VisualAge Smalltalk

Visualization Tools User's Guide

Version 5.5

Note

Before using this document, read the general information under “Chapter 1. Notices” on page 1.

August 2000

This edition applies to Version 5.5 of the VisualAge Smalltalk products, and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product. The term “VisualAge,” as used in this publication, refers to the VisualAge Smalltalk product set.

Portions of this book describe materials developed by Object Technology International Inc. of Ottawa, Ontario, Canada. Object Technology International Inc. is a subsidiary of the IBM Corporation.

If you have comments about the product or this document, address them to: IBM Corporation, Attn: IBM Smalltalk Group, 621-107 Hutton Street, Raleigh, NC 27606-1490. You can fax comments to (919) 828-9633.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Chapter 1. Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of the intellectual property rights of IBM may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY, USA 10594.

IBM may change this publication, the product described herein, or both.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

- IBM
- VisualAge

Chapter 2. About this book

This book helps you use the Visualization Tools for VisualAge Smalltalk.

Conventions used in this book

This book uses the following highlighting conventions:

Highlight style	Used for	Example
Boldface	New terms the first time they are used	VisualAge uses **construction from parts** to develop software by assembling and connecting reusable components called **parts**.
	Items you can select, such as push buttons and menu choices	Select **Add Part** from the **Options** pull-down. Type the part's class and select **OK**.
Italics	Special emphasis	Do *not* save the image.
	Titles of publications	Refer to the *VisualAge Smalltalk User's Guide*.
	Text that the product displays	The status area displays *Category: Data Entry*.
	VisualAge programming objects, such as *attributes, actions, events, composite parts,* and *script names*	Connect the window's *aboutToOpenWidget* event to the *initializeWhereClause* script.

`Monospace font`	VisualAge scripts and other examples of Smalltalk code	`doSomething`
		`  |  aNumber aString  |`
		`  aNumber := 5 * 10.`
		`  aString := 'abc'.`

	Text you can enter	For the customer name, type `John Doe`

Chapter 3. Introduction to the Visualization Tools

The Visualization Tools feature is a growing suite of visual tools and associated accessories for rapidly gaining an understanding of program (mis)behavior.

You can use these tools to observe program behavior, as it occurs, and to explore object state and relationships.

The following chapters explain how to use the tools of this feature.

- "Chapter 4. The Object Visualizer" on page 7 describes a tool which provides a visual representation of the behavior of objects in your program.
- "Chapter 5. The Widget Scope" on page 17 describes a tool which reveals the objects "behind the scenes" of a visual interactive Smalltalk program, and allows individual objects to be added to the Object Visualizer display.
- "Chapter 6. The Snooper" on page 21 describes a tool which provides a hierarchical textual view as a means of exploring object state and relationships, in order to discover objects to be added to the Object Visualizer displays.

Together, and along with the other tools of the IBM® Smalltalk programming environment — debuggers, inspectors, and browsers — these tools form a very powerful facility for understanding program behavior. This can be invaluable to a programmer, because gaining an understanding of program behavior is central to debugging, tuning, and reuse of programs, class libraries, and application frameworks.

Although very simple in concept, these tools can lead to significant improvements in the productivity of day-to-day Smalltalk programming.

Chapter 4. The Object Visualizer

The Object Visualizer is a tool that enables you to analyze object activity and interaction. With this tool, you can **visualize** objects in your application, which enables you to do the following:

- See how many instances of a class exist at any time during application execution
- Watch a visual representation of message traffic between objects
- Identify which objects are busiest, and which are most idle
- Determine which clusters of objects are closely related to one another, based on the amount of message traffic between them

These tools can help you understand your application in order to correct problems or improve performance. By analyzing the message traffic in your application, you can identify potential trouble spots or bottlenecks.

The Object Visualizer provides two views that you can use to visually analyze your application performance:

- The main **Object Visualizer** window is a visual representation of object instances. For each visualized class, the Object Visualizer displays an icon representing each instance of that class. This view also provides several indicators you can use to track the level of message traffic between the instances.
- The **Cluster View** is a dynamic visual representation of the relationships between objects. The Cluster View measures the level of interaction between objects, visually arranging the objects to show which objects are closely related (based upon the amount of message traffic between them). You can use the Cluster View to analyze either classes or instances.

Note: This chapter describes the Object Visualizer as it appears in the base VisualAge® for Smalltalk environment. When the Distributed Feature for IBM Smalltalk is loaded, the Object Visualizer has additional function. Please see the Distributed Feature User's Guide for a description of that additional function.

Opening the Object Visualizer

To open the Object Visualizer, select **Open Object Visualizer** from the **Visualization** menu of the System Transcript window. The Object Visualizer window opens.

The Object Visualizer has two panes. On the left side is a list of the classes you have selected for visualization. On the right side is an area where the Object Visualizer displays a visual representation of the instances of each class. (If you have not yet selected any classes, both panes are empty.)

Selecting Classes to Visualize

To visualize a class in the Object Visualizer, follow these steps:

1. Select **Add Classes** from the **Class** menu, or from the popup menu of the left-hand pane.
2. A window appears listing all applications currently loaded in the image. Select the application containing the class you want to visualize.

3. In the **Classes** list, select the class you want to visualize.
4. Select the >> push-button to add the class to the **Selected Classes** list.
5. When you have selected all of the classes you want to visualize, select the **OK** push-button.

The names of the classes you selected appear in the left-hand pane of the Object Visualizer.

To remove a class from the Object Visualizer, select the class and then select **Remove Classes** from the **Class** menu, or from the popup menu in the left-hand pane.

Analyzing Object Activity with the Object Visualizer

After you have added classes to the Object Visualizer, you can watch as instances of those classes are created and as they send and receive messages. For each instance of a visualized class, a box-shaped icon appears in the right-hand pane of the Object Visualizer window, next to the name of its class.

The Object Visualizer uses animation to depict activity of the object instances it displays. The animation provides three indicators of object activity:

- The **activity indicator**, a small horizontal line through each instance icon, moves upward each time an object sends or receives a message. You can watch for this motion in order to identify which objects are active.
- When an object is active, its color gradually changes. By default, each icon starts out blue, which indicates the "coolest" state. As an object receives messages and executes its methods, the color gradually changes to red, the "warmest" state. This color change gives you a visual indicator of which objects in your application are "hot"—that is, which objects are undergoing the most activity.

  Each icon is divided into two halves, which change color independently:
  - The left side represents received messages. The more messages the object receives, the warmer the left side becomes.
  - The right side represents CPU utilization. The more processor time the object uses, the warmer the right side becomes.

    Note: The warming and cooling values of visualized objects are relative; in other words, an object is considered hot if it has been very active in comparison with other objects in the application. Therefore, an object might appear hot at first, but then appear cooler as other objects in the application catch up with it.
  - If both the sender and receiver of a message are represented by icons in the Object Visualizer, the view shows the message itself as a line drawn between the objects. These lines always originate from the right side of the sender icon and end at the left side of the receiver icon.

Getting More Information about Objects

The Object Visualizer also provides several options you can use to get more information about the visualized objects. These options are available from the popup menus of the icons in the **Instances** pane of the Object Visualizer. To access this menu, place the mouse pointer over an instance icon and press mouse button 2. (You can also select an instance and then select an option from the **Instance** menu.)

- Select **Browse State** to open a browser displaying the values of the object's instance variables. This browser is dynamically updated, so you can see the values change as your application runs.
- Select **Browse Behavior** to open a browser displaying information about the object's methods. For each method, this browser indicates how many times that method has been called, and how much CPU time it has used. This browser is also dynamically updated as your application runs.
- Select **Inspect** to open a Smalltalk inspector on the object.
- Select **Show Creator** to see a visual indication of which object created the selected object (usually by calling *new*). This indicator appears as a line drawn in the Object Visualizer window, from the creator to the selected object. (If the creator is not a visualized object, this line runs to the edge of the Object Visualizer window.)

  Select **Hide Creator** to remove the visual indicator of the object's creator.

