Comp 401 - Assignment 4: Commands and Graphics

Date Assigned: Thu Sep 12, 2013
Completion Date: Fri Sep 20, 2013
Early Submission Date: Wed Sep 18, 2013

This assignment has two parts having to do with scanning and graphics.

In the scanning part, you will classify some of the words, not into vanilla word tokens, but instead into command tokens. This means you will implement several new command token classes. These classes, like the original token classes, will be instantiated by the scanner. In addition, the scanner will define an array property that contains an array of token objects. This part gives you more practice with concepts you have exercised in the previous assignment and should not take much time.

In the graphics part, you will define a rotating moving line, which might require some time-consuming debugging, though I pretty much give an algorithm for those who have trouble with the geometric aspects of it.

The main class will now animate (a) the scanner by assigning different values to the editable String property of the scanner while it is being displayed by ObjectEditor, and (b) the line by making it move and rotate.

You will use the Properties, EditableProperties and Tags annotations for all of the Bean classes. In addition, you will use the Tags annotation for the rotate line method. See the previous assignment for the tags to use for bean classes you have already implemented. In addition, you will use appropriate StructurePattern annotations for graphics and other Bean classes – ObjectEditor will scream if you do not put these.
**Tag Annotations**

This part was extra credit in the earlier assignment, it is required now.

Tag each of the scanner-bean, number-token, word-token, quoted-string token, start-token, end-token, plus-token, and minus-token classes using the tags: “Scanner Bean,” “Number Token”, “Word Token”, “Quote Token”, “Start Token”, “End Token”, “Plus Token”, “Minus Token”, respectively. It will be best to copy and paste these strings into your annotations to make sure you do not make errors. The quotes around these strings will probably not paste properly, so you may have to retype them.

The syntax for tagging a class was given in the lecture and is illustrated below for the scanner bean class.

```java
import util.annotations.Tags;
@Tags({"Scanner Bean")
public class <Class Name> implements ... {
...
}
```

As you see above, you need to import and add the annotation before the declaration of the tagged class. For this to work, you need have the latest ObjectEditor library (oeall22) linked to your project.

You should not tag the interfaces, and should tag only the classes. Two classes should not have the same tag.

**Command Classes**

Create a command token class for each of the following command names: “move”, “say”, “rotateLeftArm”, “rotateRightArm”, “repeat”, “define”, “call”, “thread”, “wait”, “proceedAll”, “sleep”, “undo”, “redo”. A command token class has the same properties and constructor(s) as the word token class, and implements the same interfaces. Thus, you will essentially copy and
paste the code of the word token class into each command class - the only difference between a command class and a word class will be the name. If you know inheritance, you can make use of it here to avoid this copying.

Tag each of these classes by the name of the associated command. Thus, put the following text before the class you define for the move command:

```java
@Tags(“move”)
public class <Class Name> ... {
    ...
}
```

Angle brackets denote placeholders.

### Scanner Bean Class

#### Instantiating Command Classes

Modify the setter method of the scanner to classify words into commands. After you have found a word (as before), check if its lowercase representation is equal to one of the command names, and if so, then create an instance of the associated command token class instead of the word class. Thus, if the scanned string contains the word “MoVE”, you would create an instance of the token class associated with the command “move” and not an instance of the word class, as in the previous assignment. You should use the String equals() (or equalsIgnoreCase()) method to test for equality of two strings. If a word is not one of the predefined commands, then it should be stored in an instance of the word token class, as in the last assignment.

**Additional array property**

The setter method of the scanner bean no longer print the properties of the token objects it instantiates. Instead it puts the tokens in a token array, first in a large array and then in a small compact, as mentioned below. This array is returned as a readonly property of the scanner bean.

More formally, the scanner now has an additional property, which is a readonly stored property of type T[], where T is the interface implemented by all tokens in the previous assignment (Please do not name this interface as T!). Let us call this property the tokens property. The getter method of this property returns an array of all token objects (instances of your token classes) created while scanning the String property of the scanner. There, should be no empty slots in the array, that is, the length of the array is the number of tokens in the editable scanner property storing the scanned string. You can assume a limit (e.g. 100) on the number of tokens in a scanned string. This means that you can create a large array (that accommodates the maximum number of tokens) whose elements are copied into the compact array returned by the getter method of the readonly property. The large array should be created once when the scanner is instantiated. The compact array would be created by the scanner setter method, each time a new string is scanned; it should not be created in the getter
You want to create a "large" array because you aren't sure how many tokens are going to be in the array, therefore you won't know the size of the array. Once the scanner has finished adding all the Token objects to the large array, you will be able to find how many have been added. You then will create a compact array that has the exact same token objects in the large array - it will just contain less elements because it won't have any extra room.

