CS193J Handout #17
Winter 2001-02

Design strategies for hw2

HW2

We're going to continue the trend of separating the assignment handout into two parts. The official assignment handout discusses all of the functional requirements while this handout offers ideas and strategies for design. Read the assignment handout first to get familiar with the scope of the problem and then use this one to get an idea of how you might approach it. This handout describes the code in the starting implementation and suggest strategies for evolving it into the final goal and what milestones to shoot for along the way. We also have some notes about the sticky parts you might encounter and discussion about good use of inheritance. Hope you find our suggestions worthwhile and helpful, even if you eventually decide choose different paths of your own for completing the assignment.

A roadmap to the starting implementation

There are six classes provided in the starting project. Each of the source files is reasonably well-commented, but let's take a look at each to get an overview:

JavaDraw.java

This is the main class for the application. It only has two methods, the static main() that kicks the whole thing off and a menu creation method. The given code instantiates the drawing window and creates and install the drawing canvas, toolbar, and menus. All of the code in this class works correctly as is and should require no changes.

Setting up a user interface in Java tends to involve tedious calls to create components, configure their parameters, and make lots of tweaks to lay them out into visually pleasing ways. Scan the code we give to see what I mean. Our code creates menus and menu items, sets keyboard accelerators and registers event listeners to pass along the menu commands to the drawing canvas. There are many anonymous inner classes used as listeners, this is a very common technique and one that you will want to become familiar with.

Toolbar.java

The Toolbar sets up and manages the group of tool buttons and color selection button installed along the bottom of the drawing window. It has a handful of public methods that allow a client to set and get the current tool and color and register a listener to be notified on color changes. All of the code works correctly as is and should require no changes.

The different tool buttons form a group where at most one button can be selected at a time—choosing a different tool always deselects the previous choice. The background of the color button shows the currently selected color. Clicking the button brings up the color chooser which allows the user to select a different color. Whenever the current color on the toolbar changes, the toolbar notifies any objects registered as change listeners by sending them a change event. Although you shouldn't need to edit the code in the toolbar, it is worth going over it to see the various details that need to be handled to set up this part of the user interface. The drawing canvas will need to interact with the toolbar object, using the methods to set and get the settings as well as registering a change listener.

SimpleObjectReader.java and **SimpleObjectWriter.java**

Like the two I/O classes we gave you for HW1, these classes are simple wrappers around standard java.io stream classes with the added feature that they handle the exceptions for you. The writer class has only three methods: a static method to open a new file for writing, a method to write an object and a method to close the file. The reader class has a parallel set of three methods for reading. The objects that are being read and written must implement the Serializable interface. All of the code works correctly as is and should require no changes.

Rect.java

The Rect class defines a simple rectangular shape object. It tracks its bounding box, selected state, and the canvas it is being drawn in. It has methods to select, move, and resize itself. It can properly draw itself and updates the canvas whenever the state or bounds of the rectangle change. The given code works properly, but you will need to extend and change the class to support additional features.

Most of the given code will end up consolidated in a common base class from which all your shape variations will inherit. Rearchitecting the code to support inheritance will require some careful planning and design to maximize code sharing. We recommend carefully thinking through your strategy before rearranging any of the code.

DrawingCanvas.java

The DrawingCanvas is a subclass of JPanel that provides a drawing surface for creating and modifying rectangles. It keeps an ArrayList of Rects, each of which is responsible for tracking its own size and position. The canvas paints by iterating over its rectangle objects and telling each draw itself. The canvas stores a selected rectangle. Mouse events on the canvas are handled by an inner class that listens for mouse events. The inner class determines which rect is under the mouse and then messages it to move or resize as the mouse is dragged. The code in the canvas class works properly, but you will need to extend and change the code to support additional features as required. The DrawingCanvas needs to eventually support all four shape types, as well as the additional operations for using the clipboard, layering, colors, and saving and loading files.

Getting your bearings

The first thing you should do is just compile and run the provided code and play with it. Drag out new rectangles, select and deselect them, move and resize them, and get a general feel for how the program works. Try out fringe cases and see if you can find the program's weak spots. Ask for your money back if you expose a bug we failed to find.

Next, take a look at the given code, in particular, focusing on the DrawingCanvas and Rect classes since those are the two that you are going to extensively modify. Map out for yourself what responsibility each object has and how it interacts with the other objects. Look carefully at the starting design and evaluate how well it meets the goals of a good obejct-oriented program. Do objects encapsulate their state properly? Are the methods well-designed to maintain object consistency? Do objects take responsibility for their own behavior? Are the relationships between the various classes clear and easy to understand?

Now, study how the graphical aspects are managed. How do the canvas and shapes cooperate on drawing? How does the shape track the state to be drawn? How does the canvas refresh when changes have been made? You will want to have a good understanding of the basic framework so that you will be able to make changes without breaking any of the given behavior.