In addition, the following options on the **Options** menu apply to all visualized objects:

- Select **Show Animation** to enable the animated activity indicator. This option is selected by default.
- Select **Show Message Path** to enable the lines drawn to represent method calls between objects. This option is selected by default.
- Select **Show Object Identifier** to label each instance icon with a unique numeric identifier.
- Select **Show Number Of Instances** to see a count of the instances of each class. The total number of instances appears after each class name in the **Classes** pane.
- Select **Set Font** to choose the font used in the Object Visualizer.
- Select **Set Color** to choose the colors used in the Object Visualizer.
- Select **Set Delay Time** to set the length of time the Object Visualizer delays after processing and displaying one message, before proceeding on to the next message. This option makes it possible to slow application execution in order to more easily see individual method calls. By default, the Object Visualizer does not insert a delay.
- Select **Reset** to reset all objects to their coolest state.

Example: Using the Object Visualizer

To see how the Object Visualizer displays object activity, try the following example.

We will examine the behavior of the sample application *VtExampleApp*, an application which does some simple graph manipulation.

Note: You can follow the steps below, and get a feel for using the Object Visualizer, even if you are unfamiliar with graph theory and don't understand what the example application is supposed to do. In fact, even though you are just a casual observer, you may gain a bit of an understanding of the coarse structure of the application's behavior, just from this very brief visualization session!

1. From **Visualization** menu of the system transcript select **Open Object Visualizer**. An Object Visualizer window will appear.

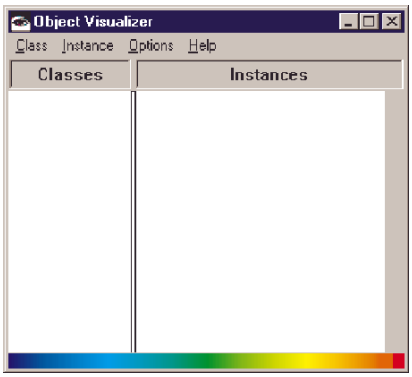


Figure 1.

2. From the **Class** menu of the Object Visualizer, select **Add Classes...**. An **Add Classes** window will appear.

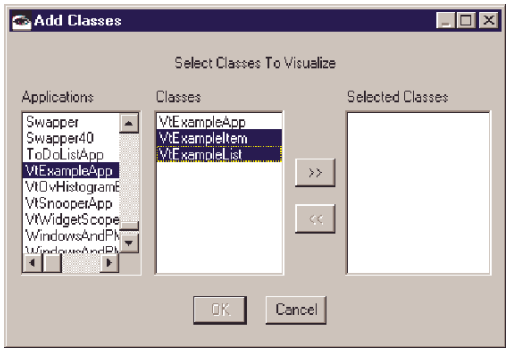


Figure 2.

3. Scroll the **Applications** column down, and select *VtExampleApp*. The **Classes** column will fill with the classes of the *VtExampleApp* application.
4. Select *VtExampleItem* and *VtExampleList* in the **Classes** column.
5. Press the **>>** button. The two classes will move from **Classes** to **Selected Classes**.
6. Press **OK**. *VtExampleList* and *VtExampleItem* will appear in the Object Visualizer's **Classes** column.

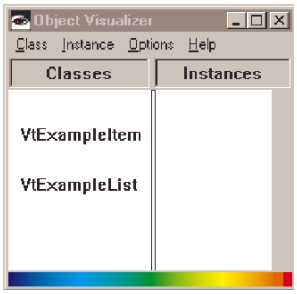


Figure 3.

Note: When you first open an Object Visualizer, there may already be some classes listed in the **Classes** column. This is because the Object Visualizer remembers which classes were being visualized in the previous session and

restores them at the start of the next session. If you wish, you can remove such classes by selecting them and choosing **Remove Classes** from the **Class** menu.

7. Open a new workspace, e.g. by selecting **New** from the **File** menu of the System Transcript, and resize it to be fairly small.
8. Arrange the windows on your screen so that no other windows overlap with the Object Visualizer window. Shrinking the workspace to barely reveal the text that it will contain is a good idea.

   This example concerns a VtExampleList, a slightly random graph of VtExampleItems. During the setup, the items figure out which ones are successors of each other. This can be seen quite clearly as follows.
9. Evaluate the expression `VtExampleList new: 8`. Watch the Object Visualizer. A VtExampleList will appear (a divided blue box), and then a stream of VtExampleItems (more divided blue boxes).

   The VtExampleList will then send a message to each of the VtExampleItems. As each message is being sent and the corresponding method is being executed, a line appears between the two boxes that represent the sender and receiver of the message.

   Then each VtExampleItem will send a message to each other VtExampleItem in turn.

   In the next step, you will see that this behavior corresponds to initializing the items of the graph, and then computing the successor sets of the items, to connect the graph components.

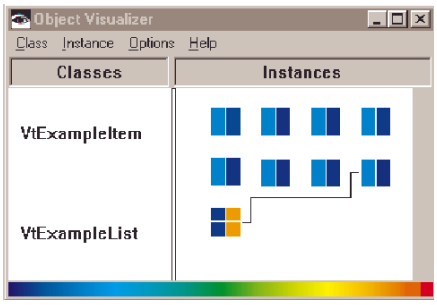


Figure 4.

10. Select the VtExampleList instance by clicking its center. A thin black border will surround it, indicating that it has in fact been selected. Then select **Browse Behavior** from the **Instance** menu of the Object Visualizer.

    A window opens showing the methods that have been active for that object. For each method, there is a count of the number of times it has been invoked, and the amount of time that has been spent executing that method. The color of an entry provides a relative indication of how active the method has been.

    The browser shows that *VtExampleList>>initialize:* has been the most active by far.
11. Open a behavior browser for the first VtExampleItem instance as well.

    This browser shows that *VtExampleItem>>shouldConnectTo:* has been most active of the VtExampleItem methods.

    If you want to understand the program execution more fully, open browsers on the application code, the methods *VtExampleList>>initialize:* and *VtExampleItem>>shouldConnectTo:* in particular, and see how the activity observed in the visualizer corresponds to the code.

12. Close the behavior browser windows opened above.
13. In the workspace opened earlier, evaluate `VtExampleList the bubbleSort`. Observe the behavior of the bubblesort computation, as the VtExampleList repeatedly asks the items to compare themselves with other items.

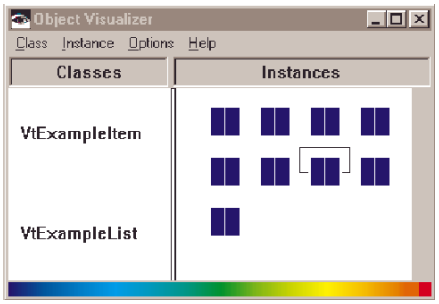


Figure 5.

14. Reset the visualizer display by selecting **Reset** from the **Options** menu.

    The boxes all become blue, as they were when they were first created.

    Now, we will visualize a rather crude breadth-first search, looking for a "heavy path", that is, one whose total heaviness is more than 10,000. (See the code for the exact definition; but note that the example is contrived for illustration rather than realism.)
15. Evaluate

    ```
    (VtExampleList the heavyPath: 10000 depth: 6) inspect
    ```

    .

    Watch as one VtExampleItem experiences a great deal of usage, a few others get some usage, and the rest are untouched. The heavily used node is the first one in the (sorted) list; the other heavily-used ones are presumably its successors. The heavy use of a few list items suggests the need for some improvements to the algorithm.

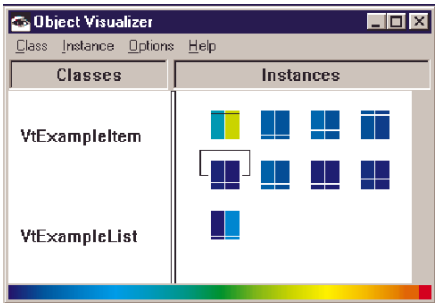


Figure 6.

16. If desired, slow the visualizer down by choosing the command **Set Delay Time** from the **Options** menu, and setting the slider to **1**. Rerun the visualization by evaluating the *heavyPath* expression again. You can see the paths from the root of the search.
17. End the visualization session as follows. Choose **Select All** from the **Instance** menu of the Object Visualizer, and then choose **Remove Instances** from the

Instance menu. Then choose **Select All** from the **Class** menu, and choose **Remove Classes** from the **Class** menu. Finally, close the Object Visualizer window.

