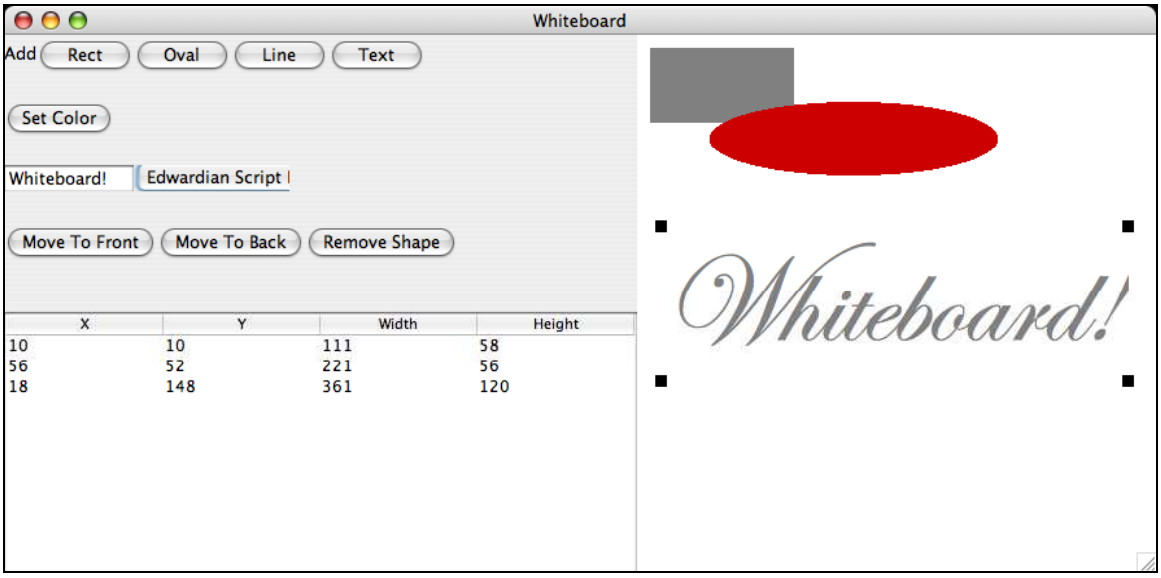
HW5 Whiteboard

Thanks to Nick Parlante for much of this handout

For this assignment, you will build a simple but functional Whiteboard/drawing program, such as you might use to make a diagram in a lecture. Part A of the program deals with the GUI and drawing. Part B deals with file saving and some networking operations that we have not covered yet. The whole thing is due midnight at the end of Mon Nov 10th. Unfortunately, absolutely **no lateness** is allowed on this assignment so that all the teams are ready to get going on the final project the next day -- think of it as giving you practice in the valuable skill of hitting a hard deadline. As usual, we will have extra office hours in the evenings immediately before the due date.

Whiteboard is, in its way, a fantastic program to build and understand, since it has examples of every important feature that a real application needs: presenting state visually, editing that state with controls and mouse gestures, separating the model and the view, saving and restoring that state from the file system. Many computational problems and situations that you will confront in the future, you will see and solve first on Whiteboard. Here's what the part A Whiteboard looks like when working…



Canvas and Whiteboard

There is very little starter code for the Whiteboard project -- this time you will create your classes from scratch, and of course it's fine to pull boilerplate code from the many lecture examples. The `Canvas` class should be a subclass of `JPanel` with a white background and an initial size of 400 x 400 -- it will contain the little drawing shapes. Create a `Whiteboard` subclass of `JFrame` that sets up the components in a frame, and its `main()` should create a single `Whiteboard` frame. Install the `Canvas` at the center of a border layout, and position the controls in the west. To fit all the controls on screen, I put groups of related controls into a horizontal box, and then put those boxes into a vertical box in the west. Your GUI layout should be functional, but does not need to look exactly like ours. By default, all the components in a vertical box are centered, which does not look good for our purpose. You can call `setAlignmentX(Box.LEFT_ALIGNMENT)` on everything within the box like this...

```
for (Component comp : box.getComponents()) {
    ((JComponent)comp).setAlignmentX(Box.LEFT_ALIGNMENT);
}
```

