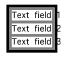
How to create a new figure with constraints

Often in *HotDraw,* one wants to create a new figure out of the building blocks of figures which have already been created. To group the component figures together, one needs to use a *CompositeFigure*. Usually some kind of spatial relationship is desired for the components in such figures. For example, a figure *CircleNline,* which looks like this: , consists of a circle and a line. The line must be constrained to run from the top left-hand corner to the bottom right-hand corner of the circle's bounding box. Another example is a box with several text fields, one above the other, like so: 

A *CompositeFigure* would move its component figures when the user moves it, but if the user resized it, it would not know what sort of relationships to mantain between the components. This is where *HotDraw-Constraints* come in. They can be used to constrain the different components of the *CompositeFigure* in the desired spatial relationship. Although this is not the only use of constraints in *HotDraw,* it is the major one, and so this pattern illustrates the use of constraints by using the example of the abovementioned simple composite figure *CircleNline*. (For an example of the use of constraints in another context, please refer to the sample *HotDraw* application *PertChart*.)

A subclass of *CompositeFigures* that has constraints requires an instance variable *constraint* since you will need to copy a figure's constraints when you copy the figure, and thus will need a way to refer to them.

The class needs a creation class method *createAt: aPoint* in the category *InstanceCreation,* which is called by the tool the user uses to make an instance of the figure.

The constraints need to be created when the figure is initialized. The component figures need to be created, and the relationships between them and the composite figure defined using constraints. To do this, write an instance method *initializeAt: aPoint* in the category *initialize*. This method

- creates the figures collection,
- creates the figures, putting them in variables, adding them to the figures collection. They are put in variables so that the constraints can refer to the figures later.
- creates the constraints collection.
- adds the constraints desired between the figures to the constraints collection.

The drawing needs to know what the new constraints are. To do this, provide the methods *at:, at: put:, constraintNamesDo:,* and *constraintsDo:* in the category *constraints*. (TO BE CHANGED BECAUSE THE SOFTWARE CHANGED)

Because copying is shallow, write the methods *postCopy* in the category *copy,* to copy multiple constraints.

The user uses the mouse to specify the size of the figure, so write the method *constrainCorner:* in the category *transforming* to do this.

Use the *ToolBuilder* to create a new tool that the user can use to create instances of the new figure.

Add the new tool to the *defaultTools*in the class method *defaults* in the class *DrawingEditor* so that the drawing editor will have the new tool in its palette.

Finally, in order that the tool would be filed out with the new class, save the tool from the tool browser into your class.

Thus, in detail, the steps you need to go through are:

1. To make a new composite figure with constraints, make a new class which is a subclass of *CompositeFigures* with instance variable *constraints*. This variable is needed since you will need to refer to the constraints when you copy them.

2. You will need a way to create an instance of the new composite figure, so create a class method *createAt: aPoint* in category *InstanceCreation*. The tool you will create later uses this to create a new instance at the location of *aPoint*.

```
createAt: aPoint
    "Create a new CircleNline at aPoint."
```

```
^self new initializeAt: aPoint
```

