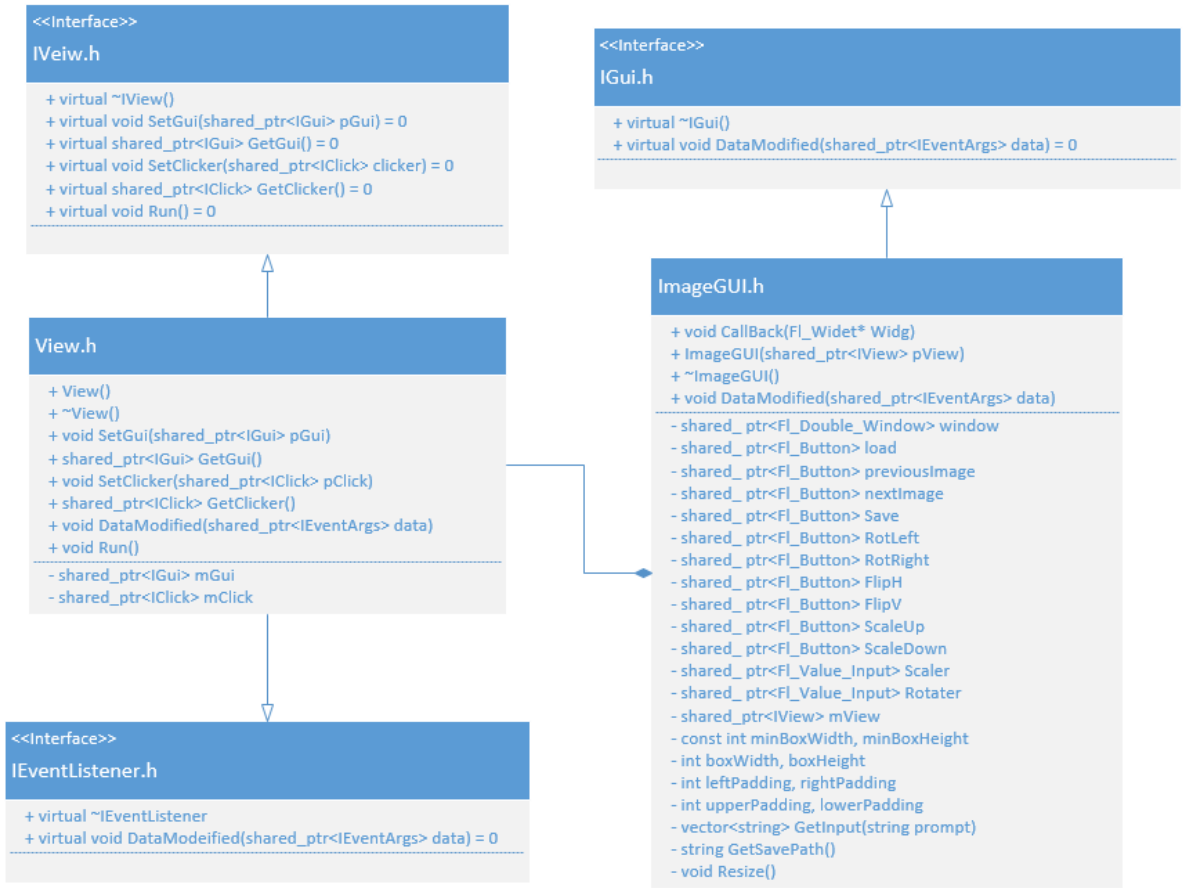
UML Communication: Basic software structure