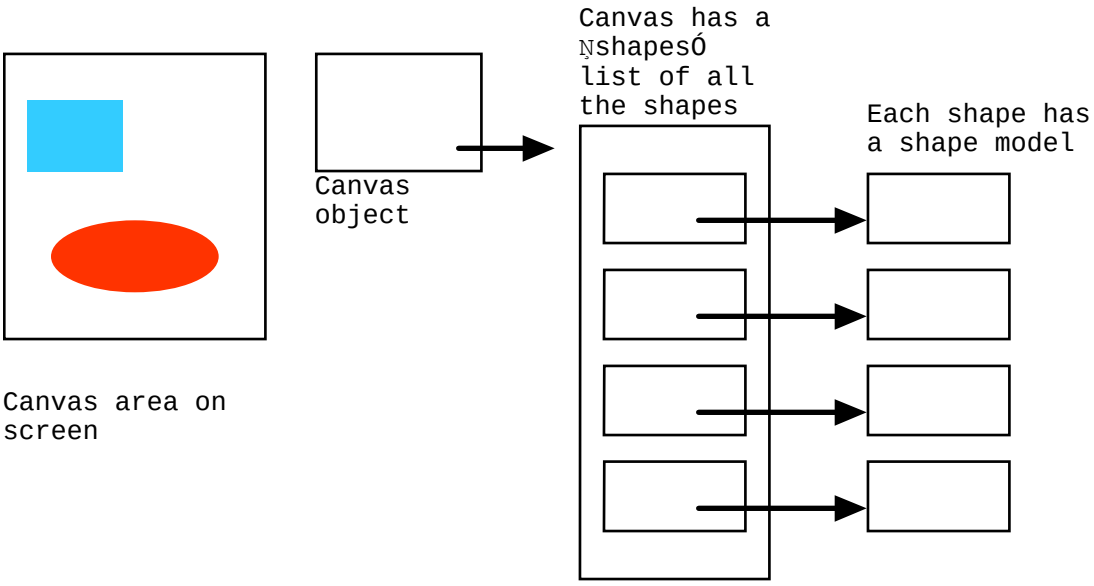
DShape and DShapeModel

We will now outline a suggested strategy for gradually building up the `Whiteboard` functionality. There are many different choices you can make about the design of individual methods, and we are not requiring a particular code solution, other than the requirement that the shapes do not store model data.

We will have three types of shape: `DRect`, `DOval`, `DLine`, and `DText`, all under the superclass `DShape` -- there is a great opportunity to use inheritance to eliminate code duplication between the 4 subclasses and the `DShape` superclass. The canvas should have a "shapes" list of its current shapes, and the canvas `paintComponent()` should loop through all the shapes and draw them. The shapes are not subclasses of `JComponent`; they are just regular java objects owned by the canvas. The shapes fill the "view" role, representing something to draw, but they do not store the data itself (i.e. MVC). The shapes list in the canvas is effectively the "document" the user is editing -- whatever is in that list is the document we are editing, and removing a shape from that list removes it from the document. The shapes list also defines a back-to-front ordering of the shapes, with the last shape in the list appearing "on top" of the other shapes for drawing and clicking.

Each shape has a pointer to a `DShapeModel` object that does not do any drawing, but stores the coordinate information for one shape. Define a `DShapeModel` superclass for the model classes, with subclasses `DRectModel` `DOvalModel`, `DLineModel`, and `DTextModel`.

`DShapeModel` superclass should store a conceptual "bounds" rectangle defined by 4 ints: x, y, width, and height, and a Color.



Basic DRect and Doval

Our suggestion is to get basic things working first with just `DRect` and `DOval`, and then add `DLine` and `DText` later.

Define `DRect`, `DOval`, and their model classes. All the classes should have a "default" zero-argument constructor which puts the object in starting, "empty" state. Whoever is using the model can call its setter methods to install its proper data. The models should start out with all the ints at zero, and the color at `Color.GRAY`.

The shape objects should not contain any `int` coordinates or colors. Instead, each shape should have a pointer to a `DShapeModel` that can provide the data whenever necessary. The `DShapeModel` will need getters and setters for the bounds rectangle and color.

