Graph Editor Construction Kit (GrapE)

Tutorial
and
Reference Manual

Jonathan W. Krueger

March 30, 1999

Table of Contents

List of Figures

DOME Tool Builder's Manual

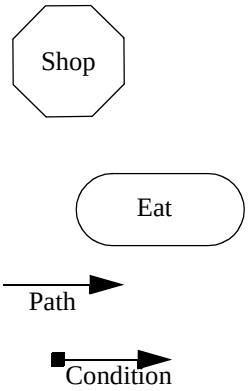
This manual introduces the GrapE graph editor construction kit along with the MetaDOME tool specification tool. It covers the basic concepts behind GrapE, which is implemented in Objectworks/Smalltalk[1] release 4. This manual assumes you are familiar with the programming interface of Objectworks/Smalltalk; the Objectworks manuals can provide some assistance in this area if you need it.

This manual is organized as a detailed tutorial supplemented by reference sections. The tutorial takes you through the steps of building a new graph editor using MetaDOME, from concept to implementation. The type of graph chosen is very simple: two types of nodes and two types of arcs.

GrapE was originally designed as a Petri Net editor. Since then, other graph editors were built by generalizing the basic object classes, all the while maintaining functioning editors. Numerous editors have been built using MetaDOME including: Petri Net (executable), State-Transition Diagrams, Dataflow Diagrams, Express-G, IDEF-1x and Coad-Yourdon Object-Oriented Analysis Notation. Once a programmer is somewhat familiar with MetaDOME, the basics of a new editor can be built in a matter of a few minutes. After a programmer is familiar with GrapE, tools specific enhancements can be made relatively swiftly.

1. Objectworks/Smalltalk is a trademark of ParcPlace Systems, Inc.

1.0 Tutorial



Our objective is to produce a tool that supports the creation and editing of a special kind of graph called a Shopping graph. A Shopping graph has two kinds of nodes: the Shop node and the Eat node, shown at left. There are also two kinds of arcs: the Conditional arc and the Path arc. A conditional arc denotes a choice of destination, whereas a path dictate a single direction. As such, there should be at most one Path arc emanating from any given node. Likewise, Conditional arcs have no meaning if a Path arc is exiting from the same node.

We will follow these steps to create the new editor:

1. Create a DOME Tool Specification Model
2. Specify the Smalltalk class information
3. Define the graph element classes (nodes and arcs)
4. Define the editor tools (icons and cursors)
5. Define the arc constraints

The tutorial will also show the new editor being used to create, save, and print a Shopping graph.

1.1 Creating a DOME Tool Specification Model

Assuming you have VisualWorks/Smalltalk (Release 4) up and running with DOME already installed, create a DOME Tool Specification model from the DoMELauncher by selecting ‘DOME Tool Specification’ from the dialog that appears after pressing the New button on the launcher. An editor will appear as shown in Figure 1.

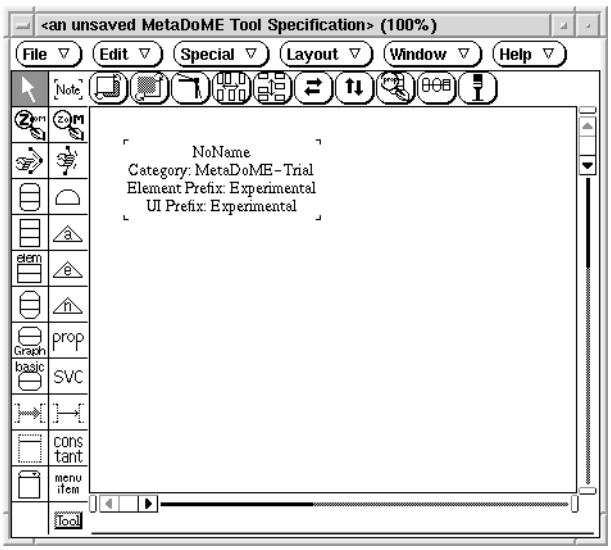


FIGURE 1. Initial DOME Tool Specification model

1.2 Specifying the Smalltalk Class Information

The first thing that should be done when creating a new DOME Tool Specification model is to set up the installation information. This information is used when a model is installed into the Smalltalk environment. To edit the information select the context node and then select Edit->inspect->properties. An editor similar to the one shown in Figure 1 will be displayed in which you should set the Class Category to 'DoMETool-Shopping', the Class Prefix to 'Shopping', and the UI Class Prefix to 'ShoppingUI'.

All generated classes are placed in the Smalltalk category specified by the Class Category. If the category does not exist then it will be created when the model is installed for the first time.

The Class Prefix is appended to the front of each Node and Arc class name that is installed as part of the model. The UI Class Prefix is used in a similar manner to the Class Prefix but is used for the Controller, Editor, and Graph classes that are installed.

The Method Category Suffix is used to name the categories that contain the generated methods for a class. All user defined methods should be placed in a category that does not end with the method category suffix since it is possible to have MetaDOME automatically remove all of the methods contained in those categories which end with the Method Category Suffix.

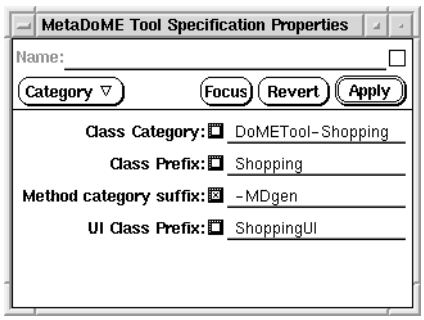


FIGURE 2. DOME Tool Specification Properties

1.2.1 Excursion: Installing a Minimal Tool Specification

This excursion describes the classes and methods that MetaDOME installs by default. Deselect the context node if it is currently selected and then select the menubar option Special->install tool. The Installation Options dialog appears from which the OK button should be pressed. The Installation Options Dialog has several toggles that control what actually gets installed, refer to the MetaDOME users manual for a detailed description of the various options. After a few seconds of computation a Smalltalk Changes window appears from which you should replay all changes.

