Programming Languages

Final Project – Assignment 1.2

Dr. Jan Springer

Name: Mitchel Gabriel H. Carreon **Date:** May 10, 2021

“So if you want to go fast, if you want to get done quickly, if you want your code to be easy to write, make it easy to read.”

- Robert C. Martin

Overview

Object-oriented and procedural programming, in a nutshell, are paradigms adhered by both professionals and students in the field of computer science. Object-oriented programming, "OOP" for short, encompasses concepts such as *encapsulation, abstraction, polymorphism,* and *inheritance*. Think: classes and its instances (i.e., *objects*). OOP intends to organize code in the hopes of achieving flexibility (which, especially, comes handy for later extension). Compassion. In contrast, procedural programming does not reciprocate such feelings. It has little to no consideration of the future. See, writing code in a *procedural* manner, from the word itself, means writing it such that it runs sequentially. That is all it cares about (if it cares about anything at all). It is convenient but short-sighted. Its apathy towards inevitable changes and numbness in welcoming such sympathetic ideals tends to create "baggage" that eventually boils down to add more tension (e.g., difficulty in finding bugs, difficulty in extending, tight-coupling, comprehensibility issues, etc.).

To elaborate further, we examine a couple of simple programs related to image manipulation; particularly, manipulating *.ppm* files. This document intends to fervently support and encourage OOP whilst maintaining an unyielding distaste towards procedural programming. It also serves as a long-winded README file for said programs. Arguments are substantiated using *Robert C. Martin*'s **SOLID** principles.

Disclaimer: It is often suggested to apply principles from multiple paradigms in mind when writing code. With that said, let us dive right in!

Note: To get a gist of how the program works and to save time, it is recommended to start at the section "How to use".

SOLID : Making the program

Object-oriented programming goes beyond the few concepts mentioned previously (e.g., encapsulation, inheritance, etc.). Robert C. Martin, one of the most influential experts in the field , introduced 5 principles to adhere to: **S**ingle-responsibility, **O**pen-closed, **L**iskov substitution, **I**nterface segregation, and **D**ependency-inversion. They serve as generic guidelines to ascertain your code is,

indeed, object-oriented; that is, adaptive, intuitively comprehensible, extendable. Here is a concise explanation of them:

- Single-responsibility– A class should only have a *single job*. Ensures related code are isolated in one portion of code.
- Open-closed– Code should be closed to modification but *open to extension*.
- Liskov substitution– Subclasses should be *swappable* to each other.
- Interface segregation– In creating classes, a *user-friendly* collection of methods should be presented to the client.
- Dependency-Inversion– Classes and the classes it depends on should be *loosely-coupled* using abstractions.

Such principles were considered in evaluating the soundness of each encapsulation, abstraction, etc.

Object-oriented programming

In both C++ and Python versions of the program, we have 3 abstract classes: *BaseImage, Pattern,* and *PatternFactory*. Each of them provides a group of related methods or an interface as to which we want the client to follow when extending (Figure 1). The intent is to have the bare minimum public; only those that are necessary are made available to clients. For instance, when creating concretions of the *BaseImage* class (i.e., *PPMImage*), we ensure that user-friendliness is encouraged by allowing only the abstract methods, *read()* and *draw(),* public. Though, you may observe that we have *getDimensions()* and *size()* as public too. Such concrete methods serve as helper methods for its dependencies, *Pattern* and *PatternFactory*, for them to accomplish their responsibilities. Extend accordingly.

