

# Comp 401 - Assignment 5: Object and Shape Composition

---

**Date Assigned: Thu Sep 17, 2015, 11:55 PM**

**Completion Date: Fri Sep 25, 2015, 11:55 PM**

**Early Submission Date: Wed Sep 23, 2015, 11:55PM**

In this assignment you will incrementally create an initial bridge scene. You will define some image shapes and then compose them with a line shape (which you can use from the extra credit part of the last assignment or praxis/lecture/recitation exercises) to create avatars, which will be composed into the initial bridge scene. Most of your time will go into the creation of the avatar.

If you have not done so, you may want to also look at my demos at the start of class for what an avatar could look like and where we are going:

[Bridge Scene - 1st day \(long\)](#)

[Bridge Scene - 2nd day \(short\)](#)

Images of Avatars

[images.zip](#)

User Interfaces	<a href="#">PowerPoint</a> <a href="#">PDF</a> <a href="#">YouTube</a>	<a href="#">Docx</a> <a href="#">PDF</a> <a href="#">Drive</a>		<a href="#">Commands and Graphics</a> <a href="#">UNCChecks</a>	<a href="#">lectures</a> <a href="#">lect</a>
Composite Objects	<a href="#">PowerPoint</a> <a href="#">PDF</a> <a href="#">YouTube</a> <a href="#">Mix</a>	<a href="#">Docx</a> <a href="#">PDF</a> <a href="#">Drive</a>		<a href="#">Composites</a>	<a href="#">lectures.com</a>
Composite: Tree, DAGS & Graphs of Objects and	<a href="#">PowerPoint</a> <a href="#">PDF</a> <a href="#">YouTube</a>	<a href="#">Docx</a> <a href="#">PDF</a> <a href="#">Drive</a>		-	<a href="#">lectures.composite.tre</a>

Windows	<a href="#">Mix</a>				
---------	---------------------	--	--	--	--

## *Packages*

Divide the classes you have into at least three packages making sure that only main classes are in the main package. How exactly you do the division is upto you, but the classes in the same package should share one or more properties not shared with classes in other packages. This constraint should be met in this and future assignments. Recall that any import that does not start with mp or grail will be classified as a foreign import and will be considered illegal unless explicitly allowed. So make sure your non-main packages are children of mp or grail, as mentioned several times.

## *Line, Image and String Class(es)*

Implement one or more classes that can be instantiated to create objects that ObjectEditor displays in the graphics window as:

1. the head of Arthur
2. the head of Lancelot
3. the head of Robin
4. the head of Galahad
5. the head of the guard.
6. a string shape with both a getter and setter method
7. a (possibly) rotating line

I have found that the easiest (for me) way to create an image file on a Windows machine is to display the image on your screen, use the Snipping Tool to copy a portion of it, paste the copied part to a PPT slide, use PPT to adjust the size of the image, and finally, use the Save As Picture right menu PPT command to save the image to a file. We are also providing you with files I used for the images ([images.zip](#)).

If you will use short file names (without “/”) the image files should be in the project folder, the one containing bin and src.

## *Angle/V Shape Class*

Implement a composite shape class that can be instantiated to create an object that ObjectEditor displays as an “angle shape” or a “V shape”. Such a shape consists of two (possibly rotating moving) readonly line properties, named LeftLine and RightLine, whose upper left corner is always at the same location, which is also considered the location of the angle. In

addition to the getter methods for the two lines in the angle, define a method to move the location of the angle by an int x and y offset, which should be given in that order in the call to the method. Tag this method as “move”. The class should define a constructor that takes no arguments. It can define other constructors.

This is a composite shape and is not to be confused with the angle property. This shape will ultimately represent the arms and legs of an avatar. (This is the reason for creating a rotating line in the previous method. The shape does not require a rotate method. So you can rotate a line by getting it from the angle shape and rotating it, as in:

```
angle.getLeftLine().rotate(5)
```

Of course this discussion is only for the ones who did the extra credit.)