Although the assignment handout lists the new shapes as the first task in the list of requirements, it is actually one of the harder jobs you have to do and one that requires the most skill. We recommend keeping the program as just a rectangle program for a while and adding in some of the easier features and then as you gain more confidence, tackle the more complex task of introducing the new shape subclasses later on.

Layering

The straightforward layering commands make a good starting point. The menu items have already been created and set up to send the canvas the appropriate front/back methods, your job is just to fill in the missing implementation. How does the canvas record and control the layering of shapes? Study the canvas class to learn what it required for a shape to be moved "above" or "beneath" all others. What will be the appropriate way to redraw the affected part of the canvas? Be sure to test all the various fringe cases such as when there is only one shape to reorder or no selected shape or you to move the topmost shape to the top and so on. Also, see the ArrayList docs. Testing hint: it's hard to determine the layering when all shapes are the same color, so you may want to temporarily have each rectangle be created with a randomly chosen color so that you can more easily see the layering relationship.

Colors

Adding color support has a bit more to it than layering, but not too much, so it makes a good second task. Start by making new shapes take on the current toolbar color. How do shapes currently decide what color they are drawn in? How can you change that? When a new shape is being created, how can the canvas determine what color to use? Once this is working, set things up so the toolbar updates to the color of the currently selected shape. Finally, hook up a listener so that changing the current color on the toolbar changes the color of the selected shape.

Clipboard

There is a Clipboard class in the java.awt package, but unfortunately, its design is a bit cumbersome and its features are overkill for what we need. We recommend ignoring the system clipboard and creating your own clipboard class that can handle the simple needs of your program. Your clipboard only needs to support storing a copy of the selected shape and later retrieving it, nothing more fancy than that. The cut, copy, and paste menu items are already set up to send the appropriate messages to the canvas, you will need to fill in their implementation to work with your clipboard. You will need to make a copy of the selected shape to store on the clipboard because storing just a reference will lead to aliasing between the original and duplicate shape (i.e. moving one moves the other or changing one's color changes the other which is not what you want!) The Cloneable interface and the clone method will be the tools you use to make a shape capable of producing a duplicate of itself. Be careful when implementing the clone method— it needs to make a full deep copy so that there are no unintended shared references between the two shapes. Be sure to test all your clipboard operations thoroughly in all their variations to shake out any lurking bugs.

Serialization

In order to save and load drawings from files, you will need to be able to archive the shapes. As is customary in the OO paradigm, each object that needs to be saved should take responsibility for writing itself out and reading itself back in. The Java means to do this is by having the object support the Serializable interface and potentially override the readObject and writeObject methods to customize the process. (We’ll get to serialization in lecture shortly.) With serialization, writing out a drawing mostly becomes a matter of having the canvas ask the shapes to write themselves. Loading a drawing means having the shapes read themselves back in. When making an object support serialization, you will need to make careful decisions about exactly what state needs to be written and read, as well as accounting for any additional actions that need to be taken when re-constructing the drawing from the newly read shapes.

The Shape classes

The canvas needs to support a number of different shapes, each having some features in common with the others, while other features are unique. A common parent class can be used to capture the similar elements, while inheritance and overriding can produce specialized variants.

The starting Rect has methods for creating, selecting, drawing, moving and resizing rectangles. Although that code that is there is specific to rectangles, a lot of it can be made to work in a common fashion for all shapes. Where possible, you want to organize the functionality to be shared. You will also need to add new features like colors, cloning, etc. to all shapes in a way that unifies all the common code.

Designing an object hierarchy

Taking the starting code and re-architecting it into a related set of shape classes is an excellent exercise in designing an inheritance tree. One of your most important jobs is going to be laying out the tree and figuring out what data and methods go with each object. We encourage you to sketch things out on paper and plan your strategy before writing a lot of code since you want to avoid committing to a design that would be difficult to implement. The most important part here is learning how to enable code re-use by factoring shared code upward in the hierarchy.

The different shapes have a lot of behavior in common— the challenge of the assignment is structuring your classes so as to avoid repeating code. You should also be able to add other types of shapes without a lot of modification. This means you will want many small well-named methods. Although may of the methods will be quite short (a line or two) by factoring into methods of their own, you enable subclasses to override the method and easily extend or change details while still inheriting all the common behavior with no effort.

Where code is not quite the same, you do not want to copy and paste methods from class to class, making small modifications. Instead factor up all the common parts and move the different pieces to small helper methods that can be overridden as needed. With all common behaviors factored out to the parent class(es), only the extensions and differences appear in the specialized subclasses. Don't give in to the temptation to just duplicate even a few lines of between similar methods. Committing yourself to making inheritance work for you on the small things will help you reap the larger benefits on inheritance when designing more complex hierarchies later.

The parent class declares the common methods, those that can be reasonably sent to any of the subclass varieties. Whether or not there is a good default implementation will decide whether the method is abstract. Be careful to not push behavior up past where it is appropriate, a parent class should only declare messages that make sense for it and all subclasses. If you find yourself subclassing and wanting to "take away" behavior from the parent rather than extend it, it may be a sign your design needs some work.