Opening the Cluster View

The Cluster View provides a dynamic representation of object relationships. In order to visualize objects with the Cluster View, you must first add them to the Object Visualizer. You can choose to view either classes or instances of objects that are represented in the Object Visualizer.

To open the Cluster View, follow these steps:

1. Open the Object Visualizer, if it is not already open.
2. Add the classes you want to analyze to the Object Visualizer. (See "Selecting Classes to Visualize" on page 7 for more information.)
3. Select either the classes or the instances you want to visualize with the Cluster View. Select classes from the left-hand pane of the Object Visualizer window; select instances from the right-hand pane. You can select classes or instances in several ways:
   - Use mouse button 1 to select individual objects.
   - Use Ctrl+mouse button 1 to select multiple objects.
   - Use marquee selection (moving the mouse while holding mouse button 1) to select multiple objects.
   - Select **Select All** from the **Class** or **Instance** menu to select all classes or instances.
4. When you have finished selecting classes or objects, open the Cluster View in one of the following ways:
   - To visualize classes, select **Browse Clusters** from the **Class** menu.
   - To visualize instances, select **Browse Clusters** from the **Instance** menu.

   The Cluster View opens, showing the classes or instances you selected.

Analyzing Object Interaction with the Cluster View

In the Cluster View, each object is represented by a colored box containing the name of the class or instance (instance names consist of the class name followed by a number). When the Cluster View first opens, all of the objects you selected appear in the center of the window. (Not all of the objects are visible, since they are occupying the same screen position.)

As the application runs, the objects in the Cluster View drift away from the center, spreading out across the window. Simultaneously, objects that send messages to one another are mutually attracted. As you watch, you should see the objects in the application form clusters based upon the amount of message traffic that passes among them. Objects that remain far apart have little direct interaction; objects that cluster tightly together send messages to one another frequently.

The pop-up menu in the Cluster View window provides several other options that can help you analyze object interactions:

- Select **Show Message Path** from the pop-up menu to see a visual depiction of method calls from one object to another. (This option is selected by default.) The Cluster View shows method calls as lines that appear between objects and disappear as method calls complete.

- Select **Show Collaborations** to see a permanent visual indication of which objects interact with one another. Collaborations appear as solid lines between objects. Any objects that have interacted at any time during application execution are collaborators.
- Select **Remove Objects** to remove the selected object or objects from the Cluster View. (This option is available only after you have selected at least one object with the mouse.)

  Removing an object from the Cluster View does not affect the object itself; it only removes the object's visual representation in the Cluster View.
- Select **Reset** to move all of the objects back to the starting position at the center of the Cluster View. This option can help you to identify different clustering that might emerge during different parts of application execution.

  For example, objects that interact heavily during start-up might not continue to interact later. By resetting the objects, you can undo the clustering that emerged during start-up and see how the objects interact the rest of the time.

Example: Using the Cluster View

To see how the Cluster View displays object interaction, try the following example.

1. Open an Object Visualizer, and add the classes *VtExampleItem* and *VtExampleList*, as in "Example: Using the Object Visualizer" on page 9.
2. Evaluate the code: `VtExampleList new: 8`.
3. In the **Instances** pane, choose **Select All** from the popup menu.
4. In the **Instances** pane, choose **Browse Clusters** from the popup menu.
5. An Instance Cluster View appears. It contains a labeled rectangle for each instance. Initially, the rectangles are all positioned one over top of the other at the center of the window.

   If you wish, you may drag each instance apart from the others a bit using the mouse.

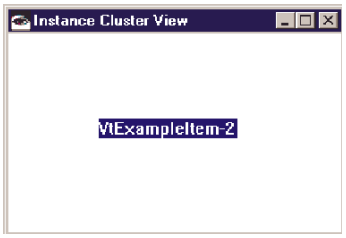


Figure 7.

6. Move the Instance Cluster View window so that it will not be hidden by the workspace you are executing code in.
7. Evaluate the code,

   ```
   VtExampleList the heavyPath: 10000 depth: 5
   ```

   .

   Watch as the items spread apart on the screen. The line flickering between items shows messages being sent. When the animation is over, a small number of instances will be clustered together near the center of the window. These are the instances which interacted heavily with one another. The rest, relatively unused, will be spread out further towards the edges of the window.

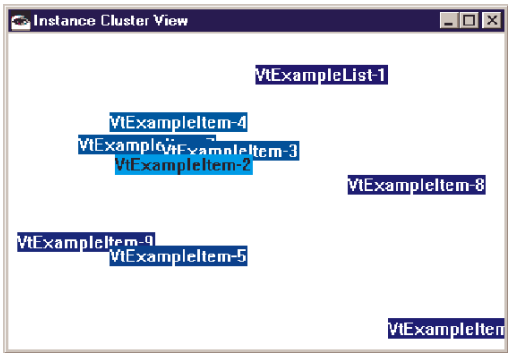


Figure 8.

8. Close the Instance Cluster View.
9. End the Object Visualizer session as in "Example: Using the Object Visualizer" on page 9.

Chapter 5. The Widget Scope

The Widget Scope is a tool that reveals objects ″behind the scenes″ of an interactive Smalltalk application.

A user simply clicks anywhere on the screen, and the Widget Scope lists the widget objects that produce that part of the display, as well as other objects that implement the function provided by the application.

Widget scoping in conjunction with visualizing and snooping (pages 7 and 21), constitutes an extremely powerful, yet practical and convenient, means of ″getting to the bottom″ of behavior of unfamiliar applications and frameworks.

Without these tools, a programmer trying to gain an understanding of how an application works (to fix it, to extend it, or to reuse parts of it) is faced with roundabout schemes involving searching for classes by pattern matching on their names, guessing at how to launch an application, and trying to get a debugger to open at opportune times.

Using the Widget Scope

To open a Widget Scope window, select **Open Widget Scope** from the **Visualization** menu of the **System Transcript**.

To reveal objects behind the scenes, you click the **Probe** button, and then click the area of the screen in which you are interested.

For example, to find out what's behind the scenes of a hierarchy browser, do the following:

1. In the **System Transcript**, select **Browse Hierarchy...** from the **Tools** menu.
2. In the displayed prompter, type `Object` and click **OK**.
3. In the **System Transcript**, select **Open Widget Scope** from the **Visualization** menu.
4. In the **Widget Scope**, click the **Probe** button. The cursor turns into a cross-hair.
5. Click in the center of the top-left pane of the hierarchy browser.

A list of objects then appears in the scrolled window of the Widget Scope. Objects are listed one per line, with each line containing a tag, such as `Hit` or `Parent`. The tag is followed by an object identifier which is the result of a *printString* message sent to the object. The tags provide an indication of why the object is listed. For instance, `Hit` indicates that the object is the one that was clicked with the probe; `Parent` indicates that the object is a parent of a preceding object in the list.

As well, each widget object can have a number of objects displayed in indented lines immediately below it. These are objects that hold callbacks on the widget. The indented line contains the name of the callback, and an object identifier.

Working with Objects in the Widget Scope List

To work with the objects listed in the Widget Scope, you select an object by clicking its line in the scrolled window of the Widget Scope, and then clicking one of the buttons at the bottom of the Widget Scope.

The **Flash** button lets the user flash the selected object in order to verify that it is in fact the object that the user thinks.

For example, after step 5 on page 17, you could:

1. Click the first line (which is the one for the object that was hit by the Probe operation).
2. Click the **Flash** button in the **Widget Scope**. The upper-left pane of the hierarchy browser disappears (!)
3. Click the **Flash** button again. The upper-left pane reappears. (Phew!)
4. Click the `Parent - CwForm(form)` line, and then click the **Flash** button twice.Watch which part of the display flashes, as you click the **Flash** button.
5. Click the `Parent - CwTopLevelShell(...)` line, and then click the **Flash** button twice.

(Note that you can use the pop-up menu of the Widget Scope scrolled window instead of the buttons at the bottom of the Widget Scope.)

The **Inspect** button opens an inspector on the selected object, the **Snoop** button lets the user snoop through the selected object (see Chapter 6), and the **Browse** button allows browsing of the class of the selected object.

