Homework 3 Final Report **Ilan Biala, Nick Roberts, Varun Sharma**
17-480, Fall 2018 ibiala, nroberts, vsharma

1 Why Processing?

Processing is an API made for designers who want to develop graphical interfaces quickly. Unfortunately, Processing is an API made *by* these same designers.

Depending on who you ask, Processing is either a language or a library. But let there be no mistake: the Processing language is Java 6 minus the top-level class. In this report we instead focus on the `processing.*` Java package.

Processing poses a unique challenge because of its target audience: newcomers to Java who want to throw together a quick and dirty interface. With an inexperienced audience, an API designer's goal is simplicity, but this goal is complicated by the fact that you can't assume familiarity with the design patterns common to the platform. Under this complication, you could imagine API designers conceding many good practices of design in the name of simplicity. Whether these concessions are valid is not a straightforward matter.

We think that Processing is plagued by too many of these concessions. You would be hard-pressed to find a method that does not modify shared mutable state, and, as such, interactions between user-written methods are hard to predict. Even something as simple as drawing on a subcanvas of the original canvas requires either complex coordinate arithmetic or state mutations. In this report, we describe our creation of a wrapper for Processing that draws from a modern API designer's toolkit while retaining the simplicity that would allow it to continue enjoying wide use in the HCI community, including CMU's own HCI institute.

2 Which sub-APIs of Processing did we consider?

Official Processing documentation: processing.org/reference

Processing is a large API, and fixing it in its entirety is outside the scope of this project. Instead, we focus on these sections of the linked documentation:

- **Drawing shapes with user-chosen settings.** These methods, like `rect` and `ellipse`, are used for drawing a variety of shapes on the canvas. There are also methods, like `fill` and `stroke` that modify properties of the color and outline weight of shapes drawn on the canvas.
- **Transform.** This is the API to apply matrix-based transformations to the canvas on which the user is drawing. It includes rotation, scaling, shearing, and translation.
- **General structure of a Processing program.** Currently, setup code is placed either as a top-level definition or as part of the `setup` method. The `draw` method is called once per frame. The user can draw on the canvas from any method, including the setup method.

3 Use cases where Processing falters

1. **Drawing on a subcanvas of the main canvas.** There are two ways to do this: manual arithmetic, and applying transformations. Either one is problematic. The first is error-prone and difficult to refactor; the second is error-prone if you omit a call to `popMatrix`

or `pushMatrix`, which save and restore the previous active canvas transformation, respectively.

To draw on a subcanvas at position (200, 200) on the main canvas, you could use this code:

```
// Manual arithmetic.
rect(200 + rectX, 200 + rectY, rectWidth, rectHeight);
text("Hello world!", 200 + textX, 200 + textY);

// Applying a mutable transform
// but don't forget to call popMatrix() afterward!
pushMatrix();
translate(200, 200);
rect(rectX, rectY, rectWidth, rectHeight);
text("Hello world!", textX, textY);
popMatrix();
```

2. **Controlling properties of drawn shapes, like color, weight, and position.**

   The color of a drawn shape is determined by the last call to `fill`; likewise, the border is controlled by `stroke` and the positioning scheme by `rectMode`. Bizarrely, this code has different meanings depending on the last call to `rectMode`:

   ```
   rect(rectX, rectY, rectWidth, rectHeight);
   ```

   `(rectX, rectY)` denotes either the upper-left corner or the center; and `(rectWidth, rectHeight)` denotes either the dimensions or the position of the bottom-right corner. This scheme where settings persist between calls to functions (like `fill`, `stroke`, and `rectMode`) is very error-prone.

3. **Creating images (Which functions can be called from where?)** This code is illegal as a class-level field definition:

   ```
   PImage image = createImage("dogs.jpg");
   ```

   But this is legal:

   ```
   PImage image;

   @Override
   public void setup() {
     image = createImage("dogs.jpg");
   }
   ```

   That is, some functions can be called only from the `setup` function. The user encounters runtime errors when code is run improperly. We want to offload this work instead on the library by making it impossible to write illegal code.

4 Implementation of our wrapper

We designed a wrapper for Processing that specifically addresses the failings we note in the preceding section. We'll show some code samples in this section, but we encourage the dedicated reader to jump into the use case code samples that accompany this project submission.

1. **Subcanvases.** We reify the canvas as a Java class, `Canvas`, upon which shapes can be drawn at offsets. The killer app of these is *subcanvases*: you can create a small canvas, draw images and text on it, and then place the canvas to be drawn on a larger canvas.

   Our design of canvases retains one aspect of mutability from the Processing design: the entities drawn on that canvas, and their order, are stored in mutable state. This allows the client to write a sequence of statements that looks like this:

   ```
   canvas.draw(Rectangle.of(4, 5),
     ShapeSettings.createWithFill(Color.RED),
     Position.centeredAt(10, 10));
   canvas.draw(Circle.ofRadius(1),
     ShapeSettings.createWithFill(Color.BLUE),
     Position.centeredAt(10, 10));
   ```

   Such a program would result in the blue circle being overlaid on top of the red rectangle. This is the only mutable aspect of our API.

   To write a Processing application with our wrapper, the client still must provide a `draw` method, but the specification is different: given a canvas, the user must perform the sequence of calls to draw on that canvas. Instead of writing a method like this:

   ```
   @Override public void draw() { /* Update global mutable state */ }
   ```

   the client will write a method that looks like this:

   ```
   @Override public void drawFrame(Canvas canvas) { /* Draw on the canvas */ }
   ```

   The overall effect of this change is to allow a more natural interpretation of a transformation: it's a way to draw on a different canvas from the main canvas. This is evident in the attached use case code for an animated screensaver with nested canvases; raw Processing code is also attached for contrast.

