University of Colorado at Boulder
CSCI 5448 – Object Oriented Design & Analysis

IPA – Image Processing Application

Timothy Mason (solo project)

Final System State

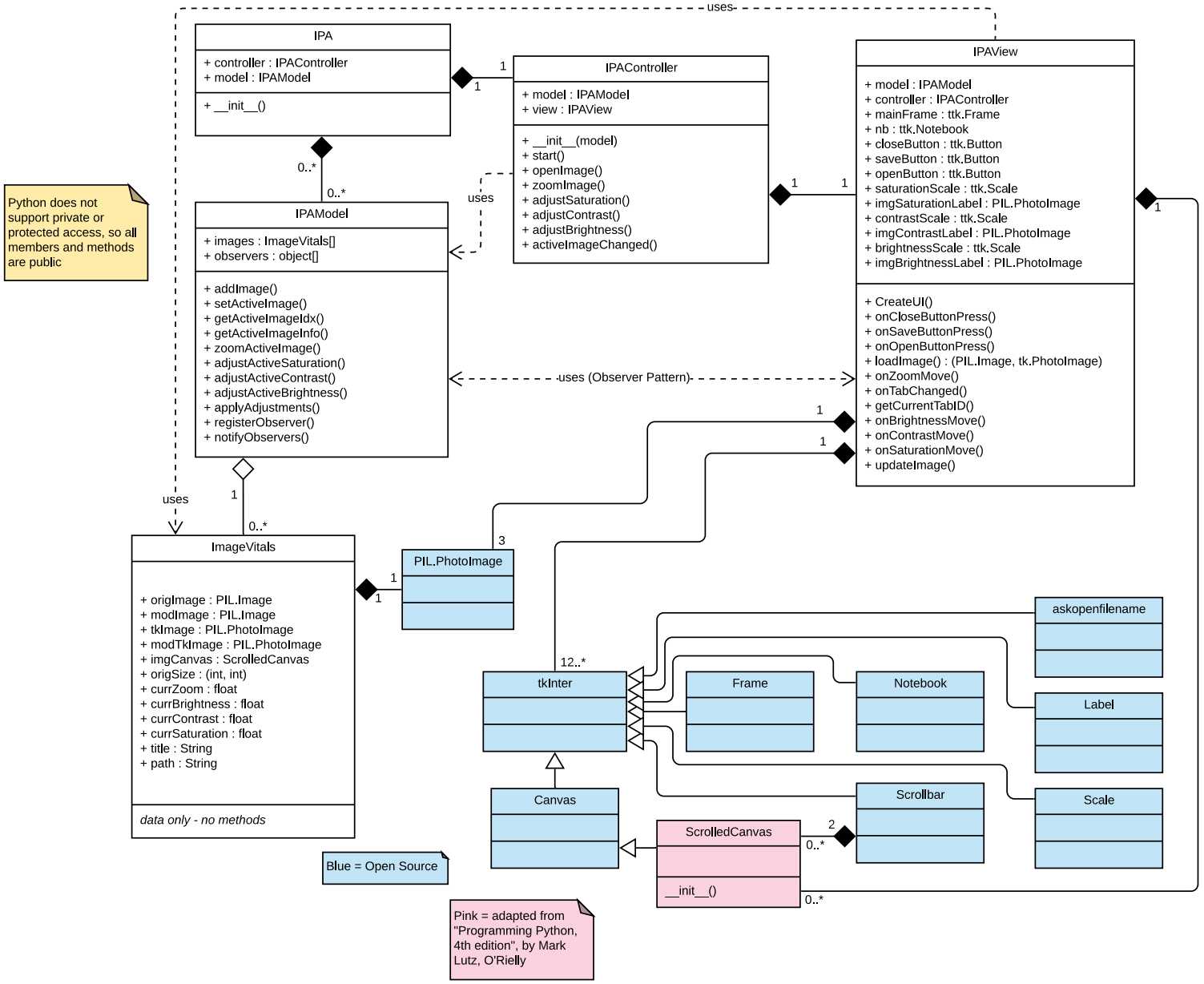
State of the System

I have succeeded in implementing the minimum image processing application as described in the initial proposal. All required features are present and implemented with a modular design that is easily expandable to allow additional features to be added. The application is written completely in Python 3 with platform-agnostic code, meaning the same application can run without modification on Mac, Windows, or Unix. Above are a couple of screenshots from **MacOS Mojave** and **Windows 10** (showing my daughter from 10 years ago and now).