The **CompEdit** button opens the VisualAge **Composition Editor** on the selected object (or its containing composite visual part) (assuming that the selected object is a VisualAge part — such parts usually appear in the Widget Scope scrolled window on lines containing strings like *Abt...View* e.g *AbtShellView*).

For example, you could:

1. Open a **Widget Scope** (as above).
2. In the **Widget Scope**, click the **Probe** button.
3. Click in the text area of the **System Transcript**.
4. In the scrolled window of the **Widget Scope**, click the first line: `Hit - CwText(text)`.
5. Click the **Snoop** button at the bottom of the **Widget Scope**. This opens a Snooper on the *CwText* object.
6. Click the **Probe** button of the **Widget Scope** again.
7. This time, click in the text area of the **Snooper** window.
8. In the scrolled window of the **Widget Scope**, click the last line. It should contain something like `...Callback - an AbtShellView(...)...`.
9. Click the **CompEdit** button at the bottom of the **Widget Scope**.

After a bit of a delay, a Composition Editor should appear showing Snoop's internal structure.

Don't touch it!!! Close it right away.

Adding Objects to the Visualizer Display from the Widget Scope

Using the Widget Scope, you often find objects whose long-term behavior would be interesting to observe. These objects can be added to the Object Visualizer display directly from the Widget Scope. Simply select the objects of interest, and click the **Visualize** button. If the Object Visualizer is not already open, it will be opened. Then the objects of interest will appear.

For example, you could:

1. Open a **Widget Scope**, as above.
2. Open a **Hierarchy Browser** on *Object*, as above.
3. Click **Probe**.
4. Click the upper-left pane of the hierarchy browser.
5. In the scrolled window of the Widget Scope, click the last line. It should contain something like `an EtClassesBrowser`.
6. In the Widget Scope, click the **Visualize** button.

   An EtClassesBrowser object will appear in the Object Visualizer display.
7. In the upper-left pane of the hierarchy browser, double-click the *Object* line.

   A large amount of activity is apparent in the visualizer display.
8. In the visualizer display, click the box representing the EtClassesBrowser object.

   A black outline appears around it to indicate that it is selected.
9. From the **Instances** menu of the visualizer, select **Browse Behavior**.

   Now you can see which methods do a lot of work as the class hierarchy is expanded and contracted in the hierarchy browser pane.

To observe more of the activity that occurs during expanding and contracting, you could select lines in the Widget Scope scrolled window, and click the **Snoop** button. This would open a Snooper (see page 21) which you could use to find more of the related objects. Then from the Snooper you could select objects and press the Snooper **Visualize** button to view the newly discovered objects.

Finding Out How an Application is Launched

To find out how an application that is normally accessed through a menu is actually launched, you can probe into menus.

Note: For this procedure to work, you must have already made at least one actual use of the menu entry of interest. Also, for menus produced using the VisualAge builder, you should instead use **CompEdit** to open a composition editor in order to see what actions result from a menu entry. Finally, in certain circumstances, probing the menubar as described below actually results in a normal probe of the menubar widget, rather than a probe of the associated menus.

To find out, for example, how a hierarchy browser is started up (perhaps for purposes of being able to start it under control of a debugger), you could:

1. Open a **Widget Scope** as above.
2. Click the **Probe** button.
3. Click the **Tools** button in the **System Transcript** menu bar.
4. In the resulting prompter, double-click **Tools**, and then double-click **Browse Hierarchy**.

5. In the scrolled window of the **Widget Scope**, click the `Menu Button - ...` line
6. Click the **Inspect** button at the bottom of the **Widget Scope**.
7. In the resulting inspector, click *selector*, and see that the message *#openHierarchyBrowser* is sent whenever the menu button is pushed. Then click **owner**, and see that the message is sent to the class *EtDevelopment*. Finally, select **Browse Class** from the **Variables** menu of the inspector, and peruse the *openHierarchyBrowser* public class method in the ET-Internal category of the class.

Congratulations! In less than a couple of minutes, you too can now launch hierarchy browsers!

Chapter 6. The Snooper

The Snooper is a powerful tool that provides a hierarchical textual view of a hierarchy of objects. Individual objects discovered using the Snooper may then be added directly into the views provided by the Object Visualizer (page 7).

The Snooper also includes a set of buttons that provide a simple and direct means for opening other tools of the IBM Smalltalk environment, such as visualizers, browsers, and inspectors, on objects of interest. In addition, the Snooper provides a code evaluation facility, a workspace, object searching, and limited views of its own.

The Snooper can be used in combination with the other tools of the environment to provide a powerful facility for investigating object structures and quickly gaining an understanding of program behavior.

Snooping Basics

One way to open a Snooper on an object is to send it the *snoop* message.

The basic Snooper display is a list containing one line per object. Each line consists of a name and a value. The value of a composite object is shown simply as `a <ClassName>`. The value of a primitive object is simply printed. The first line in the list is for the object being snooped. The name on the first line is *self*, and the value is that of the object being snooped.

The subsequent lines in the list are for the components of the object being snooped. By default, these lines are ordered alpahbetically according to name. The names of the components are indented. In some cases, the sub-components of a component will be shown immediately below it. Their names will be indented even farther. Thus, the Snooper uses indentation to show structure.

Double-clicking a line of the Snooper display will expand an object, showing its components on subsequent lines. If the object's components are already shown, double-clicking contracts the object, removing its components from the display.

Objects frequently have components that you are not currently interested in. You may remove lines from the display by selecting them and choosing **Hide Selected Items** from the **Hide** menu. **Reveal Hidden Items** will return them to the display.

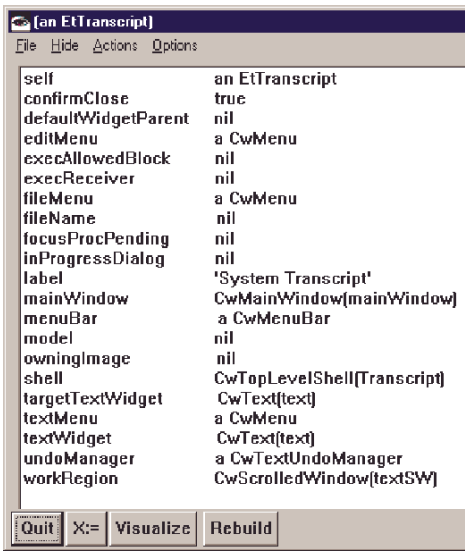
If you are snooping many objects of similar classes, you may know that you will never care about certain instance variables. You may tell the Snooper to always hide those variables by the selecting them and choosing **Hide Selected Names** from the **Hide** menu. **Edit Automatically Hidden Names** will let you tell the Snooper to show those instance variables again.

Example: Snooping the System Transcript

The running example in this chapter deals with snooping around the System Transcript object.

Opening the Snooper

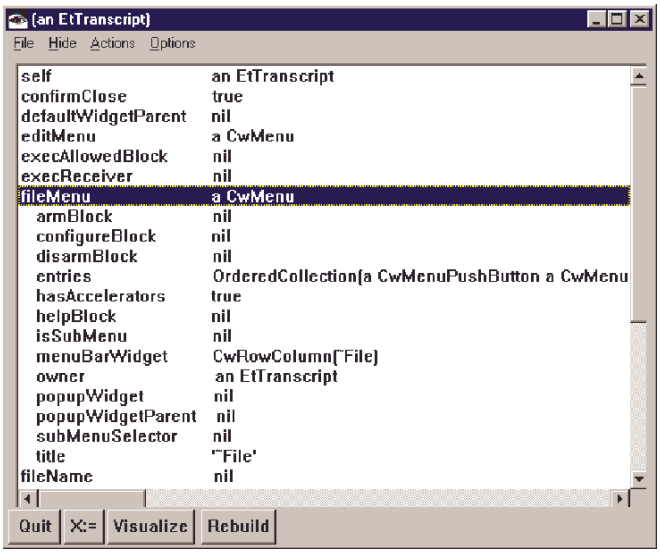
1. In any workspace (e.g, the System Transcript itself) type `Transcript snoop`.
2. Select the words `Transcript snoop`.
3. Right-click, and choose *Execute* from the pop-up menu.
4. A Snooper will appear.