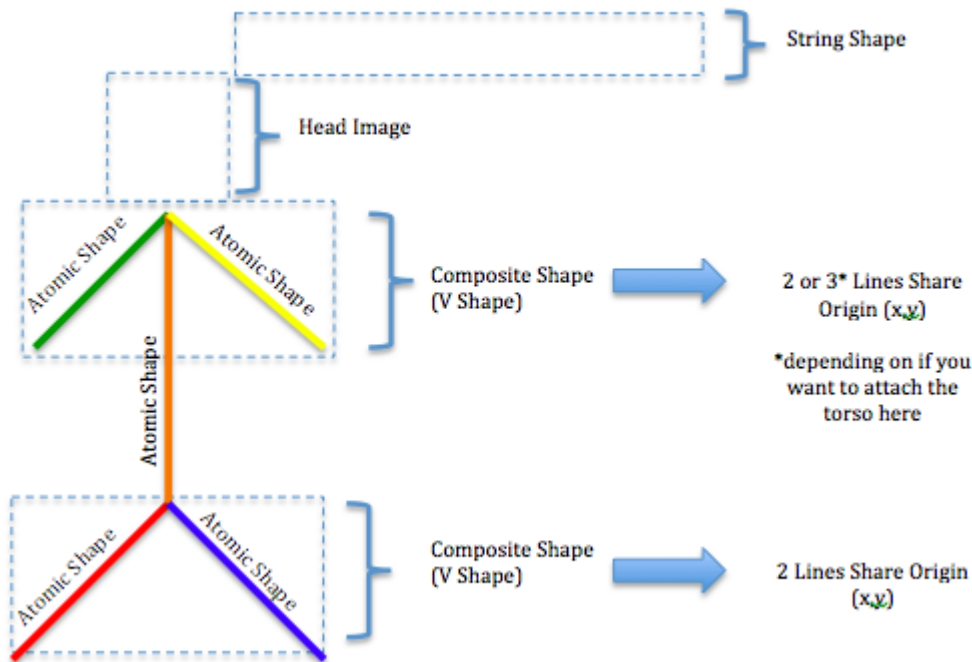
*Tag this class as “Angle”.*

## ***Avatar***

Create an avatar (by defining appropriate classes and interfaces) composed of *the following* shape properties recognized by ObjectEditor whose names are given before the colon.

1. Text (Editable): One of them is a String shape representing the utterance “spoken” by the avatar.
2. Head (Readonly): Another is an image shape representing its head.
3. Arms and Legs (Readonly): The two of them are angle objects, which represent the arms and legs of the avatar.
4. Other body parts (Readonly): Finally, you should have *one or more* additional shapes representing the other body parts of the shape. You are free to choose the names of these properties and the type – the only constraint is that the logical structure of each of these properties contain at least one object recognized by ObjectEditor as a shape.

These components should connect at the avatar joints. Here is a great graphic from Sean Jarecki (from a previous class) that gives one way to do the avatar.



In my implementation (which is in the video) I also had a neck, which of course is optional

The class of the complete avatar should define a constructors that takes as an argument the head shape. You are free to define other constructors. You are free to customize the avatar in other ways such as allowing its color to be specified by the user of the class.

The avatar should define a method to move it. Like the angle, the method takes two int parameters, giving the (positive or negative) distance (in pixels) to be moved in the X and Y directions respectively, declared in this order. Again, tag this method as “move”;

When an avatar moves, the relative distances between its components remain fixed. Thus, each component should move by the same amount. To implement this method, you may want to define a particular point in the avatar (such as the upper left corner of its head or the point at which the neck meets the head) as its location. The code that properly connects the body parts should not be duplicated. Many of you will have code in your avatar class that explicitly connects the body parts, though there are ways to avoid writing such code. If you do, you should write a single method that, given an avatar position, connects the body parts. This method should be called by all methods (including constructors) that initialize or move the position.

To properly position the head, you will need to know its height or its width. See the class `AnImageWithHeight` in the composition PPT to see how the size of an image in a file can be determined.

As mentioned above, given that the arms and legs are angles, without doing extra work, you should be able to rotate each arm and leg while keeping the avatar connected. You do not have to define extra methods in an avatar to rotate the limbs (you can simply get one of these lines and rotate it) as in:

```
avatar.getArms().getLeftLine().rotate(3)
```

However, you are free to define these and additional methods in the avatar.

Tag this class as “Avatar”.

### ***Part 3: Bridge Scene***

Create another composite shape object consisting of five instances of the avatar class defined above, which represent the five characters of interest: Arthur, Lancelot, Robin, Galahad, and Guard. These avatars must have different heads – you are free to add other idiosyncrasies. For example you can color them differently. Each avatar will be a separate property in the scene object. We will add additional objects to the scene later such as the bridge, for now this much seems enough.

Tag this class as “BridgeScene”. This class should have a readonly property for each of the five characters, whose name is the name of the character.

### ***Animating Demoing Main Class***

To demonstrate your work, write a main class that instantiates the bridge scene, displays it in an ObjectEditor window, and animates the movement and (for extra credit) arm rotation of at least one avatar in the scene. (It is sufficient to rotate just one arm). As before, an animation is done by using the refresh and sleep calls. For fun and preparation for future assignments (but not extra credit) make the avatars march and speak in your animations.