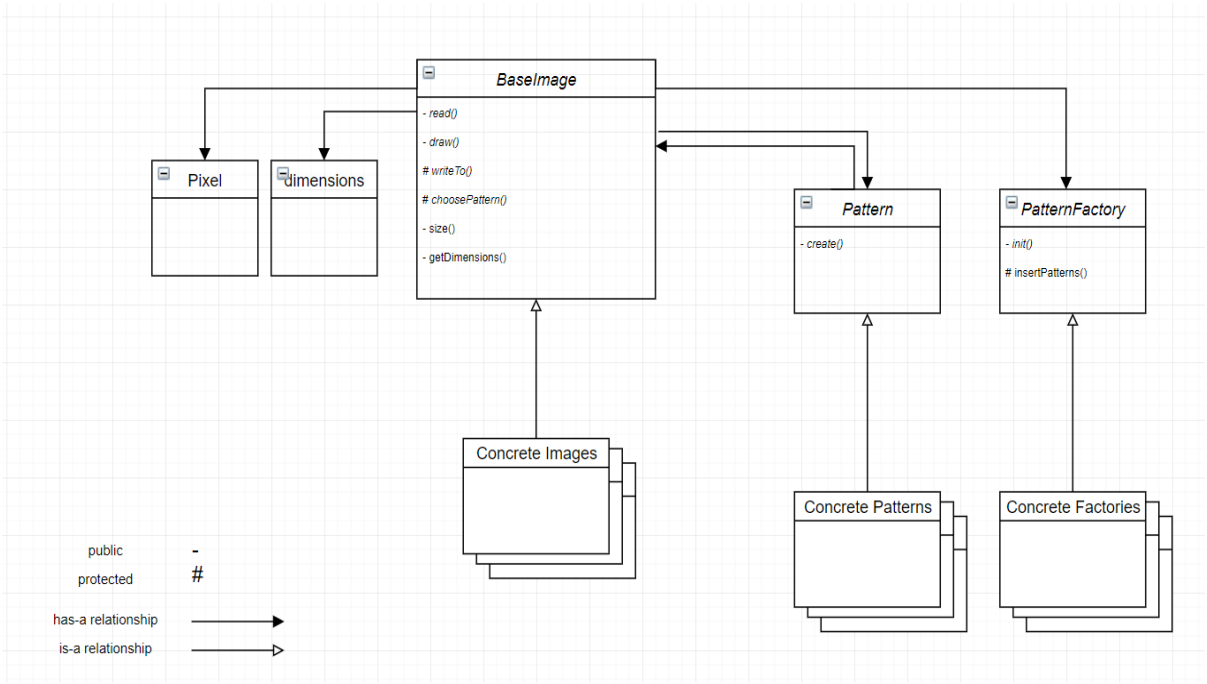

Figure 1: Program structure (Simplified UML)

BaseImage's dependencies are also abstract. Its child classes may be deemed as clients as well. The intent is to encourage these clients to follow the signature of its methods when subclassing. The return type and the parameter profile of each method serves as a guide for proper inheritance; proper, in the sense that it is within the bounds of the program's intent. By doing so, we can extend code without adding tasks unintended for said classes. Its job is its job.

```cpp
struct pixel {
    pixel() = delete;
    pixel(int red, int green, int blue)
        : red(red), green(green), blue(blue){}
    pixel(std::tuple<int, int, int> rgb) {
        std::tie(this->red, this->green, this->blue) = rgb;
    }

    int red;
    int green;
    int blue;

    friend std::ostream& operator<<(std::ostream& out, const pixel& pixel) {
        out << pixel.red << " " << pixel.green << " " << pixel.blue;
        return out;
    }
};
struct dimensions {
    int width;
    int height;
};
```

Figure 2: Abstractions aren't always necessary (C++ version).

Due to the relatively small size of *Pixel* and *dimensions,* they are made as concrete classes. Since they aren't aggregates and only exist in the context of a *BaseImage* (e.g., by this, I mean its concretions as abstract classes can’t be instantiated), they are defined within the class definition of *BaseImage*. The lack of abstract methods or pure virtual functions indicate the programmer's intent of having them untouched. Though its future is considered, the program limits itself to manipulation of Netbpm formats only and does not intend to go beyond that.

Specifics

How can you subclass and extend? Let's start by reiterating the exact responsibilities of each abstract class:

- *BaseImage* – Encapsulates all that has anything to do with a Netbpm image (e.g., pixels, dimensions, binary format, manipulating them, etc.). It is composed of an array of patterns, a theme (*PatternFactory*), etc. Everything you need to create a Netbpm format has been considered and can be found in this class' interface.
- *Pattern* – Encapsulates and isolates algorithms for writing awesome drawings (NOTE: the ones already built-in serve only as simple examples and are, inarguably, mediocre)
- *PatternFactory* – Encapsulates pattern instantiation. This initializes the *BaseImage*'s patterns array with a logical and intuitive group of *Pattern* concretions.

It is of utmost importance that extensions be made by following the signature of the methods within each abstract class. Failure to do so would create "baggage": half-hearted commitments to responsibilities, difficulty in finding & correcting mistakes, logic mismatch (comprehensibility issues), clingy (restrictive code; tight-coupling), longevity issues (hard to extend), etc. Basically, each time a client doesn't follow the program's logic, it gets worse. Eventually, a time will come when you must let go and move on. OOP is compassionate; we want what's best for the future. Change is inevitable, keep extensions in mind.

“Building the future and keeping the past alive are one and the same thing.”