It's convenient to use Java's built in classes `Rectangle` (x,y,width,height), and `Point` (x,y). Your methods can create and pass around these objects to communicate. When changing the data inside a `Point` or `Rectangle`, make sure that change is not going to mess up some other part of the program which is using that same object. To avoid that problem, it's fine to make or pass a copy of the `Point` or `Rectangle`.

Each shape should have a `draw(Graphics g)` method where it draws itself. The canvas can loop through all the shapes, sending each the `draw()` message.

Create an `addShape(DShapeModel)` bottleneck method in the canvas that, given a correctly filled in `DShapeModel`, creates and sets up a shape with that model in the canvas. The Add shape buttons, can possibly indirectly, call through to the `addShape()` method. Use `instanceof` on the passed in model to figure out what sort of `DShape` to create (this is one of those rare cases where `instanceof` is the simplest solution.) Later on, the file reading code can use this same `addShape()` bottleneck method to populate the canvas with the models from a file.

Milestone

Create the *Add Rect* and *Add Oval* buttons in the frame. Wire the buttons up to create new models at random sizes and positions, and add them to the canvas.

Selection

The canvas should have a pointer of a single `selected` shape at any one time. The `selected` shape should draw differently -- for now just draw a little "x" or something on the `selected` shape. When the mouse is clicked on the canvas, figure out if the click was on a shape, and set it to be the `selected` shape. A click on a spot where two or more shapes overlap should select the topmost shape. It's handy if new shapes are selected, but it's not a requirement.

You will need a method like `shape.getBounds()` to ask each shape its rectangle boundary. Internally, the shape does not know the rectangle, so it has to turn around and call something like `model.getBounds()`. Determining the selection gets data from the model, but does not change the model. The idea of the selection is a feature of the canvas (view), not the model.

Change and MVC Listeners

`Whiteboard` should use MVC to handle changes in the model objects. Each model will tell its shape about any changes that have happened in the data. We'll use the interface `ModelListener`. Any object that wants to hear about changes in a shape model should implement `ModelListener` and the `modelChanged()` method.

```
// ModelListener.java
/*
 Interface to listen for shape change notifications.
 The modelChanged() notification includes a pointer to the
 model that changed. There is not detail about
 what the exact change was.
*/
public interface ModelListener {
    public void modelChanged(DShapeModel model);
}
```

Enhance the `DShapeModel` so that it keeps a list of listeners, and provide methods for the listeners to be able to add and remove themselves from the list. When the model changes, in a setter like `setColor()`, it should loop through its listeners, and send the `modelChanged()` notification to each listener.

When a shape is set to use a model, the shape should register as a listener to that model. When the shape gets the `modelChanged()` notification, the shape needs to re-draw itself on the canvas with the changed appearance. `ModelListener` should deal with changes to an individual shape data, like changing its size or color. `ModelListener` does not deal with larger scale changes about a shape, like adding or removing a shape from the canvas. We're only using `ModelListener` at the small scale of tracking changes on a shape.

SetColor

Initially, all the shapes just draw as gray. Add a *Set Color* button in the window. Use the Swing class `JColorChooser` to let the user select a color. When the user clicks the *Set Color* button, bring up a color chooser with the current color of the selected shape. If no shape is selected, then just do nothing. If the user clicks the ok button on the color chooser, set the selected shape to be that color. When everything is working right, the `setColor()` message on the model should automatically trigger a notification that ultimately gets the shape to draw in the new color. The color operation is simple, but it tests the setter-model-view setter notification chain.

Moving

Clicking a shape selects it, and then dragging with the mouse button down should move that shape until the mouse button is released. The mouse drag code should send some sort of setter message to the model of the selected shape which will in turn notify the shape. It's fine if the user moves a shape past the boundary at the edge of the canvas -- we'll let them do that. They can also move a shape to a place where it disappears underneath some other, non-moving shape. That's fine.

Knob Drawing

To work like a real draw program, the currently selected shape should draw with "knobs" at its four corners. The knobs should be handled by the shape and canvas classes, since the knobs are part of the drawing presentation, not part of the real model data. We'll implement the knobs in a slightly general way, so it will work for `DLine` and `DText` later on. Implement a `getKnobs()` method in `DShape` that returns a list of four `Point` objects, with each x,y point being the pixel just inside the four corners of the shape bounds rectangle.