List of Features Implemented

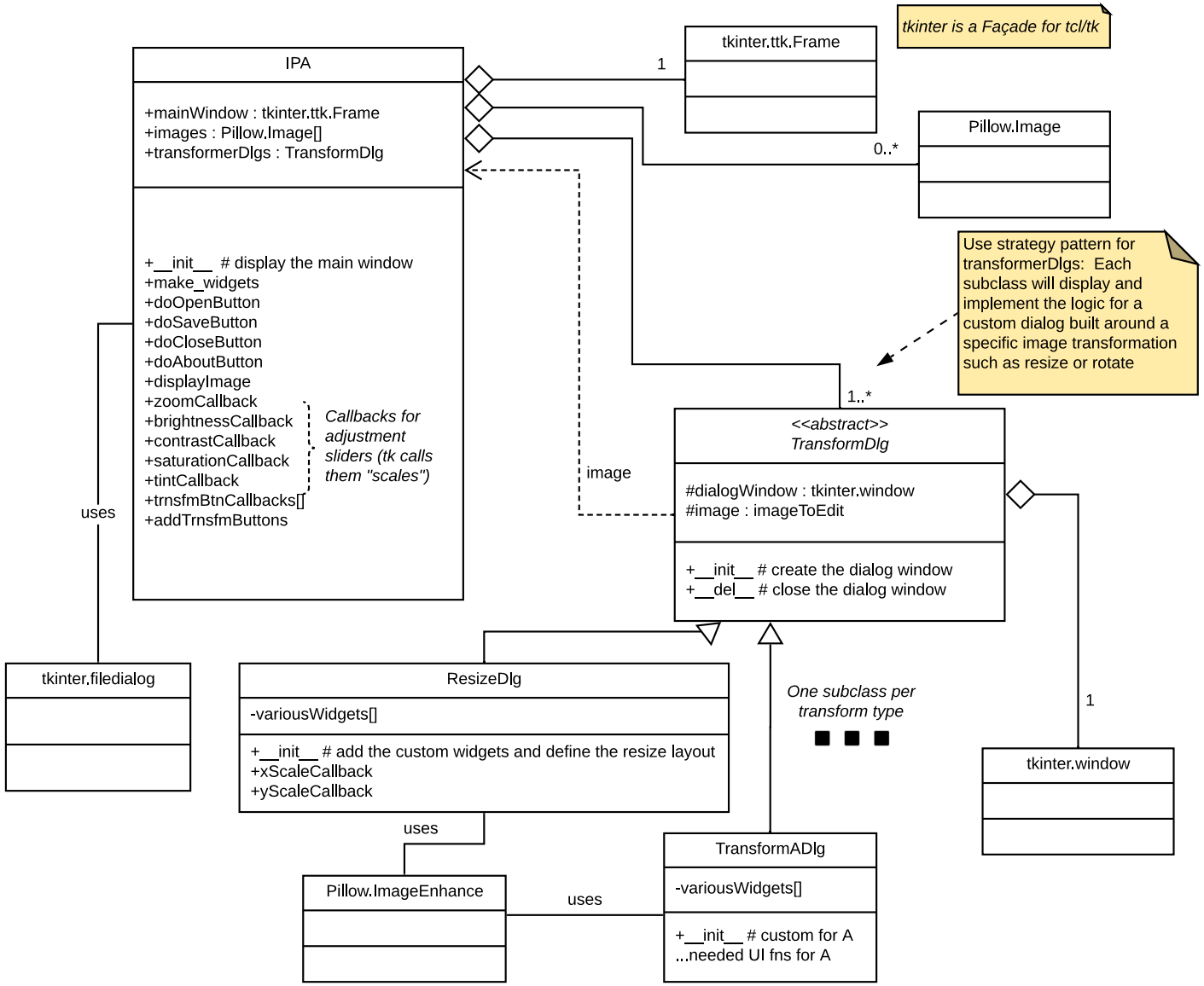
- Multiple image formats supported based on image file extension (BMP, EPS, GIF, ICNS, ICO, IM, JPEG, JPEG 2000, MSP, PCX, PNG, PPM, SGI, SPIDER, TIFF, XBM)
- Conversion of image formats – simply save an image with a different file extension and the image file format will match the extension
- Multiple images open simultaneously using a "tabbed" paradigm.
- Image editing:
    - Image resizing using a high quality "Lanczos" resampling filter; controlled via an intuitive slider interface.
    - Brightness adjustment via intuitive slider interface.
    - Contrast adjustment via intuitive slider interface.
    - Saturation adjustment via intuitive slider interface.
- Any or all of the above editing sliders may be adjusted in combination.
- The state of each editing slider is persistent with each image currently open – changing to a different image tab will 'remember' and restore the position of all sliders.
- If edits have been made to an image, then saving the image to disk will create an image file which reflects the edits made.
- The application window can be resized smaller than the current image being edited. If needed, scrollbars will automatically appear in the image area to allow movement within the virtual canvas.