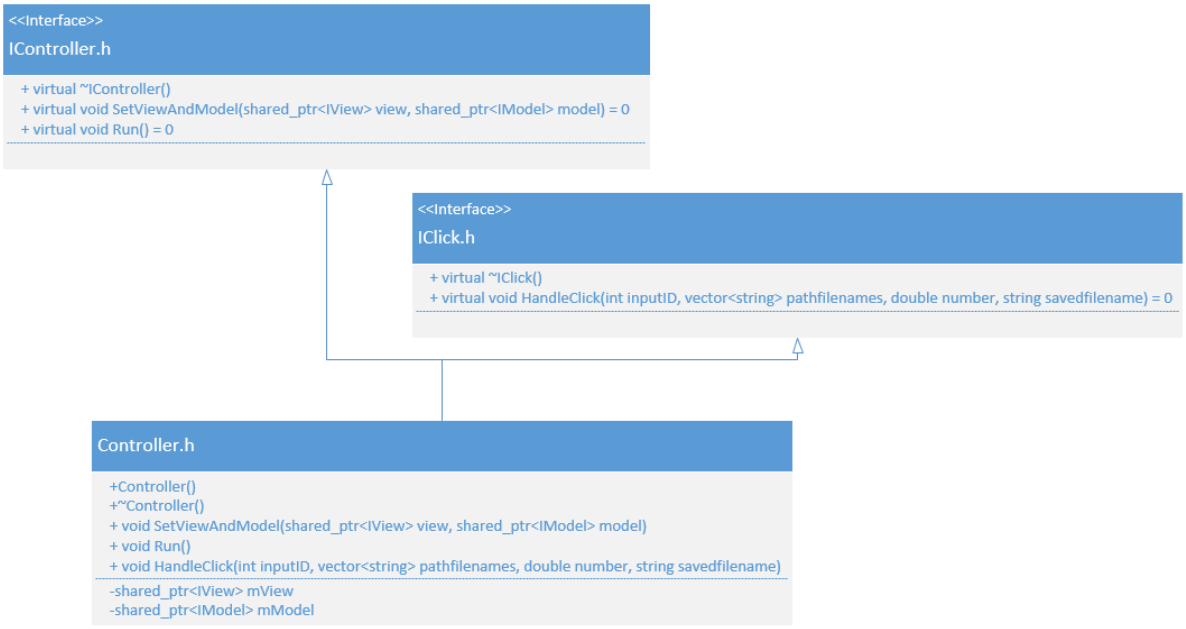
This diagram demonstrates how all of the classes are interconnected.