Modify the canvas/shape draw code so the selected shape has black squares 9 pixels (define in a constant) on a side, centered above each knob point, so a selected `DRect` or `DOval` look like the following (of course in reality, only one would be selected at a time). For simplicity, the knobs can draw in the same front-back layer as the shape itself, covered by the same things that cover the shape.

Knob Resizing

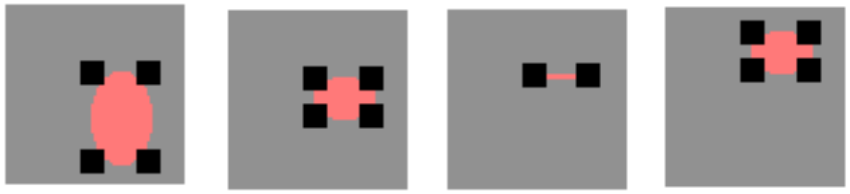
If a click-drag is on a knob, we want to resize that shape. Modify the click detection code so that it can notice if a click falls on a knob of a shape. A click on one of the knobs of a shape initiates a resize of that shape. Otherwise, a click within the bounds of the shape should initiate a move of that shape. Any click within the bounds rectangle will count to select that shape -- we won't worry about the empty space for a `DOval` where the drawn oval falls inside the bounds.

Here is an algorithm for knob resize that works well:

- The initial click is on some knob/point -- call that the "moving" point.
- From the list of knobs, find the point in the opposite corner from the moving point. Call that the "anchor" point, and remember it for the duration of the mouse drag.

- During the drag, update the moving point while the anchor point does not move. Now consider the rectangle defined by anchor point in its original location and the moving point wherever it is now. That rectangle should be the new bounds rectangle for the shape.

This algorithm works even if we grab, say, the lower right knob and drag it straight up so it appears to flip the oval like this…



The resize logic is tricky, but ultimately it should come down to some setter calls sent to the model to change the bounds rectangle.

Note that the existence of the knobs does not make the bounds rectangle in the model any bigger. The bounds is still just the bounds of the drawn shape. However, the shape now also has different, bigger bounds that includes the shape and also its knobs. You may want to add a `getBigBounds()` convenience method on shape that returns the bigger bounds for clients who need it.

Delete Shape, Add Buttons

Add a *Delete Shape* button which deletes the selected shape, or does nothing if there is no selected shape. Change the *Add Shape* button so it creates every new shape at x=10, y=10, with width=20, height=20. As a general rule in complex MVC apps, when you delete an object, it may also be appropriate to delete attendant objects (models, listeners) that have a 1-to-1 relationship with the deleted object.

Front/Back

Add *Move Front/Move Back* buttons that re-arrange the shape front-back order. The front/back order of the shapes is just a function of their order in the canvas shapes list. Whatever is last in the list appears to be frontmost. So the front/back controls can just move the selected shape to one end or the other of the shapes list.

Milestone

You should be able to add and remove shapes, select them, adjust their front/back order, click within their bounds to move, click on their knobs to resize.

DLine

Now add The `DLine` shape and `DLineModel` classes to the setup. The `DLine` is significantly different from the other, basically rectangular shapes. The `DLine` is not defined by a bounds rectangle. Instead, `DLine` is defined by two points, which we will just call `p1` and `p2`. The `DLineModel` class should subclass `DShapeModel`, and contain two points. There is no special ordering between `p1` and `p2` -- as the user moves and resizes the line, either `p1` or `p2` may be the leftmost, topmost and so on. `DLine` and `DLineModel` should override methods used for moving, resizing, etc. to provide `DLine` specific versions. For example, the The `DLine` should have 2 knobs, not 4. For changes during a move or re-size code, `DLine` should apply those changes to the `p1`/`p2` data that the line has. The idea is that the canvas can just treat all the shapes as `DShapes`, just sending them messages to move and resize. The `DLine` uses overriding so it works with the same messages from the canvas as `DRect` and `DOval`. Here is a selected `DLine`…

The `DLineModel` still maintains a bounds rectangle (from its superclass). Whenever `p1` or `p2` changes, `DLineModel` should adjust its bounds rectangle to just include the two points. So for the above drawing, the drawn line runs along the upper-left to lower-right diagonal of the conceptual bounds rectangle, which is not drawn. We will not support `setBounds()` as a way to change `p1`/`p2` (there is no totally rational way to do it). If code wants to move the points, then it should call setters on `p1`/`p2` directly, and then the bounds will adjust to fit the points.