Class Diagram:



Comparison to Initial Design

Class Diagram from Homework 4→



Discussion of Changes

The initial design called for the main UI code to be implemented in a single controller class. As I began to write this code, the coupling and interdependencies quickly became unmanageable. So, I took a step back and re-implemented it using the Model View Controller design pattern from "Head First Design Patterns". I based my implementation on the "djview" example code from that book. Note: I think my implementation is still a bit messy, with code in the View class which should be in the controller or the model. However, the structure is sound and could be cleaned up if needed.

Also, I had envisioned implementing advanced image editing dialogs using the Strategy pattern for a sort of "plug in" architecture. That is still possible for future enhancements, but it was not realized in time for the final project submission. Bonus: If time allows in the next week prior to my project demo, I may go ahead and add some of this enhanced functionality. If I do, I will be sure to clearly identify it as "not part of the original submission" during the demonstration.

Third-Party Code Vs. Original Code

The main program logic is contained in the IPAModel, IPAController, and IPAView classes. The structure of those three classes is based on the "djview" example code from "Head First Design Patterns". However, my application is substantially different from "djview", and Python is different enough from Java that I feel comfortable claiming that these classes were written by me.

The underlying GUI library (tkinter) is a third-party component, distributed as part of the Python Standard Library. I also made use of an open-source image processing library (Pillow), and the **ScrolledCanvas** class is based heavily on example code from "Programming Python, 4th edition".

List of Third-Party Code Elements with Cited Sources

- **Tkinter (GUI Library):** According to Wikipedia (https://en.wikipedia.org/wiki/Tkinter) "**Tkinter** is a Python binding to the Tk GUI toolkit. It is the standard Python interface to the Tk GUI toolkit, and is Python's *de facto* standard GUI. Tkinter is included with standard Linux, Microsoft Windows and Mac OS X installs of Python."

  https://docs.python.org/3/library/tkinter.html

- **Pillow (Image Processing Library):** "Pillow is the friendly PIL fork by Alex Clark and Contributors. PIL is the Python Imaging Library by Fredrik Lundh and Contributors."

  https://pillow.readthedocs.io/en/stable/

- Elements of my Model-View-Controller and Observer pattern implementations are adapted from example code in **Head First Design Patterns**.