If you have done a good job, when you're done it should be easy to add other shape variants with minimal effort. There is a fair amount of latitude in the way you can construct a reasonable set of classes. The most important expectation is that your code uses sensible object-oriented design. Your classes should present a clean abstraction that successfully coordinates with other classes through a reasonable interface. Classes should take responsibility for their own behaviors (displaying, changing status, etc.) rather than being externally manipulated. Code should not be repeated between methods or classes. And so on...

Ovals

The oval is probably the easiest subclass since it is so similar to the rectangle. The only mildly complex issue for the oval is doing hit-detection to determine if a given point is inside the oval. Rather than force you to consult your old geometry textbook, some pseudocode to guide you:

```
x and y are the point we are testing
centerX and centerY are the center of the oval
height & weight are the oval dimensions

May want to enlarge width/height to allow for a little extra tolerance
around outside edge of oval and avoid divide by zero in following code

double normx = (x - centerX) / (width/2.0);
double normy = (y - centerY) / (height/2.0)
inside = (normx * normx + normy * normy) <= 1.0;
```

Lines

The line has a few little quirks of its own. Although its bounding box behavior is the same as rectangles, it also needs to track the direction of the diagonal across the box. It also is a bit different in the knob handling since a line only has knobs on the two ends. However, with some careful planning in the Shape parent class, you can make sure that the Line only has to make minimal changes to override and tweak the behaviors as necessary.

The hit-detection for a line can be a bit messy. Here is some psuedocode you might find helpful. This computes the angles of the two points relative to the start and then invokes good old Pythagoras to compute the distance of the point away from the line. Figure that any click within 5 or 6 pixels is "close enough" to be considered a hit on the line.

```
startx & starty is one end of line, endx & endy the other
x and y are the point in question to test

Before doing anything complicated, make sure the point is within the
bounding box of the line  -- enlarge the rect a bit (6 pixels) to allow for
some tolerance, if not inside that box, don't bother with the rest here

width = endx - startx, height = endy - starty;
dx = x - startx, dy = y - starty;

 if (dx != 0 && width != 0) { // if both lines are not vertical
     angleToCorner = Math.atan(height/width);
     angleToPoint = Math.atan((dy/dx);

     distance = Math.sqrt(dx*dx+dy*dy)*
                     Math.sin(Math.abs(angleToCorner-angleToPoint));
 } else       // one or both slopes have zero divisor (are vertical)
     distance = Math.abs(dx);
inside = distance < tolerance (5-6 pixels?)
```

Scribble

The scribble is the most divergent of the shapes, but still has a lot in common with the others and can leverage a lot of the shared behavior. Its geometry cannot be expressed with just a simple bounding box, since it also has to track the sequence of points along the scribble path, so it will require some extra state and handling to work correctly for the various operations. As usual, try to make use of as much of the common code as you can and only override where absolutely necessary. I think you'll find the Scribble the most interesting of the shapes to implement since it does offer some neat challenges in unifying its behavior with the others.

Hit-testing for a Scribble involves hit-testing over a lot of line segments, so it needs to incorporate the same logic presented above for the Line. Take care to unify the line-point-distance code that will be needed by both Line and Scribble. This is not necessarily through an inheritance relationship, but perhaps a shared a helper static method that both use, but it is just good form to not duplicate code in more than one place.

Incremental testing

This program is complex enough that just starting to add code wherever and hack your way to completion is unlikely to be a fun or successful endeavor. The program we gives you work as is, take is as a goal to always keep your program not far from a working version as you go. Make small changes, test them thoroughly, and then move on to the next step. Adding 300 lines of code that add 4 new features at once without even stopping to compile along the way is a sure means to make debugging an exercise in torture. If you just added the cut command, you can focus on testing it in all uses without worrying about what other unintended side effects are coming from the other things you changed at the same time. As you add each new shape subclass, thoroughly test it before moving on. Does it properly select and deselect? Move? Resize? Change colors? Cut and paste? Only when you're sure it has all the required features working should you move on to the next shape challenge. This strategy will be more effective and satisfying, and as a side bonus it ensures that you're always near a good stopping point. If you run out of time near the end of the project, would you rather submit a program that attempts 100% of the functionality and gets it right 80% of the time or a program that has 80% of the features perfectly implemented?

A few random details

- All the previous class design guidelines still apply (compiles cleanly, no public instance variables, functionality in right place, etc.).
- Although it might be tempting, you should avoid using the runtime type identification (i.e. `instanceof` and `getClass`) to figure out what class an object is so you can make decisions based on its class. You should instead design things so you send the same message and expect the objects to respond in their own way through the miracle of dynamic binding. This is the value of polymorphism— don't figure out what class an object is and construct a switch or if statement to route control yourself!

- You might find it convenient to take a class name expressed as a String or a Class object itself and create an instance of that class, such as creating a shape by name or class. If you're curious how to do this, check out the documentation on java.lang.Class which has methods that allow you to retrieve a class by name and create an instance.