The move code can still use the bounds of the `DLine` to detect clicks, just like the other shapes. So you can click anywhere in the `DLine` bounds to move it, not necessarily near the actual line pixels. This is not ideal, but we have enough other problems to solve. (optional) The fix for this is to override the notion of click detection for `DLine` to do some arithmetic to compute if the click x,y is within a few pixels of the actual line.

DText

The `DText` is basicaly rectangular, like `DRect` and `DOval`, but with String text and String font name in its model. The text of a new `DText` should be "Hello" and the font should be "Dialog".

- The text shape will display inside a bounding rectangle, just like the other shapes.
- There is a `JTextField` that allows the user to see and edit the string used by the selected text shape. Use the `setMaximumSize()` feature of `JComponent` to prevent the text field from getting enormous as the frame is resized. Entering text in the text field should set the text of the selected shape, if it is a text shape.
- There is a `JComboBox` to control the font. The combo box should list all the available font family names. See the docs for the `JComboBox`, `Font`, and `GraphicsEnvironment` classes. Changing the selection in the font control should change the font name of the selected shape, if it is a text shape.
- The font size used will be a function of the height of the shape. The font will be just large enough so that the letters are tall enough to fill the shape height. The width of the shape rectangle is only used for clipping -- whatever part of the string extends past the width will not be drawn (see clipping below).

It turns out that the most convenient place to compute the `Font` is in `paintComponent()`. (You'll need to dig around in the docs for the `Font` class to see how it works.) Write a utility method called `computeFont()` that is called by the `DText` draw code, and uses its `Graphics` object. `computeFont()` should use the following strategy...

- The `Font` objects should be determined by the shape font name. We will not worry about the case where the font name does not match a font on our system, although a good default behavior in that case would be to substitute in the Dialog font.
- Start with `double size = 1.0;`. Get a `Font` object using the (`int`) of that size.
- Get the `FontMetrics` for that `Font`. Check to see if the height of the `Font` fits within the shape rectangle height. (Dig around in the Font docs.)
- (looping) If the `Font` does fit, then try an approximately 10% bigger size with adjustment `size = (size*1.10)+1;`. Check to see if that `Font` size fits. Continue doing this until a `Font` size **does not** fit. Use the `Font` size from the previous iteration. (The +1 in the equation helps its behavior when the font size is small.)
- (optional) Computing the font size every time we draw the shape is costly, especially since many times the height is not even changing. Add code to cache the result of the most recent font computation for a shape, so that if its height has not changed, it knows what

font size to use without recomputing it. This optimization can make dragging a text shape look more smooth.

The draw shape code should call `computeFont()` to get a `Font` object. Then set the graphics object to use that font and draw the current string at the bounds left and font "descent" pixels up from the bounds bottom. Usually we try to avoid doing heavy computation in `paintComponent()`, but it's really the best approach available since we need a `Graphics` object context to do `Font` computations at all.

There is a problem that the text can easily draw outside of the shape bounds rect. The fix is to manipulate the clipping rectangle of the graphics when drawing the text string. The "clip" is a boundary property of the Graphics object that limits draw operations to only change pixels that are within the clip. We want to temporarily change the clip to be the intersection of the old clip and the text shape bounds, then draw the text, then restore the old clip. Here's the code for that case:

```
// Get the current clip
Shape clip = g.getClip();

// Intersect the clip with the text shape bounds.
// i.e. we won't lay down any pixels that fall outside our bounds
g.setClip(clip.getBounds().createIntersection(getBounds()));

  -- draw the text --

// Restore the old clip
g.setClip(clip);
```

This will cut off any pixels that fall outside the shape bounds rect. This technique is only needed for the text shape. The other shapes intrinsically draw within their bounds.