  Head First Design Patterns
  Elisabeth Freeman, Eric Freeman, Bert Bates, Kathy Sierra
  Copyright © 2009 O'Reilly Media, Inc., Bert Bates and Kathy Sierra
  (Kindle edition, no ISBN given)

- **ScrolledCanvas.py:** The class **ScrolledCanvas** is based heavily on example code from **Programming Python, 4th Edition**

  Programming Python , Fourth Edition, by Mark Lutz
  (O'Reilly). Copyright © 2011 Mark Lutz, 978-0-596-15810-1

List of Original Code Elements of Design

- **IPA.py**: class IPA is the entry point of the application. It instantiates an IPAModel and IPAController, then tells the controller to start the application running
- **IPAModel.py**: class IPAModel implements the *model* portion of the MVC pattern for the IPA application.
- **IPAController.py**: class IPAController is the *controller* portion of the MVC pattern
- **IPAView.py**: class IPAView has all of the GUI manipulation code. It is the *view* of MVC.
- **ImageVitals.py**: contains **ImageVitals** – a data-only class (C++ would call this a structure not a class) which aggregates all of the vital information that needs to be retained regarding an image that is 'open' in the application.

Design Patterns

List of Design Patterns Used

- The **Model-View-Controller** pattern was used to implement the main logic of the program. The Model contains the code for image manipulation and persistence, the controller has the glue logic, and the View implements the GUI.

- The **Observer** pattern is also used between the Model and the View so that the View can subscribe to notifications from the Model when an image changes and needs to be refreshed on-screen.
- An **Iterator** pattern (Python built-in) is used as part of the Model's Observer pattern to iterate through the list of subscribers when sending notifications.
- Also, according to "Head First Design Patterns":
    - *Controller uses **Strategy**. The controller is the behavior of the view, and it can be easily exchanged with another controller if you want different behavior.*
    - *View also uses **Strategy** - the view is an object that is configured with a strategy. The controller provides the strategy*

Learnings on OOAD

For me, the heavy emphasis on up-front design was a bit too much. I did not have sufficient experience with my chosen toolset (Python 3 with Tkinter and Pillow) to make truly informed design decisions. I did my best to read through documentation and become familiar enough with the tools, but in the end the initial design was flawed in ways I didn't anticipate. Only after trying to implement my original design, and iteratively trying things to find what worked, did I truly understand the requirements of the toolset sufficiently to create a viable design.

This is all part of the normal application process. If left to my own devices, I would have started writing code snippets interleaved with doing the design work at a much earlier stage in the process.

Through the experience of designing this application, I have become a solid believer in the power of the MVC design pattern. When I initially started coding my design, I ended up with a single monolithic class which attempted to implement all of the GUI creation within a class constructor, plus a few extra methods for callbacks. I managed to spaghetti-code that together sufficiently to get multiple images loading and I got the "resize slider" working, complete with state persistence! (the slider 'remembered' it's position for each image). However, when I looked at adding more sliders for Brightness, Contrast, and Saturation, I realized my design was hopelessly flawed.

I found salvation in the MVC pattern. After carefully reading about MVC in the "Compound Patterns" chapter of Head First Design Patterns (HFDP), I decided to rewrite using MVC. According to my GitHub timestamps, it was 15 hours later when I finally had an MVC implementation that was functionally equivalent to my previous "spaghetti code" implementation. From there, the application development accelerated rapidly.

I am now 12 hours from the deadline to submit this project, and I am at a convenient stopping point. So, I will turn in the application as-is. Given more time, I could easily continue development and implement an expansion architecture allowing Strategy-pattern-based "plugins" for advanced image editing capabilities. Without too much trouble, I could also follow the example in HFDP and implement an alternate interface such as a web interface or a mobile device app.

In summary: I have learned that OO design with design patterns, while requiring some forethought to properly choose and implement relevant patterns, provides great benefits for code clarity and maintainability.