UML Class: Software Functionality

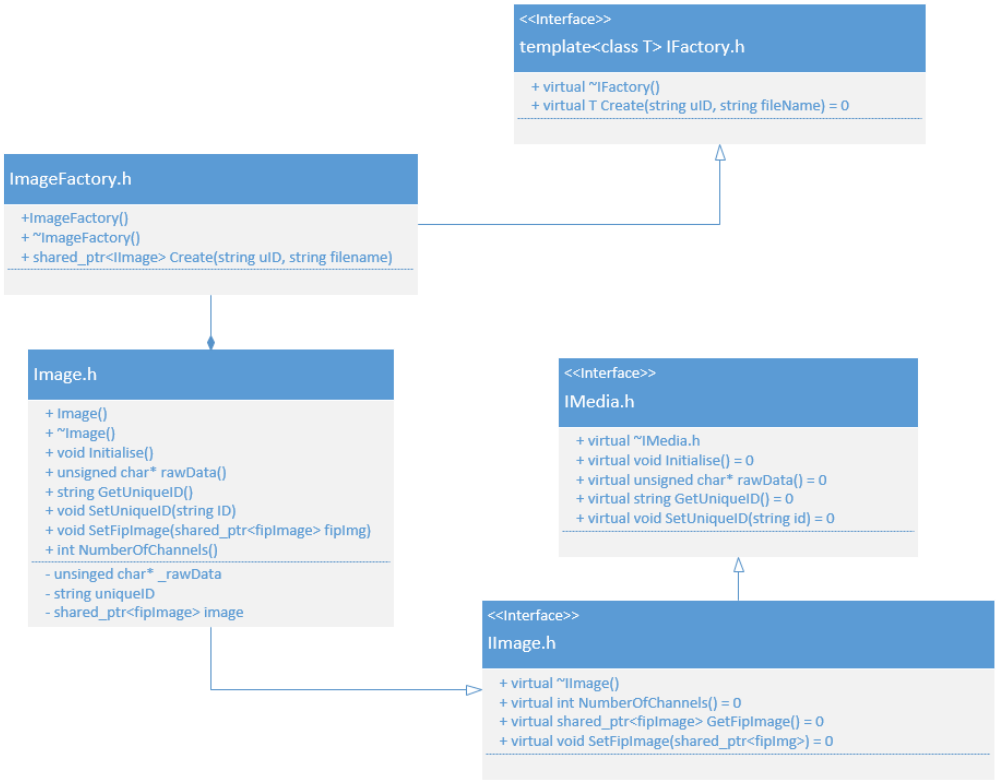
UML Class: View and ImageGUI



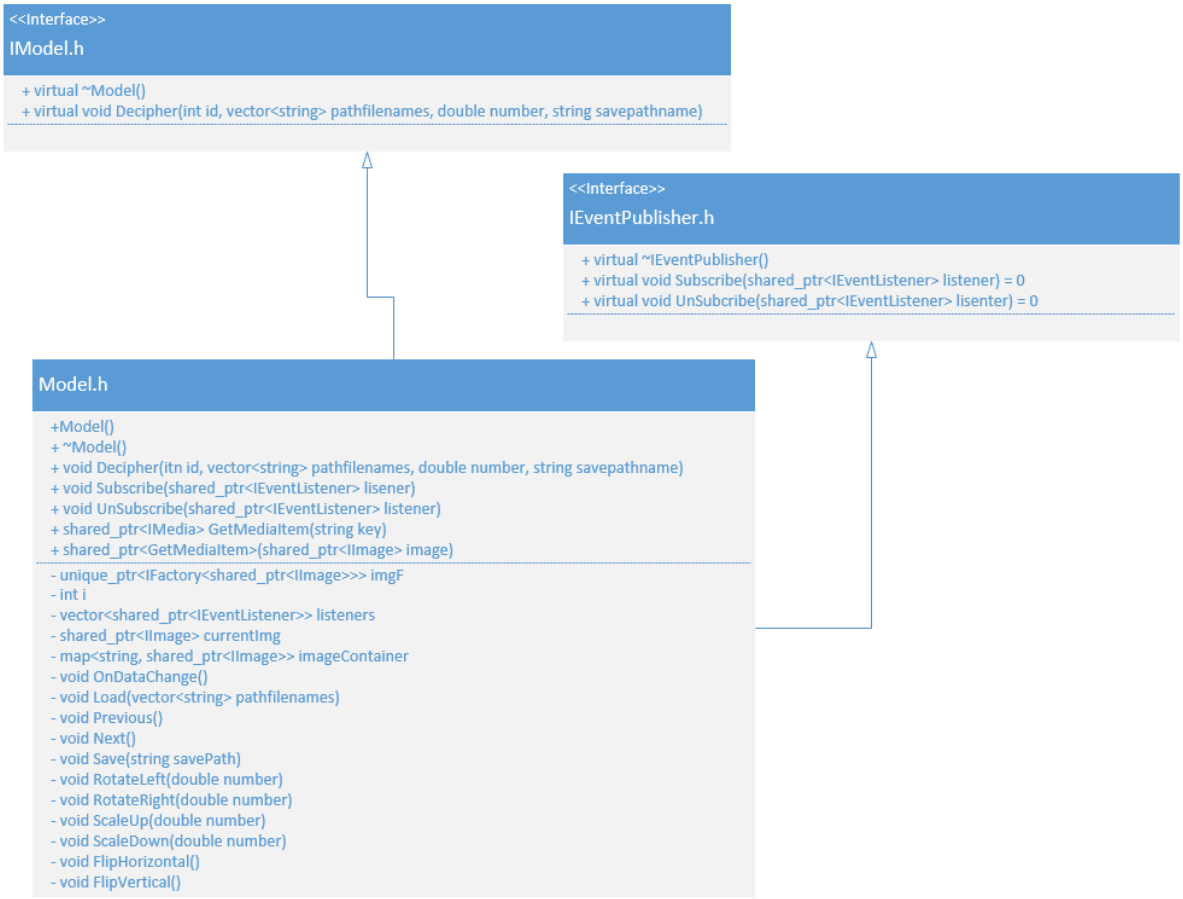
UML Class: Controller



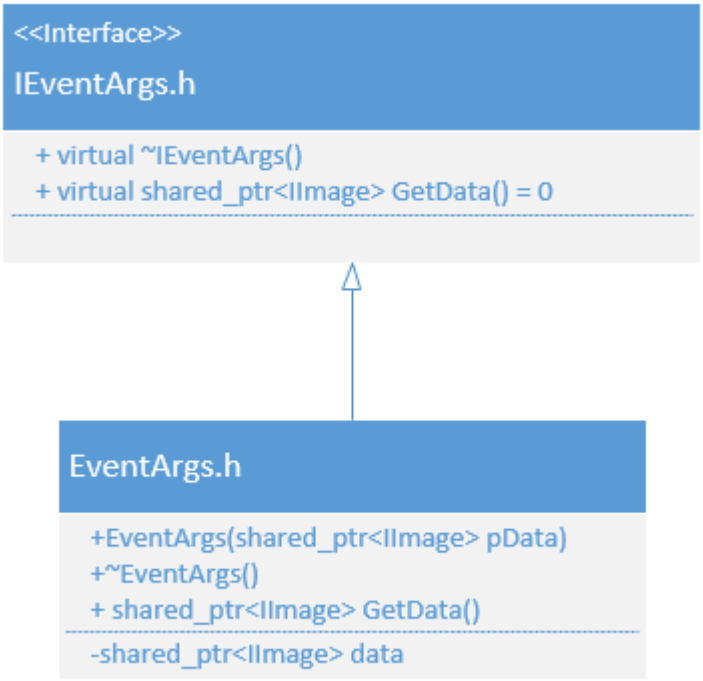
UML Class: ImageFactory and Image



UML Class: Model

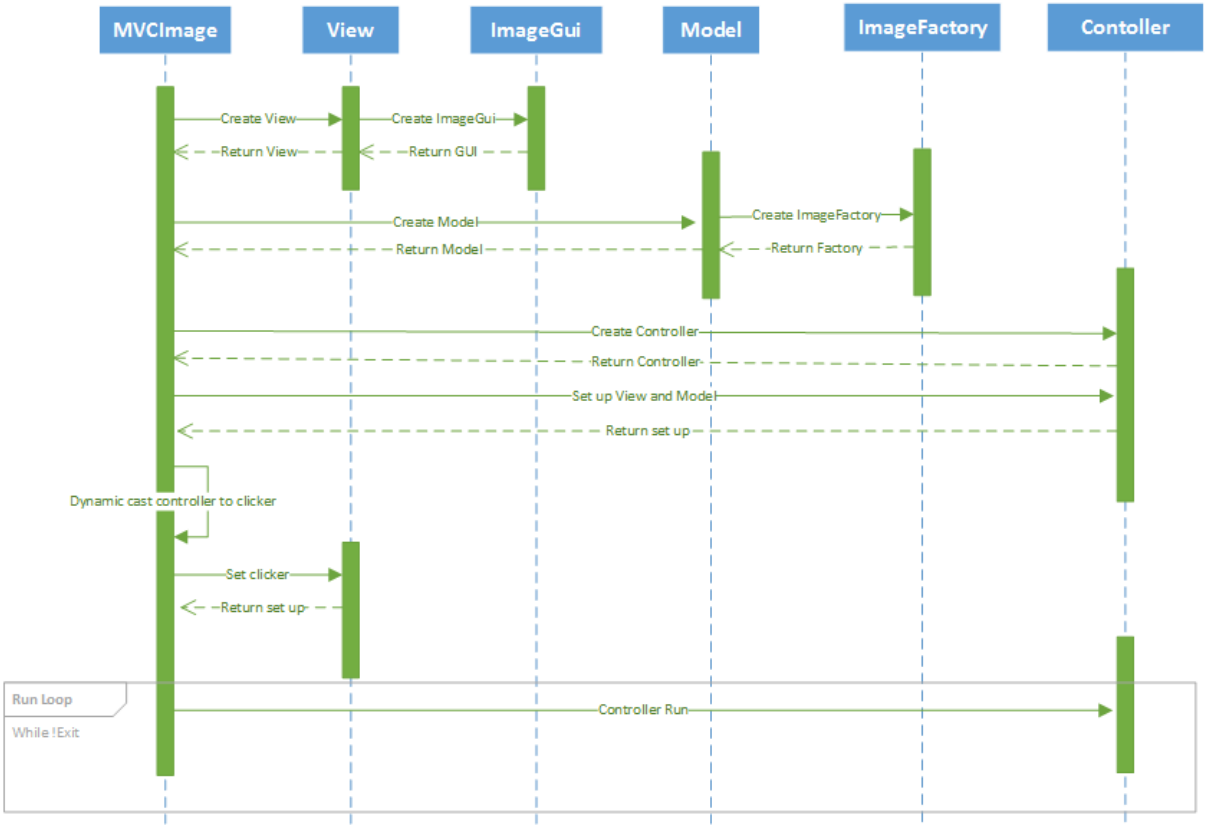


UML Class: EventArgs



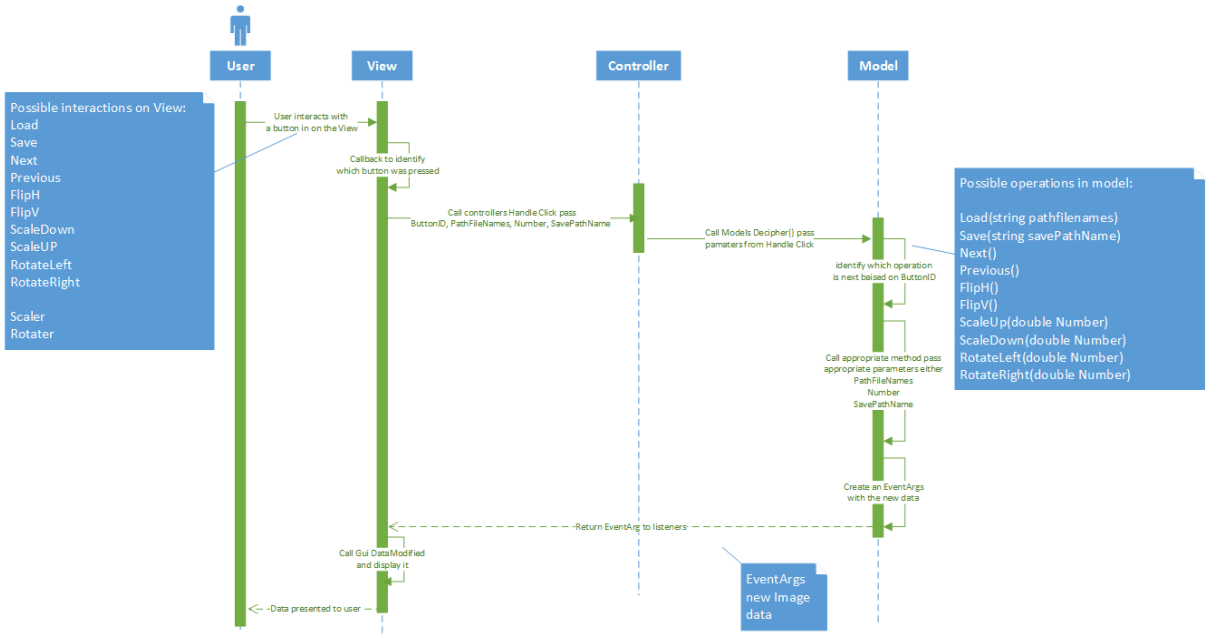
These UML diagrams demonstrate the underlining functionality in the software.

UML Sequence: Software setup / run



This sequence diagram demonstrates how the software setups the MVC architecture in order to get to the run loop of the software.

UML Sequence: Software GUI User Interaction



This diagram demonstrates how the user input is handled through the software. Currently the graphical user interface will allow 10 different operations and 2 different adjusters.

Demonstrates requirements

The operations are:

- **Load** – used to load images into the software the user can load as many different images and image types as they wish. Images will be scaled down to fit the screen if they are too large.
- **Save** – allows the user to save the current image to somewhere on that machine, keeping all adjustments made to the image before saving. The current image can only be saved as a PNG (Portable Network Graphic) file.
- **Next** – displays the next image in the list of loaded images, will go back to beginning if they are on the last image.
- **Previous** – displays the previous image in the list of images, will go to end if the user is on the first image.
- **FlipH** – Flips the image on the Y axis.
- **FlipV** – flips the image on the X axis.
- **ScaleDown** – scales down the image biased on the input of the scalar adjuster.
- **ScaleUp** – scales up the image biased on the input of the scaler adjuster.
- **RotateLeft** – rotates the image anticlockwise biased on the rotator adjuster.