The installation creates three classes which constitute the core of the user interface. ShoppingUIGraph represents the model class whose instances contain Shop and Eat nodes. ShoppingUIEditor represents the editor class which displays the graph so that it may be edited. ShoppingUIController represents the controller class which interprets gestures (mouse movement, button clicks and keypresses). The model, editor, and con-

troller, together with other, default scaffolding such as editing functions, forms the graph editing application.

Before we examine the methods that were created as part of the installation lets create a Shopping model and see what core capabilities GrapE offers. From the DOMELauncher create a ShoppingUI Diagram. By default, a shopping diagram looks very similar to other DOME tools except there are no tool icons on the toolbar from which to create model specific nodes and arcs as shown in Figure 3. Granted, it is not yet complete, but you can see for yourself the default functionality provided by GrapE.

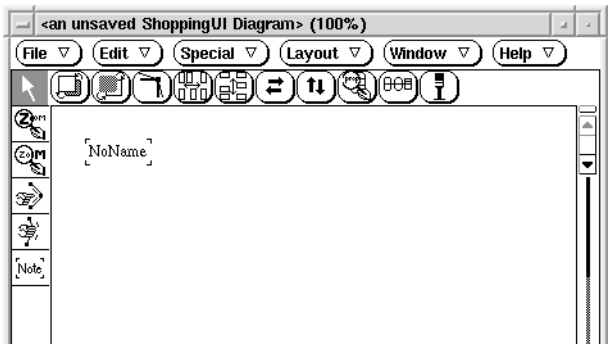


FIGURE 3. Bare-bones Shopping editor

Notice the *Note* tool. This tool is automatically supplied to all GrapE-based applications for annotating the graphs. It is a subclass of NamedNode. Take some time now to familiarize yourself with the default editing tools and the default menu (press the middle mouse button to pop it up). When you are done, close the window in the normal Smalltalk fashion.

To End this excursion, each of the methods that were installed are described in detail in Table 1.

TABLE 1. Methods installed for bare-bones Shopping tool

Class	Method	Description
ShoppingUIGraph	defaultController-Class	Specifies the controller class to use when creating an editor for a shopping graph.
	editorClass	Specifies the editor class to use when editing a shopping graph.
	arcClasses	Returns the arc classes whose instances may be created as part of the graph.

TABLE 1. Methods installed for bare-bones Shopping tool

Class	Method	Description
ShoppingUIGraph class	newWin-dowName	Returns the name that the DOMELauncher will use when the user wishes to create a Shop-ping model. This method is slightly different than most of the others in that it is a stub method which means that you can modify it and not have to worry about it being over-writ-ten when another install is done from this specification.
	toolClasses	Returns a collection of those classes which can be created from the editor of a Shopping graph.
	allDefinedClasses	Returns a collection of class symbols that were installed as part of the tool.
ShoppingUIEditor	tools	Return a collection of tool descriptions that specify the tools that are available from the editor.
ShoppingUIController class	tools	This method is very similar to the tools method on the editor but the returned collec-tion contains a little more information about each tool. Eventually the tools method on the editor will be removed.

1.3 Defining the Graph Element Classes

Our application has two types of nodes and two types of arcs. We will create the node classes first, then the arc classes.

To create the Shop node class:

1. Place a NodeSpec on the graph.
2. Name it ‘Shop’.
3. Set its NamePosition to *Center.*
4. Set its Corners property to *Chorded* with a radius factor of *0.22*. This will give the node an octagon appearance.

To create the Eat node class:

1. Place a NodeSpec on the graph.
2. Name it ‘Eat’.
3. Set its NamePosition to *Center.*
4. Set its Corners property to *Rounded*

To create the Condition arc class:

1. Place an ArcSpec on the graph.

2. Name it ‘Condition’.
3. Sets its Origin Head to be *Always* and its style to *Filled Square*.

To create the Path arc class:

1. Place an ArcSpec on the graph.
2. Name it ‘Path’.
3. Set its Name Presentation to *Never.*

1.4 Defining the Editor Tools (Icons, Cursors, and Keyboard Accelerators)

In order to endow our editor with the ability to create nodes and arcs specific to our Shopping formalism, we need to design some icons and cursors. The icons are placed on the toolbar of the editor while the cursors are used to change the mouse pointer when it is time to create an instance of the object represented by the selected icon. This tutorial will take you through the details of creating one icon (for the Condition class), then the rest are up to you. Fortunately, MetaDOME offers a large selection of icons and cursors from which to choose.

The icons and cursors are specified as properties to the Creation Buttons that were automatically created inside the Tool Palette when the node and arc specs were placed on the graph. Use Table 2 for the information necessary to specify the icons, cursors, and keyboard accelerators. Only the Condition and Path Icons need to be created since all of the others are already available. To set the icon of cursor, first inspect one of the creation buttons, click on the icon or cursor symbol, select an appropriate symbol, and press OK.

TABLE 2. Tool Properties

Class	Icon	Cursor	Key
Shop			s
Eat			e
Condition			c
Path			p

1.4.1 Creating an Icon

To create an icon you need to attempt to set the icon as described in the previous paragraph but instead of selecting one of the available icons you need to select the Add Icon button. After selecting the Add Icon button, the mask editor is opened to allow you to edit your icon. Edit your icon and then press Install when you are happy with it. Installing an icon currently produces some Smalltalk code that is associated with the ShoppingUIEditor class, so provide a name that is a valid Smalltalk method name preferably appended with ‘Icon’, such as conditionArcIcon. After the icon has been installed you may close the mask editor. Finally, set the icon by the process described in the previous paragraph.

1.4.2 Creating a Cursor

Creating a cursor is identical to creating an icon except the cursor's size is kept to 16x16 while the icon's size is kept to 24x24. Also, the Smalltalk code for the cursor is associated with the controller class instead of the editor class.

1.4.3 Excursion: Installing a partially specified tool

This excursion describes the classes and methods that MetaDOME installs for the information that we have specified to this point. Go ahead and install the specification again, see Table 1 for a description of the new and modified methods. Four classes are created as part of the installation as one would expect. The only special thing about the installed classes are who they inherit from. Both Shop and Eat inherit from NamedNode since they have names. Path inherits from NetArc since we specified that a path does not have a name and Condition inherits from NamedNetArc.