2. **Settings.** We decouple the creation of shapes (a thing with a geometric form, a width, and a height) from the creation of settings (color and outline). Like canvases, we reify the concepts of shapes and settings as *data*: immutable Java classes that can be returned from methods, passed as arguments, and stored in data structures.

   For simplicity, the creation of settings involves no intermediary builder class, since this might be unfamiliar to the user base of Processing. Instead, you can create instances with static factory methods, and chain `with`*Property* methods to construct your own settings. This makes it simple to change only one parameter of an existing settings object.

   A more advanced user may wish to take advantage of some facts:

   - Shapes, like rectangles and circles, have something in common: they can be associated with shape settings (like color and outline) and drawn on a canvas.

- Canvases have something in common with shapes (once they've been associated with shape settings): they can be drawn on the screen.

Making note of such facts, an advanced user might want to be able to write a method that returns any kind of shape, associate settings with that shape, and draw it on the screen. Alternatively, they might want to write a heterogeneous data structure that stores many drawable objects (such as canvases and shapes-with-settings), retrieve objects from that data structure, and draw them on the screen.

Our API's class hierarchy makes such a program structure available to advanced users while allowing novice users to use the API in a less complex fashion. We introduce `Shape` as a common superclass to rectangle and ellipse; and we introduce `Drawable` to represent either a canvas, a shape packaged together with shape settings, or an image packaged together with image settings. Intuitively, a drawable is something that's ready to be drawn on the screen (once you provide a position). This staged approach to drawing something on the screen affords the client better abstractions than the flimsy barrier between user and system provided by Processing. The novice user can continue to call `Canvas#draw` with `Rectangle` and `Ellipse` arguments directly, permitting them to avoid the indirection entirely.

The internal implementation of such a class hierarchy is decidedly inelegant: Java has terrible support for tagged union types, forcing us to store a runtime representation of the type (or, equivalently, using `instanceof` checks). Graciously, the client of our API is spared so long as they don't peek inside the black box.

The attached solar system simulator is a great example of reuse of settings code.

3. **Top-level structure of Processing app.** We already modified the Processing applet interface (`void drawFrame(Canvas canvas)` instead of `void draw()`), and so, at no extra cost, we can fix the structural problems with launching an applet. We write a wrapper method `ProcessingApplet.run` that runs the code in the provided Processing app—*but only after the Processing* `setup` *method has been called.* This culls the illegal state space that was previously within the client's reach.

   Previously disallowed programs are now easy to write, as evidenced in the attached code for loading and displaying some images from file.

5 On the unusual effectiveness of immutability in API design

Immutability makes for clarity in design; principled deviations from immutability smooth over some of the more complex design patterns associated with it. The strengths of our API lie in the construction and use and re-use of settings and shapes; areas for improvement include the class hierarchy, especially as it applies to user extension.

5.1 Strengths

Our API fixes many aspects of Processing and provides a framework for fixing other problems. To name a few fixes:

- It replaces the mutability that plagued Processing with a way of constructing immutable objects and drawing them on the screen with user-chosen settings.
- It provides a principled way of constructing an object to draw in stages. (First the base object, then the settings, then the position.)

- It constructs explicit representations of Processing's computations as composable Java objects, most prominently replacing the translation and transformation *actions* of Processing with the `Canvas` *class*.
- It removes some of the illegal states that Processing permitted.

5.2 Areas for improvement

- More complete coverage of the Processing APIs. We did not attempt to fix the event handling framework or to integrate event handling into the draw loop; this integration would require more careful thought as to how data should be passed between the draw loop and the event handlers.
- A user-extensible class hierarchy. Currently, the `Shape-Drawable` class hierarchy relies on internal reconstruction of the underlying type of a drawable object; the canvas drawing code can case on this representation to make appropriate calls to the Processing library. This forces the library to deny the client avenues to extension—for example, the client can't create subclasses of `Shape` since the library wouldn't know how to draw them. Allowing the user to register new shapes would therefore require them to specify how the shape is to be drawn, and without exposing full access to the underlying Processing API, such an extension would necessitate API design beyond the scope of this project.

6 Why we did not write unit tests

The algorithmically interesting part of our API is the interface between the simplified API we wrote and the underlying Processing API. This is the canvas update code: after a call to `drawFrame`, the resulting canvas is committed to the screen by recursively committing each of its constituent elements at its designated position. *The most error-prone part of this procedure, and the part that would be desirable to test, is the code that calls the underlying Processing API to set settings, occlude parts of the screen that should not be drawn on, and actually draw the shapes to the screen.* That is, we would want to write unit tests that confirm that our understanding of the Processing API is correct: if I set this setting here, and this setting here, and draw this shape here, the outcome on the screen is what I expect.

The only way to test the interface between our program and Processing is by visually inspecting the result on the screen. Writing unit tests would be futile because, in our attempt to mimic Processing's logic in test code, we would likewise mimic any misunderstandings we have of Processing's API.

As recompense, we provide clear, runnable use case code.

7 Description of work

Our main issue was building a city on top of a live volcano. Processing, like the citizens of Alaska, has a lot of state. Extremely useful methods (like `clip`, which confines the legal drawing area to a subrectangle of the screen) are buried in pages of cruft.

Ilan, Nick, and Varun collaborated extensively on the underlying API design. All of their opinions and design decisions are reflected in the final draft. Nick was responsible for the implementation of the class hierarchy of shapes and drawable objects and led compilation of the submission documents. Ilan wrote the code for settings and led the development of use-case code. Varun led the development of the new top-level API for Processing apps and wrote the algorithm for committing results to the screen. All collaborated on the submission documents.