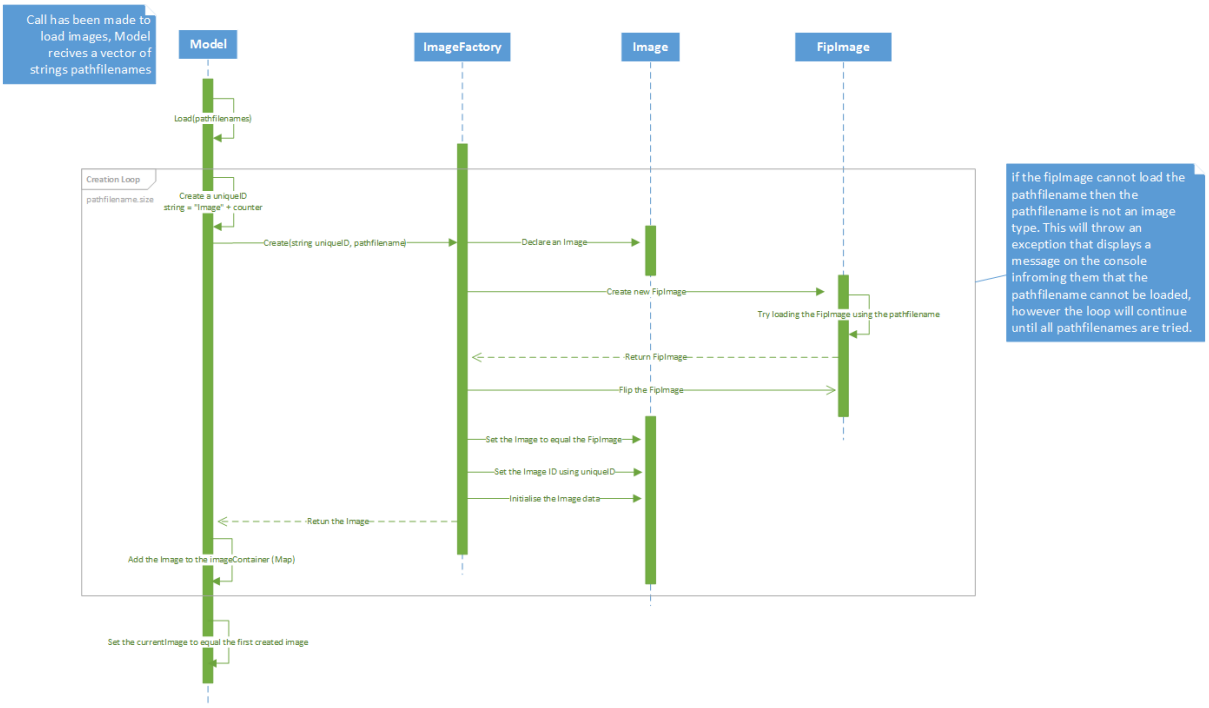
- **RotateRight** – rotates the image clockwise based on the rotator adjuster.

The adjusters are:

- **Scaler** – allows the user to define the amount an image should be scaled, currently the software is scaled through percentages. So if the user defines 50 the image will be scaled by 50.

- **Rotator** – allows the user to define the amount an image should be rotated.

UML Sequence: Software Image loading/Creation



This diagram demonstrates how images are actually created during the load loop if the model class. If a non-image type is loaded into the software, then the software will handle this exception and inform the user that this operation could be performed for that loaded particular file and continues with the loading loop.

Design patterns

MVC Architecture

MVC is a software architecture which is used for manipulating graphical user interfaces. It stands for Model-View-Controller and these three types are interconnected and can easily pass data between them whilst maintaining encapsulation.

In our design, we use the Observer Pattern and make model a Publisher that would update listeners on data changes, the view implements the Listener functions to receive the data passed by the model in order to update the GUI.

We also use the strategy pattern to work as a go-between for the view and the model as when information is updated using the view it can then be passed to the model through the controller strategy.

Model

- The model is the container for all of the data in the software and passes adjusted data to the view.

View

- The display the relevant data from the model.

Controller

- The controller handles interaction between the view and the model.