- Solid Snake

Subclassing example

Here, we recreate a *.ppm* by inheriting from *BaseImage*. A couple of things to consider here. First, notice that it has its own member for storing the maximum color value of a pixel (e.g., 255, 10, etc.). The reason being is that this isn't common to all Netbpm formats. Binary formats (e.g., P3, P6, etc.), on the other hand, can be found in all Netbpm images. As implied in the snippet below (Figure 3), *_binaryFormat* is in the abstract class and is intended for internal use only (NOTE: python convention values underscores. A single trailing underscore indicates that it is protected).

```
class PPMImage(BaseImage):
    def __init__(self, dimensions, themes, binary_format='P3', max_color_val=255):
        super().__init__(dimensions, themes, binary_format)
        self._maxColorVal = max_color_val

    def read(self, file_name):
        with open(file_name, 'r') as file:
            for line in file:
                print(line)

    def draw(self, file_name):
        self._themes.init(self, self._patterns)
        self._choose_pattern().create()
        self._write_to(file_name)

    def _choose_pattern(self):
        print('Choose # from the list of built-in patterns')
        for pattern in self._patterns:
            print(pattern)
        choice = int(input('Input: '))
        if choice <= -1:
            print('CHOICES START AT 0. By default, pattern 0 is chosen for you.')
            return self._patterns[0]

        return self._patterns[choice]

    def _write_to(self, file_name):
        print('Writing to file...')

        with open(file_name, 'w') as file:
            file.write(f'{self._binaryFormat}\n'
                       f'{self._Dimensions.width} {self._Dimensions.height}\n'
                       f'{self._maxColorVal}\n')

            for pixel in self._pixels:
                file.write(str(pixel))

        print("PPM Created")
```

Figure 3: PPMImage class (Python version)

The method, *read()*, is overridden to get the preferred behavior the client desires when reading a format of this file type. Just like *read(), draw()* serves as part of the interface for non-subclasses. The responsibility of these methods is to restrict users to calling *draw()* when they want to make an image and *read()* when they want to see its contents. The program's intent is to present a generic interface by abstracting such logic within encapsulations.

```
class SimpleTheme(PatternFactory):
    def init(self, image, init_patterns):
        init_patterns.extend([
            PatternStyles.Stripes(image),
            PatternStyles.Waves(image),
                                ])
```

Figure 4: _patterns is modified (Python version)

The helper method, *write_to()*, must also be overridden. Now, this seems common to all image formats. However, making this method abstract allows control over the textual representation of the image. The method requires the client to pass a file name or a path as an argument; allowing flexibility in what directory it gets saved in.

In the same token, *_choose_pattern()* is also abstract; allowing clients to create their own custom user interface. In this example, its definition produces a simple "command line interface" that asks users which pattern they'd like to have displayed in the image. The *Pattern*, in turn, may then call *create()* to generate the correct pixel

```
#ifndef PPMIMAGE_H
#define PPMIMAGE_H

#include <string>
#include "BaseImage.h"

class PPMImage : public baseImage {
public:
    PPMImage() = delete;
    PPMImage(dimensions Dimensions, patternFactory* themes, std::string binaryFormat = "P3", int maxColorVal = 255);


    void read(std::string fileName) override;
    void draw(std::string fileName) override;

protected:
    pattern& choosePattern() override;
    void writeTo(std::string fileName) override;
    int maxColorVal;

};


#endif
```

Figure 5: PPMImage class (C++ version)

values. Now that all elements within the *Pixel* list is initialized, *_write_to()* loops throughout the list and outputs their values into a *.ppm* file.

Responsibilities must be well-defined and internal details should be abstracted away from clients by loosely-coupling with other classes. This is so that we can provide a solid foundation to work on. The intent is to prolong the program's longevity (no matter how it short it is). However restrictive your extensions may be, it is important to give some leeway for further extensions by your partner (e.g., future editors, etc.). It is suggested to partially define virtual methods to isolate commonalities of code, allowing easier debugging, among other things.