Text Inspector

The text controls -- the text field and font combo box -- should set the state of the selected text shape. However, going the other direction should also work. As the selection changes, the text controls should change to show the state of the currently selected text shape. If the currently selected shape is not a text shape or there is not selection, the text controls should disable (`setEnabled(false)`) which will give them a grayed-out appearance. This is sometimes called the "inspector" paradigm -- that controls instantly switch to show the state of whatever is selected.

Milestone

You should be able to create line and text shapes, move and resize them. The text controls should work and should synchronize with the current selection.

Table

Finally, to show off the flexible data-handling of MVC, create a `JTable` at the bottom left that shows the x/y/width/height bounds of all the shapes in the canvas. Use `setPreferredSize()` on the table scroller if it's taking up too much space on screen. Create a subclass of `AbstractTableModel` that responds to `getValueAt()` using the adapter pattern -- pull the values dynamically out of the shape models. The table should show the shapes in the same order that the canvas has them, so the last shape in the canvas list will be the last row in the table. Adding and removing shapes should add and remove rows in the table. Moving and resizing shapes should change the analogous ints in the table.

How can the table model know when a shape changes? The table model should register as a listener to every shape model, and so get `modelChanged()` notifications. When a shape model changes, the table model should figure out the corresponding row in the table, and call `fireTableRowsUpdated(rowNum, rowNum)` which re-draws a single row in the table. The code that adds and removes shapes will need to send some sort of add/remove message to the table model so it can know to start or stop listening to that model. When shapes are added or removed, the table model can just `fireTableDataChanged();` which tells the `JTable` to refigure all the rows. Refiguring all the rows is overkill, but it's simple and good enough for the

add/remove case. We want to be fast and specific (`fireTableRowsUpdated()`) for the one-shape-changing case, since that one that happens **continuously** during a mouse drag.

Hints and Suggestions

Here are some other general ideas about the Whiteboard code…

- Our one design requirement is that the `DShape` classes do not store the model data. Instead, the `DShape` has a pointer to a `DShapeModel` object, and it stores the data. The canvas and the shapes should be concerned with drawing, selection, and the knobs.

- It's fine to add methods to the canvas, shape, and model classes as your design grows to support all the features. For example, it's fine to add setters to the model for the convenience of the other classes, such as a `setBounds(Rectangle)` and a `moveBy(dx, dy)`. Under the hood, you can simplify things by having the convenient method `moveBy()` just call `setBounds()`.

- Performance is mostly not going to be a problem, so aim for a design that is clean and correct first, without worrying too much about performance. The one exception is that we do care about performance during move and resize animation. Therefore, canvas `paintComponent()` should be as straightforward as possible -- avoid extraneous computation and just draw all the shapes.

Part A represents a little more than half the work. Part B layers on some neat data-handling features that build on the solid MVC design of part A.

Part B -- File Saving and Networking

Now we turn to the "advanced" features of the Whiteboard. Before working on these, you want to have core MVC-mouse-draw part of the code totally cleaned up and debugged, since the advanced features build on the basic MVC core. This handout presents the file saving first, then the network operations, although in reality you could add the features in either order. There is no additional starter code for these features, but feel free to get code from the relevant lecture examples (the Java code for the lecture examples is available in the hw directory).

File Save/Open

Add support for very basic file save/open support with two buttons -- *Save* and *Open*. This part of the code should be relatively easy, as it builds on the earlier parts. Save should prompt the user for a filename and write the current model to it. Save should work if the whiteboard is in normal, client, or server mode. The Open button should prompt the user to select file, clear out any existing state, and read in the state from the file. Open only needs to work with the Whiteboard in normal mode, not client or server mode. We are not implementing the dirty bit or Save As. To implement save, build a temporary `DShapeModel[]` array, and then use Java's built in XML encode machinery to write it out. To read a file, use the XML decoder to re-create the array, and then run those models through the add-shape bottleneck to populate the canvas. The shapes when read in should appear to have the same back-front order they had when saved out. See the lecture save/open example, which demonstrates exactly this strategy -- you may need to adjust the getters/setter public interface of `DShapeModel` a little so the XML machinery can use your model properly. After saving a file, open it up in a text editor to see that the object is being written out the way you intend. Finally, add a *Save Image* button that prompts the user for a filename and saves the current appearance of the canvas to a PNG file (see the lecture example). Create some sort of drawing in your program and save it as "art-yourname.png" which is one of the deliverables. Detail: don't draw the knobs in the saved image of the canvas. Optionally, invest a little artistic effort in your picture to expresses your feelings about Java, OOP, or something else -- if time allows, we'll have a little in class art show towards the end of the quarter with the most interesting portraits.