### ***Extra Credit***

1. Scalable avatar: allow each avatar to be scaled, that is, define a method taking a single int argument (tagged “scale”) to change by the same proportion the size of each scalable component (line, rectangle, oval) of the avatar while keeping the avatar connected. You do not have to worry about changing the size of the head or the text spoken by the avatar, though you would have to move the text to account for avatar growth and shrinkage.
2. Colorable avatar: allow coloring of the avatar, that is, define one or more methods to change the color of the each colorable component (line, rectangle, oval, text) of the avatar. See the previous assignment on how color is specified to ObjectEditor. No extra credit points will be given for this.

### ***Constraints***

1. If you know inheritance, feel free to make use of it in this assignment.

2. As mentioned in class, in this and all other assignments, your getters and setters should not create objects. This means that you should not move, resize, color or change other aspects of a shape by replacing it with a new one. One exception to the rule is that you are free to create new immutable Point instances to represent the movement of a shape. Also, the setter in the scanner can create both the arrays.
3. Recall that in every assignment, every public method of an instantiated class must be in some interface implemented by that class. Do not use classes as types of variable or array elements – use interfaces instead. Follow these two constraints in all future assignments also, even if they are not explicitly stated.
4. Remember also to use the PropertyNames and EditablePropertyNames annotations (discussed in the User Interface lectures ) for all non-graphics Bean classes in this and future assignments. Please note that ObjectEditor puts spaces in the middle of property names to “beutify” the names. Please use the property names that in the code – the getters and setters – which do not have spaces in them. If you get them wrong, they will not be displayed by ObjectEditor.
5. Use the appropriate StructurePattern annotations for the graphics and other Bean classes in this and future assignments. You know you have used them correctly when you do not get any warnings from ObjectEditor telling you of missing structure pattern annotations.
6. Encapsulate all of your classes, as we discussed in lectures, in this and future assignments. This means should not make any non-final variable public.
7. Do not use any libraries I have not covered in class or authorized that make your task easier, such as ArrayList or Vector.
8. You can use any Math function in this and other assignments.
9. No errors other than refresh warnings from ObjectEditor.
10. Expected new tags: Angle, Avatar, BridgeScene
11. A new checkstyle jar file is linked from the home page for this assignment.

### ***Submission Instructions***

- These are the same as in the previous assignment. The TAs will run the main method to see the test cases animate
- Be sure to follow the conventions for the name and package of the main class.
- Good luck!

## *Optional ObjectEditor Features*

Here are some ObjectEditor features you could use to spruce up your drawing windows. If there is anything else you would like, post a message to Piazza, and we may be able to help you.

Graphics objects such as lines, ovals, and rectangles can have the following properties given special meaning by ObjectEditor:

**Filled:** This is a boolean property. If its value is true, then an oval or a rectangle graphics object is filled.

**Color:** This property is of type `java.awt.Color`. It can be set to any instance of this class. Some standard values are provided as constants, such as `Color.BLUE` and `Color.RED`. It specifies both the color of the outline of the shape and its filling, if the shape is filled. Thus, if a rectangle has a Color property that is `Color.BLUE` and a Filled property that is true, then both its outline and its interior will be blue.

**Stroke:** This is a property of type `java.awt.Stroke`. It can be assigned any value of this type. In particular you can assign it values of type `java.awt.BasicStroke`. For example, if this property has the value `new BasicStroke(8.0f)` you will get a very thick line.

**Font:** This property is of type `java.awt.Font`. It determines the font used in a String shape.

**Rounded:** This property is of type boolean and determines if the edges of a rectangle are rounded.

The above properties apply only to atomic shapes such as `Rectangle` and `Line`. If you store these values in a non atomic shape, do not make them visible to ObjectEditor, as some of these value are large or non Tree structures that it cannot handle. This means do not make these values visible properties. (You can use the `@Visible(false)` annotation to make a property invisible, as discussed in class).

You may want control over how properties of an object are ordered in both the main window and graphics window. In the latter the ordering is important when the positions of two objects overlap and you want one displayed over another.

Given an object you can order its properties by putting before the getter of a property the annotation:

`@Position(n)`

where `n` is the order of the property.

You need the import:

```
import util.annotations.Position
```

A property (and its children) that has a smaller position number is shown above the one with a larger one.

If you use this annotation for one property of an object, be sure to use it for all of the other properties of the object also, and use consecutive numbers, starting from 0, for the positions.

As mentioned in `StateProperties.PPTX`, you can also order properties using the `PropertyNames` annotation as in:

```
@PropertyNames ({“height”, “weight”, “bmi”})
```

Positions are given in left to right order, with the left most being the smallest. Do not use `@Position` if you use `@PropertyNames`.