3. The contstraints need to be created when the figure is initialized. Make an instance method *initalizeAt: aPoint* in category *initialize*.This method
   a. creates the figures collection.
   b. creates the figures, and puts them in variables, then adds them to the figures collection. This is because the constraints need to refer to the figures later.
   c. creates the constraints collection.
   d. adds the constraints desired between the figures to the constraints collection. Always use the figure's *constraintClass* to create the constraints, because a figure gets its constraint class from its drawing. All the constraints in a drawing need to be of the same class, but different drawings will have different constraint classes. This makes it easy to switch from a centralized solver to a distributed one. There are many different kinds of constraints (see appendix A for a list) but for the example below, we illustrate with the most common ones, *equalityBetween: and:* and *equalityBetween: and: plusOffset:*

```
initializeAt: aPoint
        "Initialize to make origin at aPoint"

        | aCollection aLine aCircle |
        "aLine and aCircle are used by our figure."

        "Create the figures"
        aCollection := OrderedCollection new.
        aLine := LineFigure createAt: aPoint.
        aCircle := EllipseFigure createAt: aPoint.
        aCollection add: aLine; add: aCircle.
        self setFigures: aCollection visibleArea: (aPoint extent: 0 @ 0)

        "Create our constraints"
        constraints := OrderedCollection new;
                "keep line and circle together"
                add: (self constraintClass equalityBetween: aLine startPointVariable and: aCircle
                    originVariable);
                add: (self constraintClass equalityBetween: aLine stopPointVariable and: aCircle
                    cornerVariable);
                "keep the circle and the composite figure together. Needs the offsets to make
                    ellipse smaller than visible area."
                add: (self constraintClass equalityBetween: self cornerVariable and: aCircle
                    cornerVariable plusOffset: 1 @ 1);
                add: (self constraintClass equalityBetween: aCircle originVariable and: self
                    originVariable plusOffset: 1 @ 1)
```

4. Tell the drawing what constraints have been created by creating these methods in the category *constraints*. These methods are also used when an instance is being copied. (TO BE CHANGED BECAUSE THE SOFTWARE CHANGED)

```
at: aVariableName
        "To get the values for use to copy constraints origin and corner are handled by
            superclass"

        1 to: constraints size do:
                [:index | aVariableName = ('constraint' , index printString)
                        ifTrue: [^constraints at: index]].
        ^super at: aVariableName

at: aVariableName put: aValue
        "To assign the values for use to copy constraints origin and corner are handled by
            superclass"

        1 to: constraints size do:
                [:index | aVariableName = ('constraint' , index printString)
                        ifTrue: [^constraints at: index put: aValue]].
        ^super at: aVariableName put: aValue

constraintNamesDo: aBlock
        "To give names to the constraints origin and corner are handled by superclass"

        1 to: constraints size do:
                [:index | aBlock value: ('constraint' , index printString)].
        ^super constraintNamesDo: aBlock

constraintsDo: aBlock
        "Gives the constraints to the block"

        constraints do: aBlock.
        ^super constraintsDo: aBlock
```

5. Because copying is shallow, you need to create the *postCopy* method in category *copy,* to copy multiple constraints. See the pattern on *Copying*.

```
postCopy
        "Takes care of copying of constraints (because copy is shallow)"

        constraints := constraints copy.
```

6. When a figure is created or modified, the user will use the mouse to specify the size of the figure. So, write a *constrainCorner* method in category *transforming*.

```
constrainCorner: aMouseVariable
        "Move the bottom right corner of the figure to match the aMouseVariable"

        ^OrderedCollection with: (self constraintClass xyMouseConstraint: aMouseVariable on:
            corner)
```

7. You need a tool with with the user can make this new figure, so open a *ToolBuilder* (See the pattern on *Tools*) and:
    a. add a new tool called *CircleNlineTool*.
    b. add to the code in *DragReader* in the category *Figure:*

    ```
    FigureCreationCommand
            class: CircleNline
            creationMessage: #createAt:
            "This part makes the following drag be the size of the CircleNline."
            constraintMessage: #constrainCorner:
    ```

    c. make an icon for the tool and the cursor.

8. So that the drawing editor will have the tool in its pallete, add the new tool to the *defaultTools* in the class method *defaults* in the class *DrawingEditor*.