TABLE 3. Methods installed for partially specified Shopping tool

Class	Method	Description
ShoppingUIGraph	arcClasses	It now returns the arc classes that we defined.
ShoppingUIGraph class	toolClasses	It now returns the four tool classes.
	allDefinedClasses	It now returns symbols for all of the classes that were installed.
ShoppingUIEditor	tools	It now returns a description of the tools.
	toolBarSpec	Returns a description of the toolbar that is used when editing a Shopping graph.
ShoppingUIController	newShop	Used to create a Shop node.
	newEat	Used to create an Eat node.
	newCondition	Used to create a Condition arc.
	newPath	Used to create a Path arc.
ShoppingUIController class	tools	It now returns a description of the tools.
ShoppingCondition	visibleOriginHead	By default the origin head is not visible. This methods makes it visible.
	originHeadStyle	The origin head will be a filled square when displayed.
ShoppingCondition class	whatAreYou	Returns a string that describes generically an instance of this class.
ShoppingEat	radiusFactor	Information used for displaying the corner of the node.
	chordedCorners	Should the corner be chorded instead of square?
	namePositionPolicy	Where should the name be displayed with respect to the border of the node.
ShoppingEat class	containerClasses	The classes that can contain an instance of this class. It is used for constraint enforcement.

Now try the tool a second time. This time, the editor should come up showing the four tool icons, as shown in Figure 4. You can now create Shop and Eat nodes but you still cannot create condition and path arcs since we have not specified the constraints for the arcs yet.

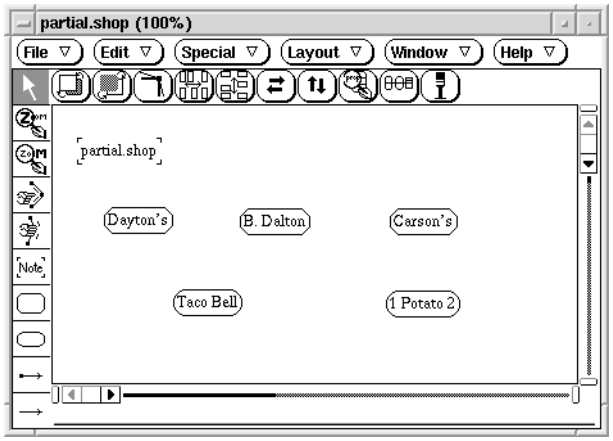


FIGURE 4. Partially functional Shopping editor

1.5 Defining the Arc Constraints

Now that we are able to create nodes it would be nice if we were able to create arcs also. To specify the constrains necessary for the Shopping tool we will need to create eight arc constraints as shown in Figure 4.

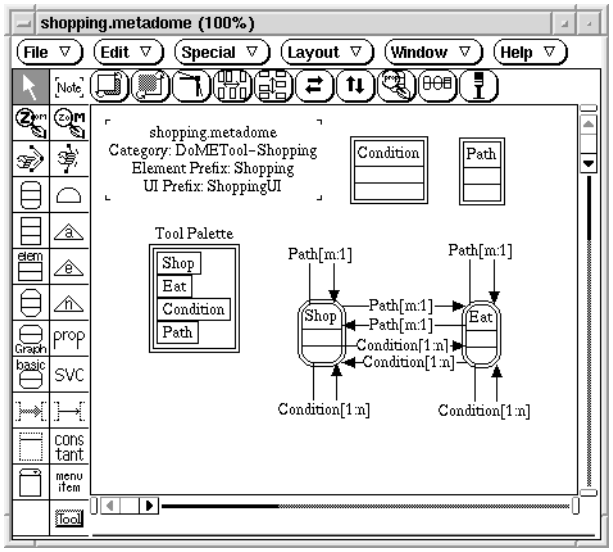


FIGURE 5. Complete DOME Tool Specification model

The arc constraints for the Condition and Path arcs are very similar as seen in Figure 4 which shows most of the information about the arc constraints except for the reflective-

ness of the arcs constraints that have the same origin and destination. For these arcs the reflectivity should be set to false. This prevents a Shop or Eat node from being both the origin and destination of the same arc.

1.5.1 Excursion: Installing the fully specified tool

This excursion describes the last set of methods that are installed as a result of specifying the arc constraints. Go ahead and install the specification again and see Table 1 for a description of the new methods. At this point the Shopping tool should be fully operational.

TABLE 4. Methods installed for fully specified Shopping tool

Class	Method	Description
ShoppingCondition class	endpointClasses	Returns information that is used in constraint enforcement.

1.6 Using the Shopping Editor

At this point, you have a fully functioning Shopping editor. Bring up the editor (if you do not already have it running), and create a graph. You can draw the graph shown in Figure 14, or you can design your own.

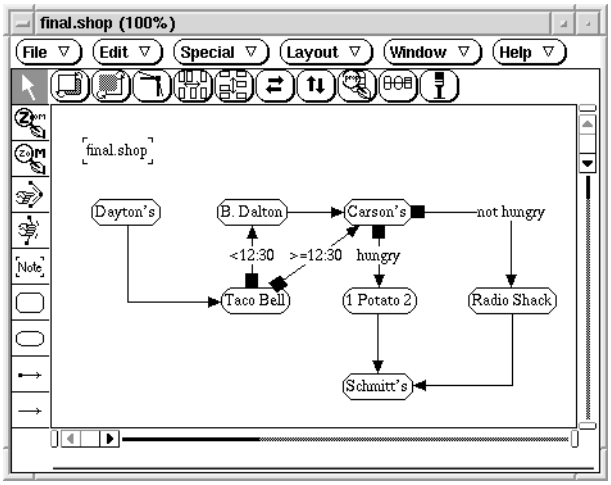


FIGURE 6. Fully functional Shopping editor

Once you have the graph looking the way you want, try printing it by selecting the "File->print" menu item with the middle mouse button.

Next, save the shopping graph by choosing the "File->save" menu. You should be presented with the requester shown in Figure 7. Type in the name of the file you wish to save the graph in and hit *return*. The cursor will change shape momentarily while the graph is being written to the file. Close the editor window and open the saved model

from the DOMELauncher. The load meter should show the model being loaded, then your previous graph should be displayed in the new window.