The Snooper shows the instance variables of the Transcript. For example, there is an instance variable called *confirmClose* which is currently *true* . There is another variable called *fileMenu* which is *a CwMenu* , which is a composite object.

Expanding and Contracting Composite Objects

1. Double-click the *fileMenu* line to see the components of the *fileMenu*.



For example, the fileMenu has an instance variable *title*, whose value is the string *'File'* .

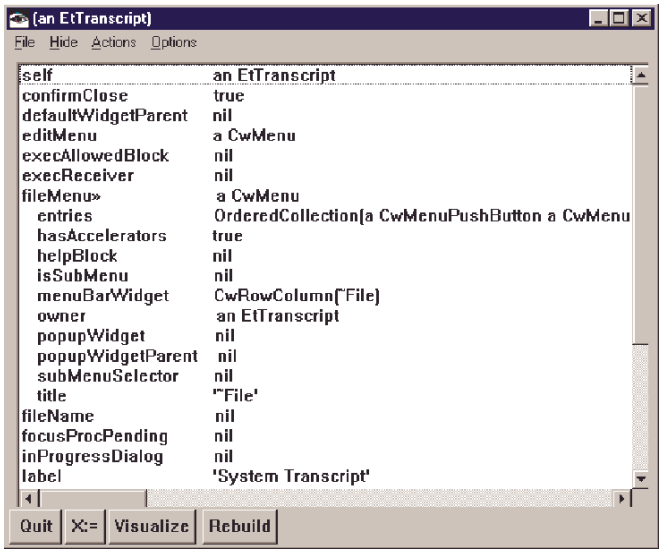
2. Double-click the *fileMenu* line a second time, to contract the *fileMenu*.

The Snooper will return to its appearance as shown in 22. The components of the *fileMenu* are no longer shown.

Advanced Hiding and Unhiding

1. Double-click the *fileMenu* line a third time, to redisplay the components of the *fileMenu*.
2. Highlight the first three items indented under *fileMenu*, that is: *armBlock*, *configureBlock*, and *disarmBlock*. (You may click the first item and drag through to the last, or control-click on the three individual items).
3. From the **Hide** menu, select **Hide Selected Items**. (Or, press control+h).

   The three lines vanish from the display. A marker appears by *fileMenu*, indicating that it has hidden children. (Depending on which platform you are using, the marker might be a right-pointing triangle, an exclamation mark, or some other symbol.)



4. From the **Hide** menu, select **Reveal Hidden Items**. The three lines reappear, as shown in 22.
5. Repeat step 3 to hide the three lines.

   Hiding items is independent of contracting/expanding them. Once an item has been hidden, it will stay hidden until it is revealed.
6. Double-click the *fileMenu* line once to contract it, and then double-click it again to re-expand it. Note that the three hidden items are still hidden.

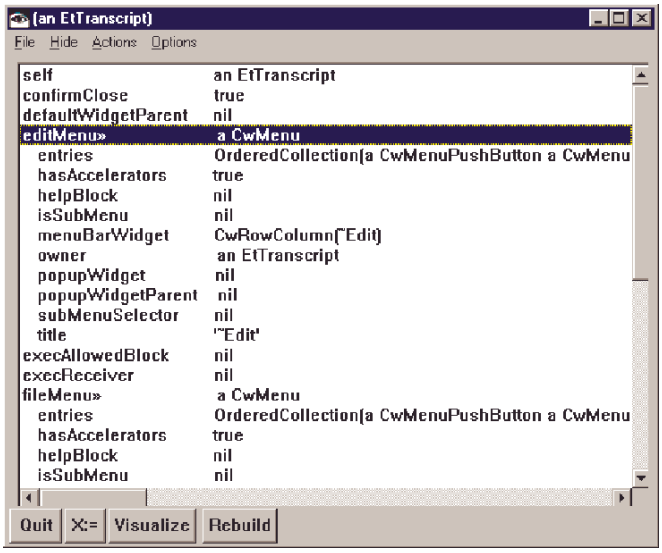
   If you have many things hidden, **Reveal Hidden Items** redisplays them all. If you want finer control, you can reveal only a few hidden items.
7. Select the *fileMenu* line. From the **Hide** menu, select **Reveal+Expand Selected** (or press control+r).

   The three hidden lines reappear, returning the display to its appearance as shown in 22.

   You may hide particular instance variables, so that they do not show for any object.
8. Select the *armBlock*, *configureBlock* , and *disarmBlock* lines.
9. From the **Hide** menu, select **Hide Selected Names**. (Or, press control+n). The three lines are hidden, as in 23.
10. Double-click the *editMenu* line.

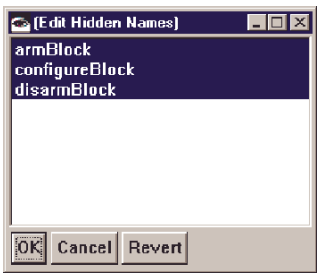
Note that the editMenu is another CwMenu, and so it has an *armBlock*, *configureBlock*, and *disarmBlock*. But they are hidden, because their names are hidden.

This is convenient for reducing clutter when exploring complicated structures. It is often the case, e.g. when exploring VisualAge parts, that there are a large number of instance variables, many of which are not of interest at a given time.



The hidden lines may be unhidden as in step 7 on page 23. However, this can only be used to reveal hidden items for objects which have already been expanded. The Snooper will continue to hide the named instance variables any time it subsequently expands an object. If you want to make the three instance variables be visible by default again, use the **Edit Automatically Hidden Names** selection in the **Hide** menu.

11. From the **Hide** menu, select **Edit Automatically Hidden Names**. A window listing the hidden names will appear.



12. Control-click the three names in the window to unselect them. Press **OK**.
13. In the snoop window, the three instance variables have reappeared. If you expand *textMenu*, you will notice that they are no longer hidden.

Forms of Display

The Snooper displays most objects by simply showing instance variable names and values. For the standard collections, however, more effective displays are possible,

Indexable collections (Arrays, OrderedCollections, etc) are displayed via index. The first element is given the name *at: 1*, the second is called *at: 2*, and so on.

Dictionaries are displayed by key. (By default, the keys are sorted alphabetically.) The elements of the dictionary are named *at: key*.

Sets and other collections without a definite order are displayed in some arbitrary order. Since the elements of a set are not distinguishable, the Snooper just calls each one *el:*.

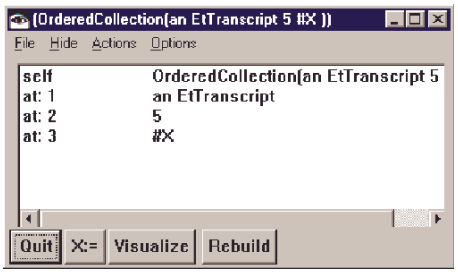
Strings and symbols are collections, but they act like constants and cannot be expanded. (Expanding them would give a long and not terribly useful batch of characters.)

Examples: Displaying Collections

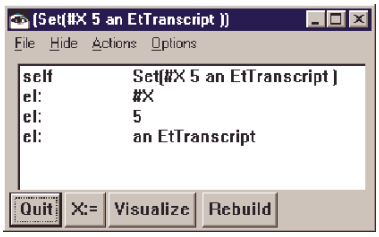
1. 1. In a workspace, type

   ```
   (OrderedCollection with: Transcript
        with:5 with: #X) snoop
   ```

   .
2. Select and evaluate the text.

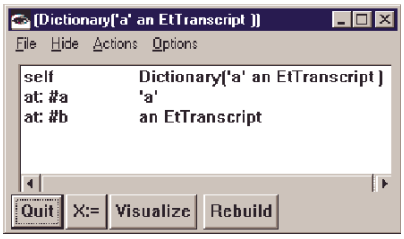


3. Replace `OrderedCollection` with `Set`, and select and evaluate the text again.



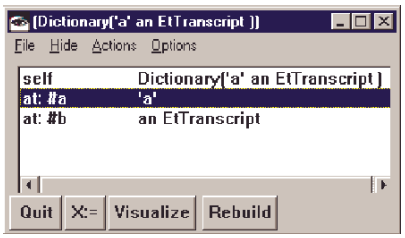
4. Type and evaluate

   ```
   |d|
   d := Dictionary new.
   d at: #a put: 'a'.
   d at:#b put: Transcript.
   d snoop.
   ```



   If you try to expand a line containing an object that has no children, a marker appears by its name. (Depending on which platform you are using, the marker might be an omega, a left angle bracket, or some other symbol.)