Here's an example:

**Input:** say "hello"

**Large array:** {word token, quote token, null, null, null, (however large you make it will have null for the rest of the elements)}

**Compact array:** {word token, quote token}

The large array is not accessible by anything outside the scanner bean class, which means there is no public method to return its value. The compact array is the one that is exported as a (readonly) property.

**Changed setter method**
The setter method of the scanner no longer print the properties of the token objects it instantiates. Instead it sets the value of the variable holding the token array.

**Scanner Extra Credit**

1. Instead of printing the errors on the console, store them in a third dependent readonly property of the scanner bean. The error property should be reset every time a new string is scanned. You are free to also print the log of errors on the console. You are free to choose the type of the error property – String, array...

**Rotating Line**

This part can be elegantly done in one step. I am breaking it into multiple steps in an attempt to make it easier.

**Rotating Fixed Line**

Create a class that implements a line shape that can be rotated around the Java origin (0, 0). The upper left corner of (the bounding box of the line) is always the Java origin. The lower-right corner of the line is always a fixed distance from the origin and can be rotated based on its current angle.

The line should be displayable by ObjectEditor. This means it must have the line properties and annotations expected by ObjectEditor. As the upper left corner is fixed, the line class does not
have setters for the location of this point. It also need not have setters for the height and width properties of a line. It should have public methods for setting the radius and angle of (the lower-right corner) of the line. These methods take double values determining the absolute radius and angle.

In addition, the class must have an instance method to change the angle of the line by a certain amount. This method must take an int argument. You are free to determine the appropriate scale. For example, you might decide that one int unit corresponds to Math.PI/32. In this case, rotating the line by 16 units adds 90 degrees (Math.PI/2) to its angle. This method must call the method for setting the angle mentioned above.

Let us call this method the rotate method. As we saw in class, you can tag both methods and classes. Use the tag “rotate” for this method. Thus, its declaration will be of the form:

```
@Tags("rotate")
public void <Method Name>(int units) {
...
}
```

Try to implement this class on your own before you read the remainder of this paragraph.

ObjectEditor does not understand radius and angle of a line, so you must rotate a line by changing its width and height, using trigonometry. One way to implement a rotateable line is given below. In addition to width and height, this line will have a radius (distance between endpoints) and angle (wrt to X axis). The constructor of the line takes these two values. The width and height will be derived from these values.

Declare an internal instance variable that stores the current lower-right corner (the end point) in an instance of the class APolarPoint we saw in lectures. This point always has the radius and angle of the line, It is the lower right corner only for this part – in the next part it is not. As we see below it is actually the storer of the width and height, This variable is not exported as a property to ensure ObjectEditor does not display it. The getter for the height and width property of the line _depend on the value assigned to this variable_return the x and y coordinates of this point, and the setters for the radius and angle of the line assign a new-immutable instance of APolarPoint to the internal variable, which, as mentioned above, has the radius and angle of the line. There is no need to define setters for its height and width - all ObjectEditor needs for displaying is the getters. There are other more elegant ways to implement a rotating line – so please use them if you can think of them. This approach requires you to do no calculations, and simply use the ones in APolarPoint.

Moving Rotating Line

Modify the class you defined above to allow the upper left corner of the line to be changed. This means you must now define setters for the line location. The line will now rotate around this corner rather than the Java origin. This means that if you have followed my implementation technique, the position of the internal lower-right corner polar point does not change when you change the upper left corner, as the. Neither does the length or width of the line do not change when the line moves, which should depend only on the coordinates of the “lower right corner”, which is no longer the lower right corner but instead the storer of the width and line of the box. You can do some trigonometry to figure out why this is right.

Do not submit the code you wrote for the fixed line – instead submit code for this line. The previous part was created to break your task into smaller steps.
Tagging the Line Class
Use the tag: “Rotating Line”

Animating Demoing Main Class
You can implement the main class in two stages.

Animating Scanner
To demonstrate the scanner part of your assignment, as before the main method creates an instance of the scanner class and assigns different values to the editable String property of the scanner. However, it no longer reads input from the console, **neither does it print on the console.** Instead it displays the scanner object using ObjectEditor and then assigns a series of test strings (which replace the input lines) to the editable property of the scanner. After each assignment, the method should refresh the ObjectEditor window and sleep so that the TAs can see the result of each assignment. We discussed this animation approach to demoing in class, and the following code illustrates it:

```
http://www.cs.unc.edu/~dewan/j2h/JavaTeaching/lectures/state_properties/ABMISpreadsheet AnimatingDemoer.java.html
```

Thus, your main method will directly make calls of the form:

```
scannerBean.setScannedString("MoVe 050 { saY "hi!" } ")
oeFrame.refresh();
ThreadSupport.sleep(3000);
scannerBean.setScannedString("RotateLeftArm 5 rotateLeftArm."")
oeFrame.refresh();
ThreadSupport.sleep(3000);
```

Instead of reading input lines from the console. It will make several calls for different test strings. You can use some of the same test strings as the ones you input in the previous assignment when you created screen shots. **A test string can contain multiple commands. Make sure every command is included in some test string.**

Animating Line
After animating the scanner, your main method should instantiate the rotating line, and display it using ObjectEditor. Next, it should animate the movement and rotation of the line. As before, make sure to call refresh and sleep so that the TAs can see the rotations and movements.