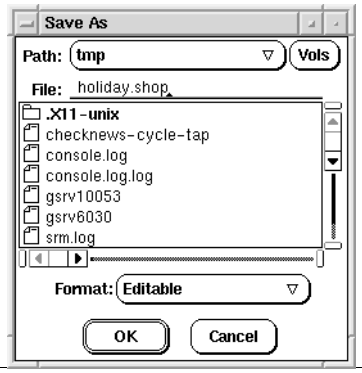


FIGURE 7. GrapE “save” requester, showing the directory /tmp

2.0 Extending the Tutorial Example

In this section we describe various features of MetaDOME and GrapE by extending the tutorial example. We enhance the Shopping model by adding the following features:

1. Add a shopping list to a shop.
2. Modify the tool column to have two rows of tools instead of just one.
3. Associate an expected price with each item we plan to buy.
4. Modify the display of the Eat node so that it appears to be a fast food restaurant.
5. Refine the connection constraints.
6. Implement a tool using Alter that produces a shopping list.
7. Add a property via the User Defined Property model.

2.1 Adding a List Element

We will first extend the tutorial example to maintain a shopping list of items that we plan to purchase at each store.

1. Create a List Element and set its name to “Item”. It is a wise practice to give a description to each node you create in a MetaDOME specification, so go ahead now and give the Item List Element node a description now.
2. Create an Element Indicator and connect it to the Shop NodeSpec.
3. Name the newly created arc “items”. By default, the objects are sorted when displayed. If we didn’t want the items sorted then we could set the sorted property to false.
4. Connect an arc from the List Element to the Element Indicator.
5. Create an Icon and Cursor for the Item Creation Button. For the example we created an icon and cursor in the shape of a dollar sign. We also set the keyboard accelerator to “i”.

The updated MetaDOME Specification can be seen in Figure 8.

2.1.1 Excursion: Installing the new capabilities

This excursion describes the set of methods that are installed as a result of adding the List Element. Go ahead and install the specification again and see Table 1 for a description of the new methods.

TABLE 5. Methods installed for list element enhancement

Class	Method	Description
ShoppingItem	facetFor:	Returns a symbol indicating which compartment of the PartitionedCollection the object should be placed in if the object’s container were the given container. An Item can be placed in the #items facet of a Shop in our example.
ShoppingItem class	containerClasses	An item can be placed in a Shop.

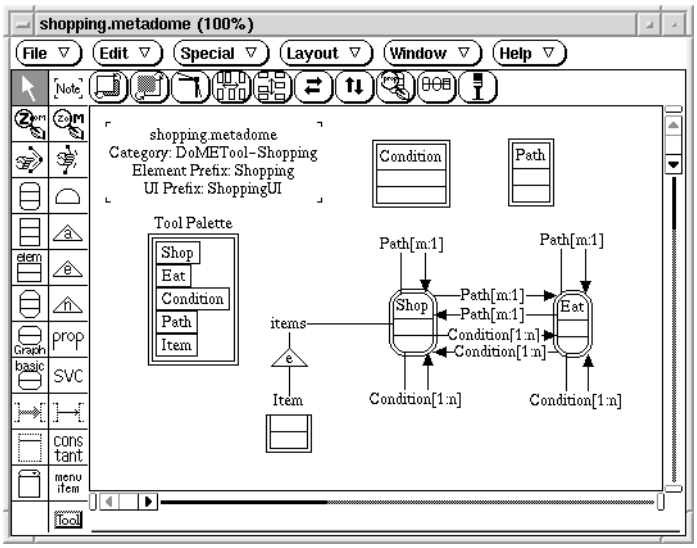


FIGURE 8. DOME Tool Specification model with Item List Element

TABLE 5. Methods installed for list element enhancement

Class	Method	Description
ShoppingShop	initializeCompo-nents	When a shop node is created, this method creates a facet into which Items are placed.
	hasListContent	Since the shop node may have components that are List Elements this method returns true.

Go ahead now and try the tool again. You are now able to add Items to Shop node as well as move items between Shop nodes. There is one problem though: the name of the Shop is in the middle of the shopping list for the store. To rectify the situation, set the Shop's name position to InsideTop and re-install. Create a new Shop node and add some items to see the new behavior. Also notice how the items are sorted alphabetically as seen in Figure 9. This is the result of setting the sorted property on the arc from the Element Indicator to the Shop to true.

2.2 Multiple columns of Tools

The next feature we plan to add to the tutorial example is to have the toolbar be two columns wide instead of just a single column. We do this because it would be good if the item tool was adjacent to the shop tool since they are used in conjunction with each other.

Creating a multi-column toolbar is very easy. Simply inspect the Tool Palette node and set its columns property to two. MetaDOME currently supports a toolbar width of one to three columns. After the columns property is modified then the Tool Palette node appears with a new Tool Column that is empty. Next move the Item Creation Button to the second column so that it is adjacent to the Shop Creation Button.

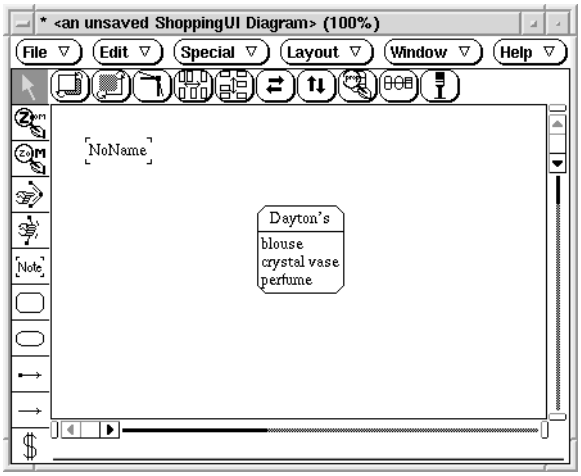


FIGURE 9. Shopping editor with shopping list

Unfortunately, this will not get us exactly what we want. If we were to install and create a new Shop model we would notice that the Item creation button is at the top of the second column while the Shop creation button appears immediately after the standard tool buttons. To get the appearance that we desire, we must add the standard tools to the DOME tool specification via Special-><add std tools> and then order the standard tools in a fashion similar to existing tools. Do this now.