5. Double-click the *a* line to try to expand the string *'a'*.



Adding Objects to the Visualizer Display from the Snooper

Objects discovered using the Snooper can be added to the Object Visualizer display. This allows the long-term behavior of interesting objects to be observed as program execution proceeds.

To add an object to the Visualizer display, simply click a line in the Snooper display, and then click the Snooper **Visualize** button. The object represented by the line will then appear in the Visualizer display. See page 19 for a description of using visualization of individual objects.

Opening Other Tools from the Snooper

Other tools of the development environment can be opened on objects displayed by the Snooper.

Browse, Composition Edit.

You may open browsers or composition editors on objects from a Snooper.

1. Evaluate `Transcript snoop` in a workspace window to open a Snooper on the transcript, as in 22.
2. Select *fileMenu*.
3. From the **Actions** menu, select **Browse**. (Or, use the popup menu, or type control+b).

   An ordinary code browser opens up.
4. From the **Actions** menu, select **Composition Edit**.

   A composition editor on the *CwMenu* class appears.

Snoop

You may open new Snoopers on objects in a Snooper.

1. Select *fileMenu*.
2. From the **Actions** menu, select **Snoop**. (Or, use the popup menu, or type control+s).

   Another Snooper appears, inspecting the value of the file menu.

Inspect and **Basic Inspect**

You may invoke inspectors from the Snooper. This may be preferable in some cases, e.g., very large collections which you do not want to display in full.

1. Select *fileMenu*
2. From the **Actions** menu select **Inspect**.

An ordinary inspector on the file menu appears.

Snoop All References

You may snoop the collection of all references to a given object.

1. In a workspace, type `78 snoop`.
2. Select the *self* line.
3. From the **Actions** menu select **Snoop All References**.
4. A new Snooper opens, showing all the objects in your image that refer to the number 78.

Watch Expressions

The Snooper allows you to attach a **watch expression** to an object; the expression's value will be displayed on its own line, indented beneath the object's line, just as if the expression were an attribute of the object.

1. Select the line of the table that the expression will be evaluated upon.
2. From **Hide** select **Watch Expression**. (Or, select **Watch Expression** from the popup menu.)

   A **(Code)** window appears.
3. (Optional) Check that you are attaching the block to the right object, by looking at the line on the window displaying its value, and the line beneath it displaying its type.
4. Fill in the text block labelled **Show value of:** with the text of a one-argument Smalltalk block, computing the desired value. The text block is initialized with the start of a one-argument block.
5. (Optional) Type a name that the block will appear under. The name is initially `block`. Hint: Starting the name with a tilde () will keep it from looking like an instance variable.
6. (Optional) Make your block more resistant to updates of the underlying object by specifying the class that the method should be applied to, and leaving **Only evaluate on specimens of that class** selected.

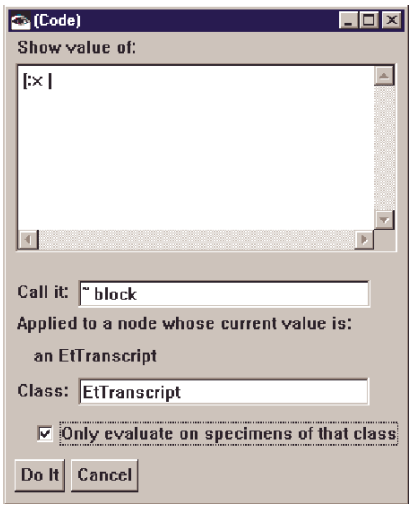
   Or, make your block more flexible by deselecting **Only evaluate on specimens of that class**. This will evaluate the block on values of all classes (e.g., if you have attached the block to an instance variable, and change the value of that variable to one of another class).

Example: Watch Expression

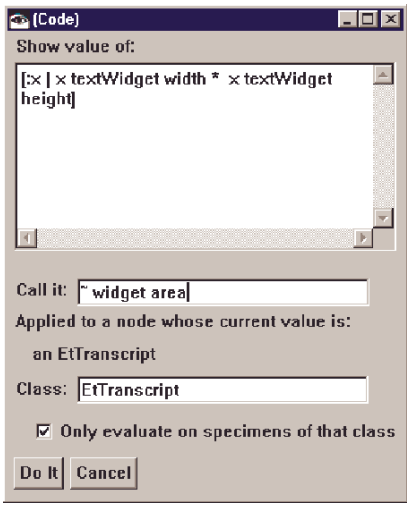
Suppose that you want to know how big the transcript window is, in pixels. You'd like this number to be easy to update. Unfortunately, it's not an instance variable of the transcript window or anything else. So, the Snooper allows you to write a block of code and display its result.

1. Open a Snooper by evaluating `Transcript snoop`.
2. Select the *self* line.
3. From **Hide** select **Watch Expression**. (Or, select **Watch Expression** from the popup menu.)

4. A window titled **(Code)** appears

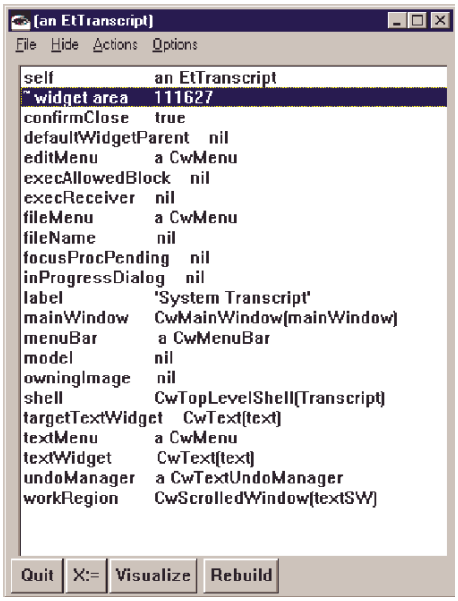


5. In the **Show value of:** box, complete the text to read *[:x | x textWidget width * x textWidget height]*
6. Modify the **Call it:** field to read **widget area**.



7. Click on **Do It**.
8. Observe that a new name **widget area** has appeared, just as if it were an instance variable. The actual value will vary, depending on how big your

Transcript window actually is.



9. Change the size of your Transcript window. Note that the value of the widget area in the Snooper does **not** change — the Snooper has no way of knowing that it should be updated.
10. Double-click any line of the Snooper, or press the **Rebuild** button. Note that the value of the widget area is updated to reflect the current size.

Searching Objects

The Snooper allows you to search through a hierarchy of objects for values satisfying a predicate (a boolean expression). The search is a depth-limited depth-first search. There are many ways of tuning the search.

1. Press the **Search** button, opening a **(Spy Search)** window.
2. In the **Search For** tab of the **(Spy Search)** window, type a one-argument block which answers `true` when applied to an object that you are looking for, and `false` otherwise. If your block fails on a given argument (e.g., if you call `x accelerator` and some `x` does not respond to `#accelerator`), the block is considered to return `false`.
3. Select the depth of the search by picking a radio button at the bottom of the window. The depth 2 is selected by default. Depths higher than 6 are available by selecting the **+** radio button, then using the spinner. The search engine is fairly slow, so large depths are not recommended.
4. (Optional) Prune the search. Select the **Prune** tab. Fill in the block of code titled **Only search under values which satisfy**. This block is a two-argument block, taking (1) the attribute name being searched, and (2) the value of that attribute. When the block returns true, the search engine will look beneath that block. Pruning can speed up the search — or can slow it down, because testing values to be pruned (even with a very simple test) is expensive.
5. (Optional) On the **Options** tab, set the options that you want for the search.
   - If you only want to open the hierarchical structure to show all the items that you found, but not change the currently selected items, deselect **Select all matching items**.
   - If you do not want to hide all the items which do not meet your search criterion, deselect **Hide non-matching items**.