Strategy pattern

The strategy pattern allows an object to perform functionality without having to implement it. This is done by injecting an interface into a class that requires the functionality.

```
class IClick
{
public:
    virtual ~IClick() {};

    // Method: HandleClick
    // Definition: Handles a user click interaction
    // Param 1: int inputID
    // Param 2: map<IImage> imageCOntainer
    // Param 3: double transform
    // Param 4: string filePathToSave
    virtual void HandleClick(int inputID, vector<string> pathfilenames, double number, string saveFileName) = 0;
};
```

After defining an interface, we have called it IClick, the Controller from the MVC architecture implements this interface. We inject the Controller as an IClick into the View class, this allows us to send data obtained from the ImageGUI class to the Controller, through the IClick implementation of the HandleClick method. The strategy pattern is used to maintain encapsulation, which can increase cohesion and decrease coupling, because the view has no reference to the model, but data is still received in the model by interaction with the view.

```
mView->GetClicker()->HandleClick(button, imageFiles, number, saveFilePath);
```

Demonstrates how the view calls the injected controllers handle click method to get data in the model. The four parameters are the buttonID which in an integer representing which button is pressed. ImageFiles a vector of strings to load. Number which can either be the scaler or rotator adjuster based on which type of button is pressed. SaveFilePath which is a string representing the save directory and name. Although all four parameters are passed no matter which operation is

performed the model will only take the relevant parameters based on the decipher method in the model.

Observer pattern

The observer pattern allows a one to many relationships between objects. Therefore, if one of the objects has been modified all of the other objects depending on that change would be notified. This will be referred to as a listener/publisher relationship. Where the listener is listening to the desired objects data change, and the publisher attaches the listener to the object in question. Listeners can be added and removed at any moment suggesting that this pattern is very versatile.

The purpose of the observer pattern is to reduce the coupling between objects, and perform the desired operations in way that applies to the S.O.L.I.D principles more appropriately by the way objects are connected and the data passed between them.

The view implements the IEventListener interface as the view listens for data change in the model class. Whereas the model implemented the IEventPublisher interface as it subscribes the view by adding it to models vector of IEventListeners. This is all done in the controller to ensure that the view and model have as little coupling as possible. When data is changed in the model via the strategy pattern, the OnDataChange method is called which informs all listeners to the model that something has changed and passes that data to the listener through an EventArgs.

Controller subscribing the listener through the publisher

```
// Method: SetViewAndModel
// Definition: Set the view and model for the controller
// Param 1: shared_ptr<IView>
// Param 2: shared_ptr<IModel>
void Controller::SetViewAndModel(shared_ptr<IView> pView, shared_ptr<IModel> pModel)
{
    mView = pView;
    mModel = pModel;

    // Cast viewer to an IEventListener
    shared_ptr<IEventListener> listener = dynamic_pointer_cast<IEventListener>(mView);
    // Cast model to an IEventPublisher
    shared_ptr<IEventPublisher> publisher = dynamic_pointer_cast<IEventPublisher>(mModel);
    // Subscribe the listener to the publisher
    publisher->Subscribe(listener);
}
```

OnDataChange Model Method

```
// Method: OnDataChange
// Definition: Called when the key data has changed
void Model::OnDataChange()
{
    // Create an IEventArgs
    shared_ptr<IEventArgs> data = shared_ptr<IEventArgs>(new EventArgs(currentImg));

    // Create an iterator of IEventListeners
    vector<shared_ptr<IEventListener>>::iterator it;
    // Loop through the listeners
    for (it = listeners.begin(); it != listeners.end(); it++)
    {
        // Tell listener through dereferencing that data has changed
        (*it)->DataModified(data);
    }
}
```

Event delegates

The event delegates were used as a container for the data being transferred. This methodology ensures that the data being passed is encapsulated. In our case the EventArgs data is an IMedia item.

```
class EventArgs : public IEventArgs
{
public:
    // Constructor:
    // Definition: Construct this class
    // Param 1: shared_ptr<IMedia> pData; the image data to be passed to the events listener
    EventArgs(shared_ptr<IMedia> pData);
    ~EventArgs();

    // Method: GetData
    // Definition: Returns the image data which is stored upon using the constructor
    // Return: shared_ptr<IMedia>
    shared_ptr<IMedia> GetData();

private:
    //Declare a shared_ptr<IMedia>
    shared_ptr<IMedia> data;
};
```

Evidencing

Efficient design

Virtual destructors

All inherited classes have a virtual destructor. If the program had destructors that are not virtual then there is the possibility of creating a memory leak as objects would not be deleted properly. Virtual destructors ensure that when the parents destructor is called then the subclasses destructor is also called completely removing that object from memory.