Next, go ahead and install the tool again. Notice that the toolBarSpec and tools methods were modified since they describe what the toolbar looks like. Finally, start a new shop tool and see that the toolbar is now two columns wide.

2.3 Adding a Property

We will next associate an expected price with each item. The price will represent the number of dollars that we expect to pay for the item. The following steps describe the process of adding a new property.

1. Select the Property tool and drop it into the Item node.
2. Name the property “expectedPrice”.
3. Inspect the properties of the expectedPrice property.
4. Set its label to “Expected price:”.
5. Set its type to Number.
6. Set it so that it cannot be TBD.
7. Give it an initial value of 0.

The updated DOME Tool Specification can be seen in Figure 8.

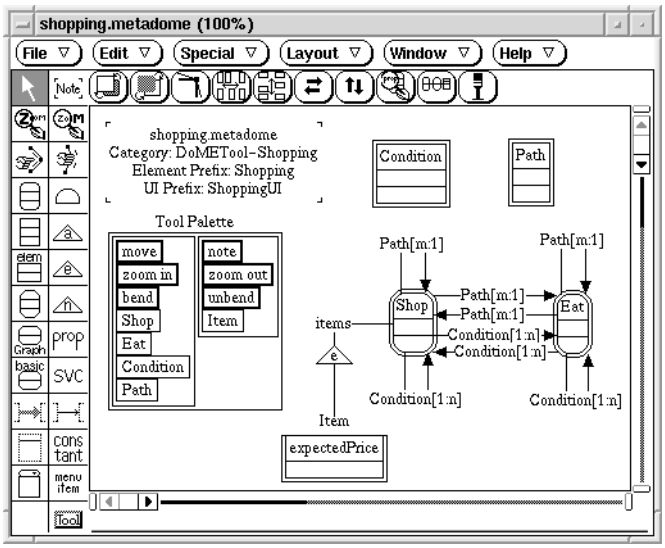


FIGURE 10. DOME Tool Specification model with the expectedPrice property

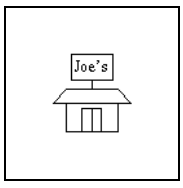
2.3.1 Excursion: Installing the new capabilities

This excursion describes the set of methods that are installed as a result of adding the expectedPrice property. Go ahead and install the specification again and see Table 1 for a description of the new methods. If you wish you can create a new Shop model with some items and set their price.

TABLE 6. Methods installed for property enhancement

Class	Method	Description
ShoppingItem	expectedPrice	The method for retrieving the expected price.
	expectedPrice:	The method for setting the expected price.
ShoppingItem class	localStandard-Properties	This method returns a set containing definitions of those properties that have been specified for this class.

2.4 Changing the Shape of a Node



We will modify the appearance of the Eat node to have a facade similar to a fast food restaurant. When a unique shape for a node is specified, the programmer must implement methods that compute the size of the shape, draw the shape, and determine where to clip lines that are attached to the node. MetaDOME generates default methods for these capabilities when a specification is installed with a custom shape. The following list describes the steps necessary to customize the shape of a node from the specification.

1. Inspect the Appearance properties of the Eat node spec.

2. Set the name position to inside top. This will allow the name to appear in the sign that will be on top of the restaurant.
3. Set the border shape to be custom.

We need to install the specification next so the default shape methods are installed. See Table 1 for a description of the methods that are removed as well as installed.

TABLE 7. Methods installed for shape enhancement

Class	Method	Description
ShoppingEat	radiusFactor	[Removed since it is not needed.]
	chordedCorners:	[Removed since it is not needed.]
	namePositionPolicy	[Removed since it is now the default.]
	clip:alongLineTo:	Return the point that should be used as the end point when connecting an arc to this node.
	computePreferredBounds	Return the preferred bounds of the node. As a side effect set the preferredBounds and borderShape instance variables.
	displayShapeOn:	Display the shape of the node on the graphics context.

If you wanted to you could create a new shopping model and see what an Eat node looks like when the default display routines are used. It looks like a rectangle. As you implement your own shape routines you should keep in mind that the three routines are closely related. The display routine and the clipping routine must abide by the bounds determined by the compute preferred bounds routine. If the display routine determines its own bounds then the act of re-displaying the node may leave graphic refuse on the display area. If the clipping routine computes its own bounds then arcs may appear not to be attached to the node. If ever graphic refuse is on the display area or an arc appears not to be attached to a node then one or both of these routines are probably implemented incorrectly.

With that all said, I can now tell you that you will never modify the three generated display routines. You will actually implement the routines that are called from the generated routines which are shown in Table 1.

TABLE 8. Methods to be implemented in order to support custom shapes

Class	Method	Description
ShoppingEat	clipCustom:alongLineTo:	Return the point that should be used as the end point when connecting an arc to this node.

TABLE 8. Methods to be implemented in order to support custom shapes

Class	Method	Description
	computeCustom-PreferredBounds	Return the preferred bounds of the node. As a side effect set the preferredBounds and border-Shape instance variables.
	displayCustom-ShapeOn:	Display the node on the graphics context.

The first step is to implement the computeCustomPreferredBounds method. The existing clipping and display routines will make use of it. It is assumed that the programmer has some experience with programming in Smalltalk so some of the Smalltalk environment related aspects will be covered very swiftly.

1. Add the category ‘displaying’ to the ShoppingEat class. The new display routines will be placed in this new category. It is important to never place any methods in those categories that end with the method category suffix specified in the tool specification, in this case: -MD-gen. If we added a method to a category that ended in -MDgen and later on we did a Special->’clean installed tool’ then we would lose that method. Also, if you modify a method that is in one the categories generated by MetaDOME then you should move that method to a different category that does not get removed.
2. Create the method computeCustomPreferredBounds. It is very important that the position of the node be equal to the center of the preferred bounds and that the bounds be based on the current pushOut value. The value returned by pushOut varies depending on the magnification of the graph. Also, be sure to set the borderShape and preferredBounds instance variables. After you have implemented this method

you should be able to create Eat nodes and notice that the shape's size seems appropriate.