- If it will suffice to find only one item matching your requirements, select **Stop searching on first match**. This can speed searches up tremendously.
- If you want to see a mediocre kind of progress indicator on the System Transcript, select **Show Progress On Transcript**.
- If you don't want to contract the tree before starting the search, deselect **Contract tree before searching**.
- If you want to search under just the selected items, instead of the *self* item, select **Search under Just the Selected Items**.
- If you want to flag the items that were found, select **Mark Found Items with**, and then type or select a symbol in the combo box.
- If you are searching a structure with many common sub-objects, select **Don't re-search objects** (There is a substantial overhead for keeping track of which objects have been searched).

6. When you are ready to search, press the **Search!** button.

   The Snooper keeps a list of the searches you have performed. If you want to repeat a search, possibly with modifications:
7. Press the **Prev** button (or **Next** button) until the search that you want to redo or modify appears.
8. (Optional) Press the **Copy** button to create a new copy of that search, so that changes you do will not modify the original one.
9. Update the controls as above, and perform your search.

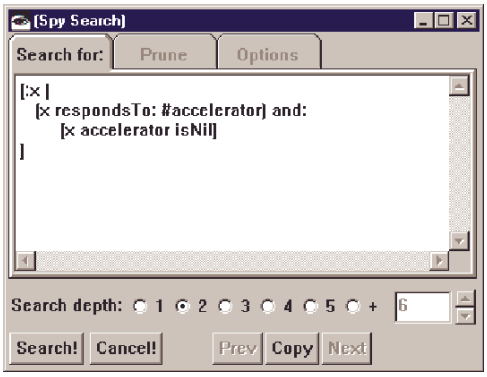
Example: Searching

Suppose that you want to find and mark all the buttons in the Transcript which have no accelerator key.

1. Open a Snooper on the Transcript window.
2. Press the **Search** button.
3. As the **Search for:** criterion, type

   ```
   [:x |
    (x respondsTo: #accelerator) and:
        [x accelerator isNil]
   ]
   ```

   (Note: the `(x respondsTo: #accelerator)` is not strictly necessary but may result in a more efficient search.)

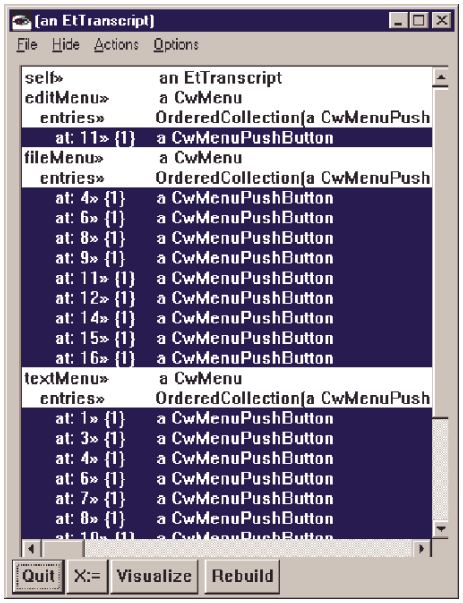


4. On the **Options** tab, select **Mark Found Items with** and select a symbol from the pulldown.
5. Press **Search!**

The Snooper display contracts to only the *self* line. This is because there are no such objects to depth 2 in the Transcript window. We will try again, increasing the depth.

6. Press **Search**.
7. In the **(Spy Search)** pane, press **Prev** once, and then **Copy**.
8. Set the **Search Depth** to 4.
9. Press **Search**, and wait somewhat longer than before.

   The Snooper shows up with certain values (those push buttons with no accelerator) selected. They are also marked with the symbol you selected. (If you later change the selection, the marks will remain.)



10. Select the first line found, and press control+r (or, from the **Hide** menu select **Reveal and Expand Selected**) and confirm that indeed that button's accelerator is nil.

Assignments

Objects displayed by the Snooper can be assigned to global variables. These objects can then be used in expressions in any workspace or in a built-in evaluation pane of the Snooper.

By evaluating expressions referring to an object, sending it various messages, observing their effect, and inspecting returned values, a user can learn more about objects which have been discovered using the basic snooping operations.

The Snooper uses the global variable names *X*, *Y*, *Z*, and *S*. It uses single-letter variable names for convenience and to avoid conflicts (most programs are likely to use more descriptive names for their global variables).

To assign an object to `X`:

1. Select the object in the Snooper.
2. Press the **X :=** button. (Or, press alt+X, or from the **Actions** menu or the popup menu select **X := selected item**).

   The variable *X* in any workspace will now refer to the object that was selected.

To capture up to three objects as X, Y, and Z:

1. Select up to three objects in the Snooper.
2. From the **Actions** menu select **X, Y, Z := selected items**.

   The global variable *X* will now refer to the first selected object; *Y* will refer to the second selected object, if there were two or three (otherwise it will be undefined); and *Z* will refer to the third selected object, if there were three (otherwise it is undefined).

To assign any number of objects to an array called X:

1. Select the items.
2. From **Actions** select **X := Array of selected items**.

   X will now refer to an array containing all of the selected objects.

To set the Snooper so that whenever an object is selected, it is assigned to S:

1. From the **Options** menu, select **S := Single Selection**. (Or, press control+alt+S).

(Note: This is very dangerous — it is far too easy to get confused about what item is currently referred to by S — but it is useful on occasion).

The Evaluation Pane

The Snooper includes an evaluation pane that allows convenient entry and recall of expressions and their values. Ordinarily the pane is not shown, to avoid taking up space.

To show the evaluation pane:

1. From **Actions**, select **Toggle Evaluation Pane**. (Or, press alt+v).

The evaluation pane actually is two panes. The upper pane is a table showing some expressions and their values; initially, it starts with just the global variables X, Y, Z, and S. The lower pane is a fairly conventional workspace, with a few minor differences. As you evaluate expressions in the workspace pane, they and their values will appear in the table pane.

Evaluating Expressions

To evaluate an expression:

1. Type the expression in the workspace pane.
2. Press control+E. (Or, select **Evaluate** from the popup menu).

   The value and the expression appear in the table pane.

Note: Unlike ordinary workspaces, it is not necessary to select the expression to be evaluated; control+E evaluates the entire workspace contents by default.

Variations:

- The **Evaluate Selection** command (control+alt+E) evaluates only the selected area of the workspace pane.
- The **Eval & Snoop** (control+Q) command evaluates the whole pane, and opens a Snooper on the value.
- The **Eval & Snoop Selection** (control+alt+Q) command evaluates the selected area and opens a Snooper on the value.

Editing commands are available on the popup menu of the workspace pane, or via keyboard shortcuts.

- **Cut** (control+x) is standard.
- **Paste** (control+v) is standard.
- **Copy** (control+c) is standard.
- **Select All** (control+a) is standard.
- **Clear** (control+l) clears the entire workspace.
- **Browse** (control+b) opens a browser on the class whose name has been selected.

Several commands are available from the popup menu of the table pane.

- **Snoop** opens a Snooper on the value in the selected line.
- **Edit** copies the text of the expression in the selected line to the workspace, suitable for modification and re-evaluation.
- **Do It** re-evaluates the expression in the selected line. The result of the expression is ignored. This command is most useful when the expression has side effects.
- **X :=** assigns the value in the selected line to `X`.
- **Clear lines** deletes the values stored in the table.

Saving Expressions

Ordinarily, as new expressions are evaluated, the old ones sink towards the bottom of the table. (By default, the global variables `X, Y, Z` do not sink.)

If you want to keep a value or expression readily available:

1. Select the line.
2. Select **Keep Line Near Top** from the popup menu of the table pane.

To reverse the process:

1. Select the line.
2. Select **Let Line Sink**.

It is sometimes useful to have a value or expression available in **all** the Snooper evaluation panes.

1. Select the line.
2. **Save Expression**

   Whenever you open a new Snooper, that expression and value are present in its list of available values.