9. *Optional step for filing-out purposes to save the tool:* Save the tool from the tool browser into your class (menu item from the browser's tool selection window.) There is a bug that makes the cursor's icon's saving not work properly, so we have to manually replace that part with *Cursor normal* to make it use the normal default cursor (arrow.) (See the pattern on *Tools*.)

editOn: aVariable
tells the constraint system aVaraible can be changed from outside the constraint system.

equalityBetween: var1 and: var2
var1 = var2
strength is set to *required*

equalityBetween: var1 and: var2 plusOffset: aConstantOffset
var1 = var2 + aConstantOffset
strength is set to *required*

equalityBetween: var1 and: var2 plusOffset: aConstantOffset sensitivity: sens
var1 = floor(var2/sens) + aConstantOffset
strength is set to *medium*; *sens* is the sensitivty to the mouse.

equalityBetween: var1 and: var2 plusOffset: aConstantOffset sensitivity: sens strength: strength
var1 = floor(var2/sens) + aConstantOffset
strength is the precedence ordering if not everything can be solved.
sens is the sensitivty to the mouse.

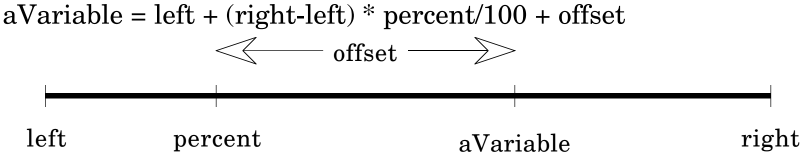
equalityBetween: var1 and: var2 plusOffset: aConstantOffset strength: strength
var1 = var2 + aConstantOffset
strength is the precedence ordering if not everything can be solved.

equalityBetween: var1 and: var2 strength: strength
var1 = var2
strength is the precedence ordering if not everything can be solved.

greaterThanEqualBetween: var1 and: var2
var1 $\geq$ var2

greaterThanEqualBetween: var1 constant: c
var1 >= c

left: left right: right percent: percent offset: offset equal: aVariable
aVariable = left + (right-left) * percent/100 + offset



maximize: aVariable with: aVariableCollection default: aConstant
aVariable = max(aVariableCollection,aConstant)

multiply: aVar1 and: aVar2 equal: aVar3
aVar1 * aVar2 = aVar3

multiply: aVar1 and: aVar2 equal: aVar3 strength: strength
aVar1 * aVar2 = aVar3
strength is the precedence ordering if not everything can be solved.

origin: origin corner: corner xPercent: x yPercent: y offset: offset equal: aPoint
Just like *left: right: percent: offset: equal:* except for 2D.

paragraph: paragraph equalToNumber: numberVariable
The string *paragraph* is made to look like the number *numberVariable*.

plus: aVar1 and: aVar2 equal: aVar3
aVar1 + aVar2 = aVar3
strength is set to *required*

plus: aVar1 and: aVar2 equal: aVar3 strength: strength
aVar1 + aVar2 = aVar3
strength is the precedence ordering if not everything can be solved.

set: aVar equalTo: aValue
aVar = aValue
aValue is a constant

set: aVar equalTo: aValue strength: aStrength
aVar = aValue
aValue is a constant
aStrength is the precedence ordering if not everything can be solved.

stayOn: aVariable
Try not to change the value of *aVariable*.

xMouseConstraint: mouseVariable on: aVariable sensitivity: sensitivity
aVariable := floor(mouseVariable/sensitivity) + aVariable
sensitivity is the sensitivity of the mouse.
strength is set to medium

xMouseConstraint: mouseVariable on: aVariable sensitivity: sensitivity strength: strength
Same as before, except with *strength* as the precedence ordering if not everything can be solved.

xyMouseConstraint: aMouseVariable on: aPointVariable
2D version of *xMouseConstraint* with *sensitivity* = 1
strength is set to *medium*

xyMouseConstraint: aMouseVariable on: aPointVariable strength: aStrength
Same as before, except with *strength* as the precedence ordering if not everything can be solved.

yMouseConstraint: mouseVariable on: aVariable sensitivity: sensitivity
Just like *xMouseConstraint* except for y.

yMouseConstraint: mouseVariable on: aVariable sensitivity: sensitivity strength: strength
Same as before, except with *strength* as the precedence ordering if not everything can be solved.