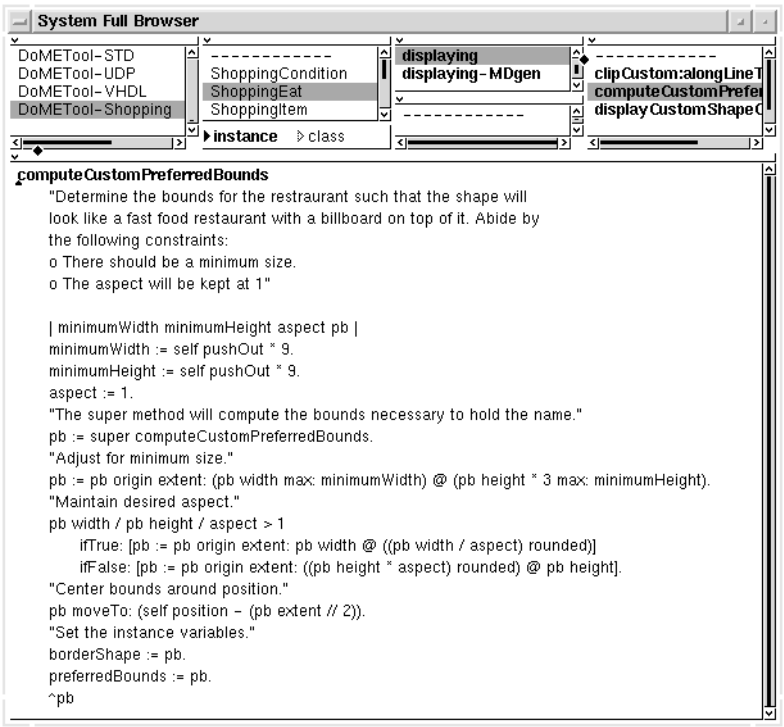


FIGURE 11. Instance method ShoppingEat>computeCustomPreferredBounds

3. Create the method displayCustomShapeOn: to actually draw your custom shape as shown in Figure 11. The most important part of the display routine is that it must be based on the preferred bounds of the node.

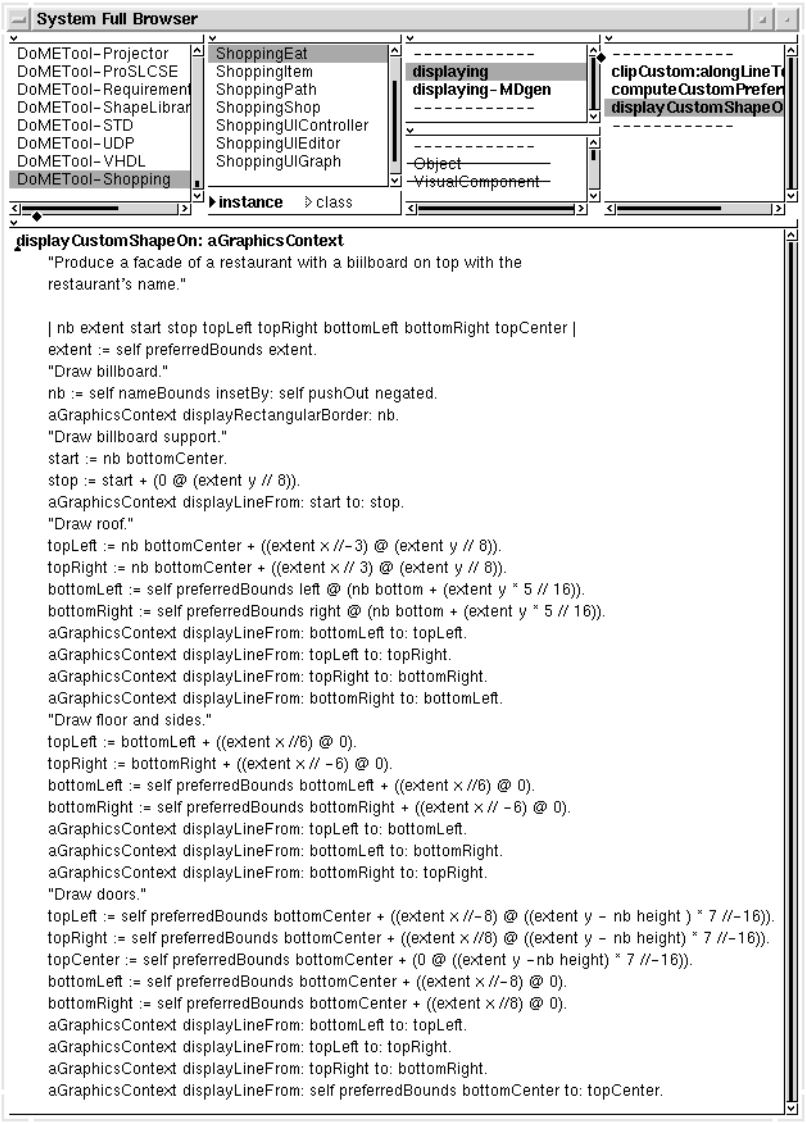


FIGURE 12. Instance method ShoppingEat>dispayCustomShapeOn:

4. Create the clipping routine if necessary. For this example, I will create a clipping routine so that all arcs are attached to the threshold of the doorway. Also, the clip-

ping routine must be based on the preferred bounds of the node. Create the method clipCustom:alongLineTo: as shown in Figure 11.

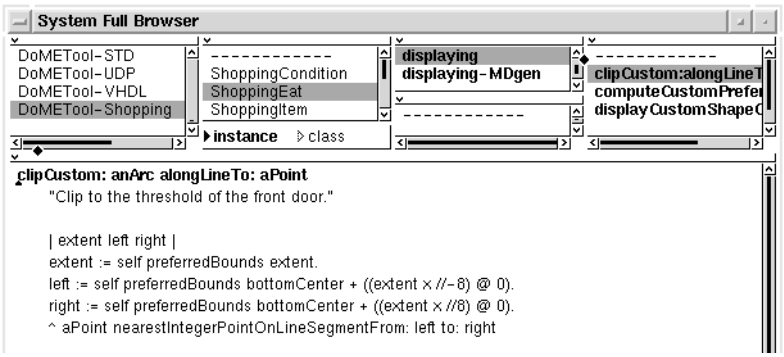


FIGURE 13. Instance method ShoppingEat>clipCustom:alongLineTo:

After the three display routines are implemented you can create a new shopping model such as that shown in Figure 14.