Networking

For the networking feature, the goal is to build in simple client/server networking support, so that any number of clients can observe the operation of a single server Whiteboard. The clients are read-only -- they just see the Whiteboard state, they can't do any of their own edits. (Making the communication two-way is possible with our architecture, it's just more work.) Our networking support will be pretty basic -- it's not going to have every feature of a real networked whiteboard. We just want it to work in a simple way, enough to work through the ideas of networked data, change propagation, etc.

Server Mode

There should be a "*Server Start*" button that tries to start a server thread. The server should prompt for a port-number to use, but provide a default, so the user can just hit return to take the default. Fix some large arbitrary value for your default port, like 39587, so that port has a good chance of being available. The user can click the Server Start button once to start the server, and that should set a little status string to the right of the server buttons to "Server mode". The user can click the button a second time, but it will fail since the first server is using the port. We won't bother with enabling/disabling the buttons to protect the user from that sort of thing, and there is not way to get out of server mode. Our networking implementation will be functional to play with the ideas, but bare-bones. A Whiteboard can go into either server mode or client mode, but not both. There is not a way to get out of client or server mode other than quitting the program. The server will hold the single "canonical" version of the model data. All the normal shape edit operations -- shape add/delete, set color, move, and resize -- should continue to work when in server mode. They edit the data model, just like before.

Client Mode

There should be a "Client Start" button that prompts for a client connection ("host:port" syntax), with "127.0.0.1:port" as the default, so the user can just hit return to start in the default way. This should try to start a client connection and set the status string to "Client mode". In Client Mode, the operation of the program is quite different. None of the edit operations -- add/delete, set color, move, resize -- should work. They should just do nothing. Instead, the client just synchronizes its model to show current model of the server. Selection can still work, but it's pretty meaningless without the ability to make any change to the selected shape.

Networking Strategy

The server should keep a list of client connections, and the idea is that the server will send changes to the data model to all those clients, to keep them up to date.

The strategy for sending information from the server to the client will be focused entirely on sending copies of `DShapeModel` objects from the server to the client. Here is the strategy: There are just a few changes that can happen to the data model on the server side -- shape add/delete, front/back, set color, move, resize. For any one of those changes, consider the line just after the change to the model. Just after the change to the model, we want to send a notification to all the clients. A simple and effective scheme is to send two things for each change: a command string, and the relevant `DShapeModel`. The command string should be one of "add", "remove", "front", "back", or "change". The command string can be sent directly using `writeObject()` (Strings are `Serializable`, so it just works). To send the state of the `DShapeModel`, we can just re-use the existing logic that writes DShapeModels to a file. Rather than writing to a file, Use `XMLEncoder` to write the `DShapeModel` into an XML string. Having converted the model into a string, the string can just be sent using `writeObject()`.The reading code just reads the command string, followed by the string containing the xml encoded version of the model:

```
    String verb = (String) in.readObject();       // Read command string, e.g. "add"
    String xmlString = (String) in.readObject();  // Read xml encoding of model
// Now use XMLDecoder to extract the DShapeModel from the xmlString
```

See the lecture example of using xml encoded objects with sockets.

For each pair, the client can take the appropriate action depending on the command:

"add" -- create a new shape using the passed model. Ideally, this can ultimately go through some common bottleneck add-new-shape-given-model method that is also used by the Add Shape buttons when not in client mode. The given model will have all its data filled in -- id, color, bounds, and p1/p2 if a line, text/font if a text shape. Just use all that data as given.

"remove" -- remove the shape corresponding to the given model -- use the "id" to identify which shape+model to remove. There should be a model with that id to remove, and if there is not, print a debugging error message. (That error should never happen if the server is sending correct data so the server and client models are all the same.)

"front"/"back" -- similar to add/remove. Make the change to the existing shape data structure on the client, finding it by id.