If ObjectEditor does not refresh properly, simply print appropriate properties of the scanner bean and bean-line after each sleep to show that your program works correctly. **Do not worry about the nature of the ObjectEditor display as long as it shows all of the properties. Do not worry about the error messages of the following kind regarding complete refresh:**
**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. **The only 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 an both the arrays, as mentioned above.**

3. You can use the String toLowerCase() method, as before. In addition you can use the String equalsIgnoreCase() method.

4. As also mentioned in earlier assignments, 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.

5. 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 yo get them wrong, they will not be displayed by ObjectEditor.**

6. 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.

7. 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.

8. Do not use any libraries I have not covered in class or authorized that make your task easier, such as ArrayList or Vector.

9. You can use any Math function in this and other assignments.

6.10. Make sure every command is included in some test string.
Submission Instructions

• These are the same as in the previous assignment except your document need not contain videos or screenshots. 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!

Controlling the width of TextFields created by ObjectEditor (For fun, not for credit)

You may find that the default width of the text fields created by ObjectEditor is not sufficient to display the values of the properties you define in this assignment. In oeall8.jar and later versions, you can associate the getter of a property with a ComponentWidth annotation, which takes the desired width (in pixels) of the text-field used to display the property, as shown below:

```java
@ComponentWidth (800)
public String getHeight() {
    return height;
}
```

The annotation above ensures that the “height” property is displayed in a text field whose width is 800 pixels.

You will need to import ComponentWidth as

```java
import util.annotations.ComponentWidth
```

depending on the version you use. The best approach is to let Eclipse tell you what the import should be. All annotations are in the util.annotation package.

In all versions of ObjectEditor (e.g. oeall3.jar), you can set the component width of all properties using the following call:

```java
ObjectEditor.setDefaultAttribute(AttributeNames.COMPONENT_WIDTH, 800);
```

In addition, you can set the component width of property P of class C to width by executing the method:

```java
ObjectEditor.setPropertyAttribute(C.class, P, AttributeNames.COMPONENT_WIDTH, width);
```

For example:

```java
ObjectEditor.setPropertyAttribute(ABMISpreadsheet.class, “height”, AttributeNames.COMPONENT_WIDTH, 800);
```
Some of you will probably make the error property an array and would want to set the component width of the elements of the array.

In general, in addition, you can set the component width of elements of array property P of class C to width by executing the method:

```
ObjectEditor.setPropertyAttribute(C.class, "P.*", AttributeNames.COMPONENT_WIDTH, width);
```

For example:

```
ObjectEditor.setPropertyAttribute(ClassWithArrayProperty.class, "ArrayProperty.*", AttributeNames.COMPONENT_WIDTH, 800);
```

makes the component width of the elements of the property ArrayProperty of class ClassWithArrayProperty as 800.

**Looking Ahead**

As always, please do not submit the looking ahead part.

**Image and String Class(es)**

Implement and tag 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

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.

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 rotating moving lines 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 one or more methods to move the location of the angle. 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 why both lines must rotate. The shape does not have a rotate method. So you rotate a line by getting it from the angle shape and rotating it. This means the lines can rotate independently.

Tag this class as “Angle”.

Avatar
Create an avatar (by defining appropriate classes and interfaces) composed of at least five atomic shapes recognized by ObjectEditor.

1. Text: One of them is a String shape representing the utterance “spoken” by the avatar.
2. Head: Another is an image shape representing its head.
3. Arms and legs: Two of them are angle objects, which represent the arms and legs of the avatar.
4. Other body parts: Finally, you should have one or more additional shapes representing the other body parts of the shape.

These components should connect at the avatar joints.

The class of the complete avatar should define constructors and/or setters for setting the head shape. 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 one or more methods to move it in the X and Y direction. When an avatar moves, the relative distances between its components remain fixed. Thus, each component should move by the same amount. You can define different methods for moving in the two directions or a single method, and these methods can specify how many units (positive or negative) the avatar should move and/or the absolute coordinates of the new position. To implement these methods, 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 next meets the head) as its location.

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) but are free to define these and additional methods in the avatar.

Tag this class as “Avatar”.