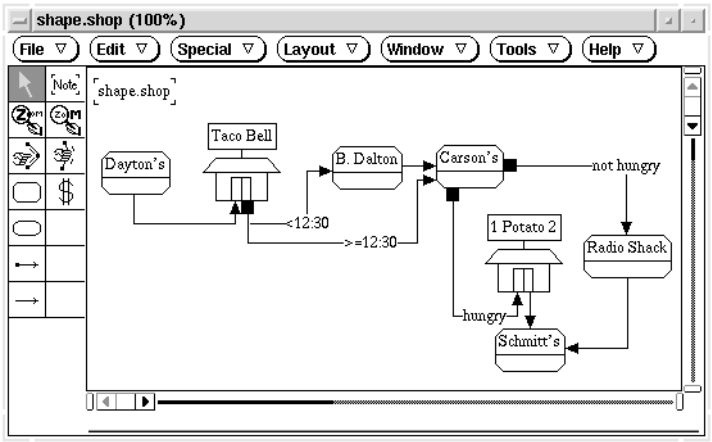


FIGURE 14. Fully functional Shopping editor with a customized Eat shape

2.5 Refining Connection Constraints

If you were to validate the shopping tool that has been implemented so far against the requirements then you would find that one of the requirements has not been fulfilled. That incomplete requirement was originally stated as, “...Conditional arcs have no meaning if a Path arc is exiting from the same node.” Currently, there is no way to specify this constraint from MetaDOME but now that you have some experience writing Smalltalk code for a tool it will be a fairly straightforward effort. We will implement this constraint by making certain that a Constraint and Path arc do not emanate from the same node.

To enforce this requirement we will need to implement a method on both the Conditional and Path classes that makes certain that there are only like kind of arcs emanating from the origin node as seen in Figure 11. The method for the Condition class is identical to this one. All arc constraints should be implemented by specializing NetArc's canConnect:to:with: class method. This method is called whenever new arcs are created as well as when an arc's origin or destination is moved to another node. There are a couple of important issues to understand when specializing this method. The first issue is that the superclass method must be called since it checks the constraints that were generated from MetaDOME when the tool was installed. The second issue is that if an arc cannot be connected from node1 to node2 then an explanation should be added to the error reporter so that the user gets some useful feedback as to why the connection could not be made.

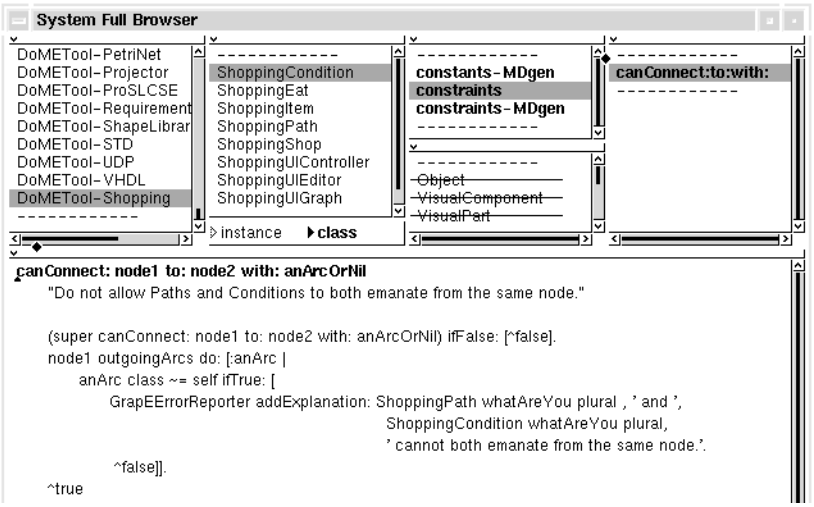


FIGURE 15. Class method ShoppingPath>canConnect:to:with:

Sometimes it is more natural to associate the constraint checking mechanism with the node being connected from or to rather than with the arc class. This can be done by specializing the instance level methods on Node named allowConnectionOf:from: and allowConnectionOf:to:.

2.6 Implementing a Tool using Alter

This enhancement provides an example of how to integrate a tool into DOME using Alter to implement the tool. Specifically, the tool selects those shopping nodes with items to be purchased and produces an alphabetically organized shopping list. Refer to the Alter Programmer's Reference Manual Writing Tools Appendix for a thorough discussion about writing tools. There are three steps necessary to implementing and integrating a tool:

1. Create a Registration File.
2. Initialize the *dome-load-path* variable.
3. Implement the tool in Alter.

2.6.1 Registration File

The registration file contains a collection of function specifications that describe the available tools. Each function specification specifies the type of object that the tool can be applied to, the source file of the tool, and the keyboard accelerator to use to invoke the tool. DOME must also be told where to find the registration file. This is done by setting the DoMEUserFunctions environment variable. For this example lets assume that the registration file exists in the /tmp directory, therefore, the variable would be set by 'setenv DoMEUserFunctions /tmp/userFunctions.dome' from a csh shell. This would imply that the registration file was named "userFunctions.dome".

The following lines are the contents of the registration file necessary for this example.

```
[DoMEUserFunctionList
 [DoMEUserFunctionSpec
   functionName: 'Shopping List'!
   graphType: #ShoppingUIGraph!
   sourceFile: 'shopping-list.alt'!
   keySequence: 'tsl'!
 ]
]
```

2.6.2 Initialize *dome-load-path* Variable

The *dome-load-path* variable provides a means for the registration file to be less site specific since it allows the source file of a function specification to not be fully specified. DOME will look through the directories specified by the *dome-load-path* variable for the source file when it comes time to invoke the tool. The *dome-load-path* is normally initialized by the '.domeinit' file in the user's home directory. The following line of Alter code is sufficient to specify the load path.

```
(define *dome-load-path* (list "/tmp" "/usr/local/dome/lib"))
```