Appropriate commenting

The software has been commented throughout. These comments are relevant to the imbedded code and are there to help outside developers understand what we are doing. Also descriptors are important to interface designed software and increase the usability of the program.

Unique ID

All objects are created with a unique ID this is used as unique identification to represent that object. This ID is a string, and lowers coupling as other classes only know the unique identifier of that object rather than what that actual object is. This also allows use to pass around string rather than objects.

Efficient logic

For loop iterators

Where need any loops were created through the for loop iterators. The newest version of C++ has these built in iterators and are faster than ordinary for loops.

Figure 2: Snippet of iterator

```
// Method: Load
// Definition: Load an image or multiple images
void Model::Load(vector<string> pathfilenames)
{
    cout << "Load... Model.cpp" << endl;

    // PROCEED only if imageFiles contains some strings:
    if (pathfilenames.size() > 0)
    {
        // Iterate using a iterator through the vector<string> pathfilenames
        for (vector<string>::iterator it = pathfilenames.begin(); it != pathfilenames.end(); it++)
        {
            // Create a unique identification to assign to the IImage object
            const string uid = "Image" + to_string(i);

            try
            {
                //cout << uid << " ImageModel.cpp Line 29" << endl;
                // Create a shared_ptr<IImage> and intialise to the imgF->Create(): pass the uid and pathfilenames current element
                shared_ptr<IImage> image = imgF->Create(uid, *it);

                //cout << uid << " ImageModel.cpp Line 31" << endl;
                // Insert the image field into the imageContainer
                imageContainer.insert(pair<string, shared_ptr<IImage>>(uid, image));
            }
            catch (bool b)
            {
                if (!b)
                {
                    cout << "File path must have extension of .png or .jpg - the following file is invalid" << endl;
                    cout << *it << endl << endl;
                    i--;
                }
            }

            // Increment i
            i++;
        }

        currentItem = 0;
        currentImg = imageContainer.at("Image" + to_string(currentItem));
    }
}
```

Smart pointers

There are two raw pointers in this software due to constraints with the FreeImagePlusD external library that is used to develop the software. The FreeImagePlusD library contains a fipImage class that is used to create the images in this software unfortunately this class also requires a raw pointer to access the images bits therefore cannot be avoided.

The second is a raw pointer is required for a FLTK call back function. Ideally these would be coded as smart pointer however the external libraries don't allow this.

Error free

No errors in the current solution.

Testing

Loading non-image types

To make the software as user friendly as possible there is a try catch statement around the load image functionality. This is designed and implemented to allow only images through the function and will not break if the user uploads a non-image type file into the software. The software should also carry on working as expected with a message delivered to user informing them of something loaded was not an image.

Figure 1: Exception handling

Figure 1 demonstrates that the software can handle non-image types being loaded into the program. This image shows an image, sound and image being chosen to load. The software loads the first image then handles the sound and replies with a console log then loads the last image. All the functionality of the software can still be used on the two images loaded.

GUI interaction without a loaded image

If the user attempts to use the GUI operations other than load without a loaded image, then software will throw an exception informing the user that they need to load in an image to perform that operation.

Cancelling the load image operation

Also if the user cancels the loading image operation the program will inform them the operation was cancelled rather than breaking the software's run loop.

Future Improvements

ImageModel/MediaManager Implementation

Currently the software contains a model class and this class is responsible for changing data based on user interaction. It currently does this by storing created image data in a map called imageContainer. This container could be created and help within another class like an ImageModel/mediaManager. This would make the model class more generic and interchangeable as it could get the relevant container class manipulate the data based on the interaction then return it to the container class and pass the new data to the view. In theory this would also mean the model class could become a template class and become the type based on the GUI implemented.

Template Classes

This has already been somewhat discussed throughout the documentation. However, the view and model classes could be templated.