How to use

The program is quite compact and simple to use. After downloading all the source files, here are the things you need to do:

1. Import the *ImageManipulator* in your source file (*#include "ImageManipulator.h"* or *from ImageManipulator import **)
2. Create a *BaseImage* instance with the power of polymorphism (Figure 6)
    a. The *BaseImage* constructor takes two arguments. One is for its dimensions (width and height, respectively) and another for the user's preferred theme.
        i. For the Python version, you can pass containers like a tuple or a list for the dimensions. For the C++ version, you can either use brace-initialization (i.e., uniform initialization) or use the *Pixel* constructor.

```cpp
#include <iostream>

#include "ImageManipulator.h"

using namespace std;

int main() {
    baseImage* myImage = new PPMImage({ 300, 200 }, new simpleTheme());
    myImage->draw("Sample.ppm");
    delete myImage;

    return 0;
}
```

Figure 6: Using the program. (C++ version)

There are currently 2 themes: *SimpleTheme* and *ComplexTheme*. The former consists of 2 patterns: *Stripes* and *Waves*. The latter consists of just 1 pattern: *Psychedelic*. Note: As recommended several times, constructors/initializers of *BaseImage* concretions, follow the same signature as their parent's (as shown in Figure 6; instantiation of a PPMImage object).

Procedural Programming

This style of programming is just that: a step-by-step procedure. You simply present a logical sequence of execution. There is little to no organization. Responsibilities or jobs may be encapsulated by functions but, nonetheless, they are still separate entities. Classes are, inherently, object-oriented constructs. They provide organization and adaptability. Procedural programming is blind to such things. In this paradigm, we make use of functions and call them in a logical order. There is little to no benefit in creating code that is purely procedural. Doing so would hurt your program's reusability. Everything discussed previously was mostly about organizing code in a concise and intuitive manner. If you were to remove all these protective layers , you would end up with something, more or less, procedural.

If we merely wanted the functionality provided by the OOP code, for instance, all we must do is extract the algorithms from the *Pattern* concretions*,* the *write()*, and *read()* method of the *BaseImage* class.

```
void setImageDimensions(pair<int,int> dimensions, pair<pair<int, int>, vector<tuple<int, int, int>>>& image) { ... }

void choosePattern(pair<pair<int, int>, vector<tuple<int, int, int>>>& image, vector<void (*) (pair<pair<int, int>, vector<tuple<int, int, int>>>& image) >& patterns) { ...
void createStripes(pair<pair<int, int>, vector<tuple<int, int, int>>>& image) { ... }
void createWaves(pair<pair<int, int>, vector<tuple<int, int, int>>>& image) { ... }

void initializePatterns(vector<void (*) (pair<pair<int, int>, vector<tuple<int, int, int>>>&) >& patterns) { ... }

void writeTo(string fileName, const pair<pair<int, int>, vector<tuple<int, int, int>>>& image) { ... }
void read(string fileName) { ... }

int main() {

    pair<pair<int, int>, vector<tuple<int, int, int>>> image;
    vector<void (*) (pair<pair<int, int>, vector<tuple<int, int, int>>>&) > patterns;

    setImageDimensions(make_pair(300, 200), image);
    initializePatterns(patterns);

    string fileName{ "Sample.ppm" };
    choosePattern(image, patterns);
    writeTo(fileName, image);

    read(fileName);

    return 0;
}
```

Figure 7: Concise conversion using the procedural paradigm (C++ version)

In this case, we still try to structure the program in a logical way. Figure 7 depicts how we run through each task in a sequential manner. The ability to extend is not possible, however. If the client intends to add on to this, he is compelled to modify existing code. For instance, to add more patterns, the function *choosePattern()* must be changed. We try to provide some sort of interface for the client as well. The signature of the functions, *createStripes()* and *createWaves()*, tells a pattern on how to do so.

The program also makes use of a vector of function pointers to organize these *create[PatternName]()* functions and is utilized by *choosePattern()* to initialize the second element in the pair, the vector of integer tuples.

The relative complexity of this code makes this difficult to comprehend for such a small program. It'll be hard to difficult to extend without modifying existing code. Object-oriented programming tells us that modifications aren't ideal. It introduces unintended behavior or bugs; may it be logical or syntactical. Tasks are, somewhat, encapsulated by isolating them in functions. However, they are tightly-coupled. Changes to one area of code compels the client to make changes to others: a ripple effect.

Of course, you can make containers that contain duplicate information (e.g., file name, dimensions, etc.) available to the client. Nonetheless, the coupling is still too restrictive. This style of programming is convenient and easy to create. But it isn't ideal for long-term use as changes will inevitably come and extensions must take place. This style is ideal if the author deems that the program will not be extended for future use. Web-scraping is one good example as websites' internal structuring are, often, different to each other; almost unique.

It is worth noting that this isn't purely procedural. We are still using some standard libraries which contain classes, structs, and other constructs that makes use of object-oriented concepts.

Note: To exacerbate the ease of making code procedurally and caring only about its functionality, take a look at the Python version of this.