2.6.3 Implementation

The implementation for the shopping list tool is fairly simple once you have a firm understanding of Alter. Keeping a copy of the Alter Programmer's Reference Manual close at hand when you are writing a tool will greatly facilitate its implementation. In order for this example to work correctly, the file containing the implementation should be located in the /tmp directory and be named shopping-list.alt. The following lines are the Alter code necessary for implementing a simple tool:

```
;; Name: shopping-list
;; Purpose:
;;   Produce a shopping list of the items to be bought at the various
;;   stores.

;; Remember the file into which the shopping list is to be written so that
;; the next time this tool is invoked it will use the same file.
(define shopping-output-file
  (if (bound? shopping-output-file) shopping-output-file nil))

;; Request a file and output the shopping list to the specified file.
(define (shopping-list-main graph)
```

```
  (let ((file (if (nil? shopping-output-file)
                  (request-new-file-name)
                  (request-new-file-name shopping-output-file))))
    (if (not (nil? file))
        (begin
          (set! shopping-output-file file)
          (with-output-to-file
            (filename->string file)
            (lambda () (shopping-list-report graph)))))))

;; Loop through the stores writing out the items to be purchased.
(define (shopping-list-report graph)
  (let ((stores (select
                  (nodes graph)
                  (lambda (shop)
                    (and
                     (is-kind-of? shop ShoppingShop)
                     (not (= 0 (length (get-property "items" shop)))))))))
    (if (= 0 (length stores))
        (warn "There are no stores with items to be purchased!")
        (begin
          (display "Shopping List")
          (newline)
          (display "==========")
          (newline)
          (show-progress-for-each
           "Processing shops"
           shopping-list-output-shop
           (sort stores (lambda (x y) (string<? (name x) (name y)))))))))

;; Write out the store name and the list of items to be purchased.
(define (shopping-list-output-shop shop)
  (let ((shopname (name shop)))
    (newline)
    (display "  ")
    (display shopname)
    (newline)
    (display "  ")
    (do ((count (length shopname) (- count 1)))
        ((= 0 count))
      (display "-"))
    (newline)
    (for-each
     (lambda (item)
       (display "   ")
       (display (name item))
       (newline))
     (sort
      (get-property "items" shop)
      (lambda (x y) (string<? (name x) (name y)))))))
```

```
;; Point DOME towards the entry point for this user-defined tool.
shopping-list-main
```

To execute the tool, create a ShoppingUI model, create some shops with some items to be purchased, and select Tools->User Defined->Shopping List. The following list is representative output from the tool:

```
        Shopping List
        ==========

Ace Hardware
---------------
 socket set

Dayton's
---------
 pants
 shirt
```

2.7 Adding a User Defined Property

Adding a property via a User Defined Property model isn't really an enhancement like the other enhancements that have been described so far but it is worthwhile to know how one can go about adding properties to a model without modifying the MetaDOME specification of the model. There are two steps necessary to add and use a user defined property. You must first create a User Defined Property model with the necessary properties defined and then you must associate the User Defined Property model with the model that will actually make use of the user defined properties. Lets first create the User Defined Property model.

1. Create a User Defined Property model via the create button of the DOME launcher.
2. Merge the tool specification of the Shopping model into the newly created User Defined Property model by selecting "File->merge from". This doesn't actually merge the nodes of the tool specification with the User Defined Property model but merely links the tool specification to the model via its filename. A view of the tool specification is automatically created as part of the User Defined Property model when the merge is done as well as when the User Defined Property model is opened for editing.
3. Add the new property to the Item List Element node. Simply select the property tool and drop a property on the Item List Element node as shown in Figure 16. The user defined properties are shown with <> brackets around them to make them more easily recognized. The new property should have the following characteristics, as shown in Figure 17:
   - Name => "actualCost"
   - Label =>"Actual cost:"
   - Show after => expectedPrice
   - Type => Number
   - Can Be TBD => false

- Initialize => true

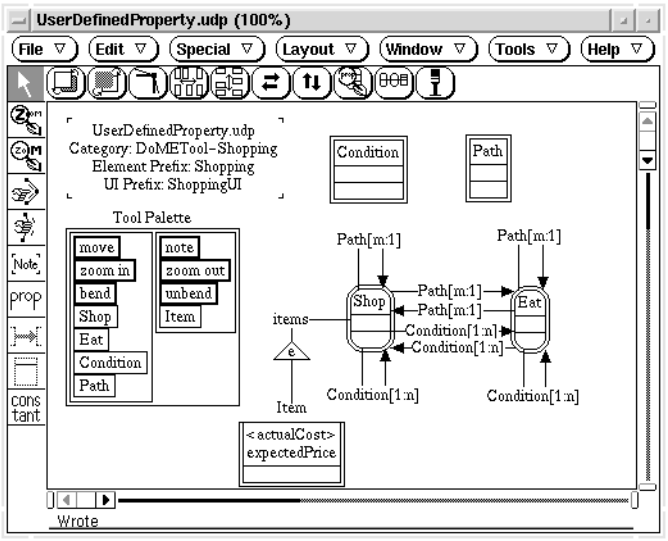


FIGURE 16. User Defined Property model with the actualCost property

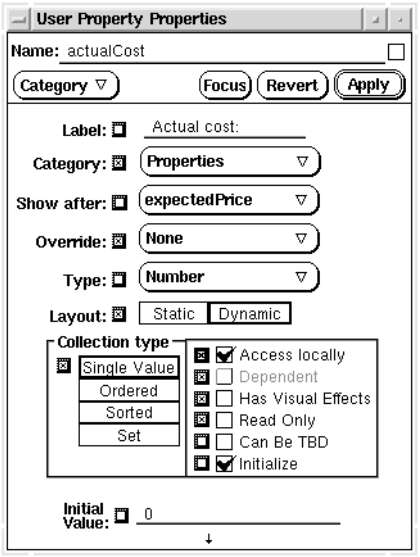


FIGURE 17. Inspector focused on the actualCost property

Save the model after you are done specifying the property. Now that the model has been saved it can be made use of by a Shopping model. To associate a user defined property model with the shopping model, you must inspect the context node of the shopping model and add the User Defined Property file as a schema file of the shopping model. Select the Property Schema category from the inspector and add the file to the schema files property. To prove that the property is now available create a Shop node with an

item in it and inspect it. The item will now have two properties, expectedPrice and actualCost.