"change" -- catch all category for all changes to an existing shape model -- move, resize, change color, change font. In this case, the passed model should match up with an existing model by id (or print a debugging message if it does not). Implement a `mimic(DShapeModel other)` method in `DShapeModel` to make the receiver model take on all the attributes (color, rectangle, p1/p2 for a line) of the passed model. So if `current` is the model currently used by a shape on the client canvas, and `passed` is the model just passed to the client from the server, then the client does a `current.mimic(passed);`. This should change the client shape model to take on the characteristics sent by the server. `DLineModel` and `DTextModel` will need some overrding to `mimic()` all their data. Overriding detail: in the `DLineModel` and `DTextModel` subclasses, the prototype of the mimic method should still take a `DShapeModel` argument, like this: `void mimic(DShapeModel other)`. Remember that for overriding to work, the method name and arguments need to look exactly the same between the superclass and subclass. All the new data we need is in the passed model, so we just use it all. What's nice about this strategy is that all those edits -- moves, resizes, color changes, font changes -- are all made to work by the one `mimic(passed)` bottleneck. The changes to the current model should trigger `modelChanged()` normally on the client side, and so the client on-screen shapes and table should update automatically.

When the server first gets a client connection, it should send a one-time series of "add" commands to populate that client with all of the current server shape models. In the server code, the places where a change happens -- add/remove, front/back, move, resize, etc. -- the server needs a bit of code to send that change to all the connected clients. For the add/remove front/back cases, you will need to insert a little code to send an update with the right command and model. All the other cases can be captured with very little work -- just use `ModelListener`. Have some object on the server side (could be the canvas or some inner class) listen for changes on all the current shape models. Whenever a model changes, send a "change" notification to all the clients. That will take care of all move/resize/color/font changes to existing shapes.

So overall, our networking strategy makes heavy use of our existing MVC and data model abstractions. To propagate changes on the network, we just send around shape models. Using xml to write the model data is not the most efficient, but it's great code re-use. It's nice that having gotten xml debugged and working for the file save, case, we now re-use that same code for networking. If efficiency turned out to be a problem, we could switch to using a more packed representation at the socket level, but none of the upper layers would need to change.

Networking Suggestions

- Place the method that writes a command+model on a socket right next to the method that reads a command+model from a socket. These two methods must be consistent with each other, so it's nice to be able to see them at the same time.

- As you debug the propagation of changes from the server, across the net, and through the client, it's natural to want debugging output to track an individual change. You may end

up debugging messages like "sent model", "got model", "applied model change" -- you may want to implement a debugging `toString()` on `DShapeModel` so your debugging messages look better.

- The server should assign each shape model a unique id number, and everyone else should just use that id scheme. It's fine to just have a serial id number that the server just increments for each added shape model.

- For the networking, we are not dealing with errors or exceptions in a robust way -- just print something and then muddle forward in whatever way is easiest.

- To debug the networking, you really want to have two Whiteboard frames up -- one to be the server and one to be a client. During development, it's fine to change `main()` to create two `Whiteboard` objects. Even though they are running in the same JVM, they can still network to each other. In general, when you click the *run* button, you want to be able to get to your debugging setup with just a few steps, and it's fine to hack the code in some way during development to facilitate that (this is also why the client and server buttons give defaults that work for debugging so you can just hit return). Turn the project in with `main()` just creating one frame

- Strategy we are not using: another strategy would be to connect the `XMLEncoder` directly to the socket stream. In some ways, that would work great. However, this conflicts with the encoder's strategy of not sending the same object twice. The encoder would "optimize" the second time we sent a model and send nothing. It does not realize that we've changed the model, and so really want to send it a second time.

Deliverables

Congratulations -- you have now built a primitive but functional program with a real MVC core, drawing, mouse gestures, file saving, and a little networking. In normal mode, we should be able to create and delete shapes of all types, select shapes, change colors, text, font, and move and resize with the mouse and see the little table update. For networking, we should be able to put one `Whiteboard` in server mode, and connect one or more client whiteboards with the clients running on the same machine as the server, or different machines. We should be able to do add/remove/front/back/change on the server side, and see all those changes reflected on the clients. The appearance of the animation on the client side may be a bit chunky. Finally, we should be able to save a file, and in normal mode and open an existing file. Congratulations -- that's some program!