To reverse the process:

1. Select the line.
2. Select **Unsave Expression**

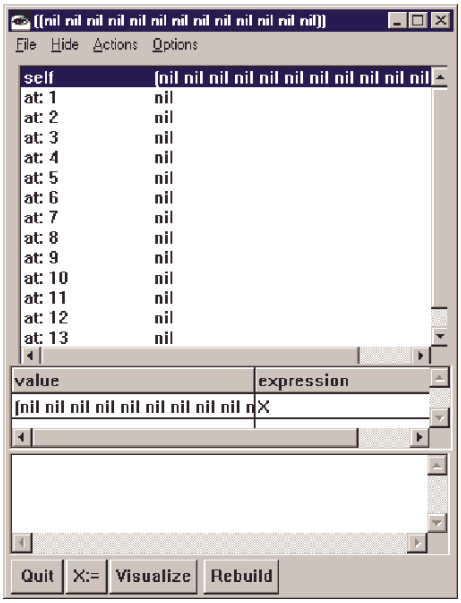
   The expression will no longer appear in new Snoopers.

   Similarly, the **Unsave Everything** command releases all the saved expressions.

Example of Working with Variables

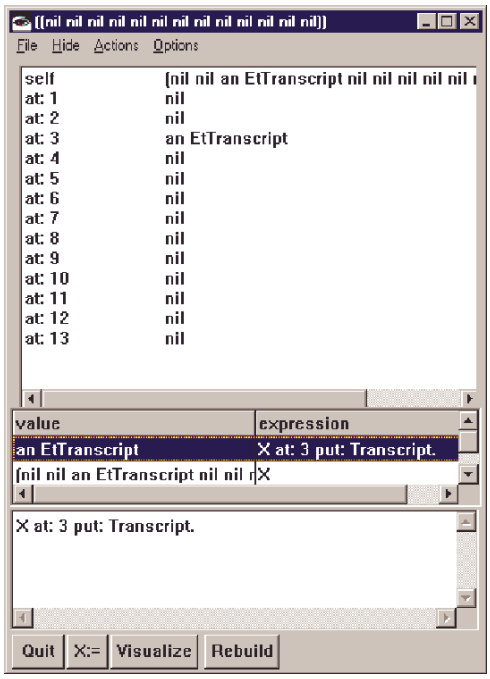
We will manipulate an array in several ways, showing off the features of the evaluation pane.

1. In a workspace, type and evaluate `(Array new: 13) snoop`.
2. In the resulting snoop window, select the *self* line.
3. Press alt+X. (Or, push the **X:=** button, or from **Actions** or the popup menu select **X := Selected item**). This sets the global variable `X` to the array.
4. Press alt+V. (Or, from **Actions** select **Toggle Evaluation Pane**).



5. Click on the workspace pane of the Snooper.
6. Type `X at: 3 put: Transcript`.
7. Press control+E. (Or, from the popup menu select **Evaluate**) to evaluate the expression.

8. Press the **Rebuild** button.



9. Clear the workspace pane (control+l, or select **Clear** from the popup menu.
10. Type `X select: [:v | v notNil]`, and evaluate it.
11. Note that a one-element array has appeared in the "value" column, and the expression you evaluated in the "expression" column of the table pane.
12. Clear the workspace pane. Type `X size`, and evaluate it.
13. Note that 13 has appeared as the first value in the table pane.
14. Scroll the table down. Note that the value and *select:* expression from step 10 are further down in the table.
15. Select that line.
16. From the popup menu select **Edit**.
17. Note that `X select: [:v | v notNil]`, the expression from step 10, appears in the workspace pane.
18. Replace `select:` by `reject:`, and evaluate the resulting expression.
19. Note than an array of *nil*'s has appeared as the first element in the table, and the *13* has dropped to a lower position.

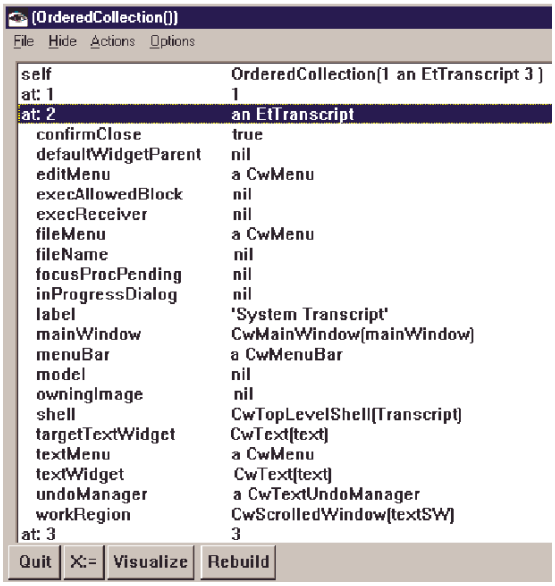
Rebuilding the Snooper

Occasionally, objects that you are snooping may get updated, without the Snooper being aware of the fact. The **Rebuild** button brings the Snooper back up to date, and attempts to keep the display in the same configuration as much as possible. That is, the Snooper attempts to keep the same parts of the object hierarchy expanded, as long as they still exist when the rebuild is done.

In this example, you will create an OrderedCollection and snoop it while it is being modified.

1. In a workspace, type and evaluate `OrderedCollection new snoop`.
2. In the Snooper showing that collection, select the *self* line, and push the **X :=** button.

3. In a workspace, type and evaluate `X add: 1; add: Transcript; add: 3`. (The values displayed in the Snooper might not change at this point).
4. Press the **Rebuild** button in the Snooper. Note that the current value of the collection appears in the Snooper.
5. Double-click the *at: 2* line to expand the Transcript.



6. In the workspace, type and evaluate `X at: 1 put: 'red fox'`.
7. Press **Rebuild** in the Snooper. Note that the current value of the collection appears, and that the Transcript is still expanded.
8. In the workspace, type and evaluate

```
X at: 3 put: (X at: 2).
X at:2 put: 'spotty cheetah'.
```

9. Press the **Rebuild** button. Note that none of the elements of the list are expanded; it changed too much to keep it expanded.

.

Snooping as a Debugging Aid

It is often helpful to put *snoop* commands in programs that you are trying to debug. The Snooper provides a few features to assist with that.

- By default, the title of a snoop window is an indication of the place from which the snoop was invoked, plus the value being snooped. For example, if you call snoop from the *bar* method of class *Foo*, on an object that prints as `baz`, then the window title will be `(Foo>>bar---baz)`.
- You may control the title of the snoop window with the *#snoop:* method. For example, `x snoop: 'x'` opens a snoop window titled simply `x`.
- *snoop* and *snoop:* answer the receiver, making them convenient to use in complex expressions.

  For example, if you are trying to understand the expression

```
aFamilyTree oldestLivingMember favoriteAunt
   coatOfArms = self coatOfArms
```

  you could insert a snoop call without restructuring your code:

```
aFamilyTree oldestLivingMember favoriteAunt
   snoop coatOfArms = self coatOfArms
```

or,

```
(aFamilyTree oldestLivingMember favoriteAunt
 snoop: 'favoriteAunt') coatOfArms = self coatOfArms
```

(Note that parentheses are required in the second expression.)

The Snooper also records the stack trace, and allows you to browse any of the last dozen or so methods called before the Snooper was invoked.

1. From **Actions** select **Show Stack Trace**.
2. A **Stack Trace** window pops up, showing the names of the last several methods called. Usually one or two methods from *SpyInspector* class — that is, from the Snooper itself — are at the top of the list, and your method is second or third.
3. Double-clicking a line of the **Stack Trace** window opens a browser on that method.

Options and Display Controls

There are several options controlling what is displayed, and how it is displayed by the Snooper.

To control the space used for names, select **Name Width** from the **Hide** menu, and then one of the percentages (10% - 60%). 30% is the default. 60% will give a great deal of space for names, and may be helpful for viewing deeply nested structures. 10% makes as much space as possible available for values.

The **Options** menu includes several toggles.

- **S := Single Selection**.

  This sets the Snooper into a mode in which, whenever a single item is selected, it is also assigned to the global variable S. (When zero, two, or more are selected, the variable is unassigned). This is quite dangerous, but useful in conjunction with the **Mark Appearances of Variables** mode.
- **Allow Error Prompters**

  Under some situations, the Snooper has the choice of presenting an error message (the default), or silently ignoring the error (which may be set with this option).
- **Show Expansion Markers**

  The symbols indicating the presence of hidden children, or expanded objects with no children, may be turned off with this toggle.
- **Show Classes**

  It is often useful to see the classes of the displayed objects, not just their values. This command toggles whether or not they are displayed.
- **Sort**

  By default, the Snooper shows instance variables in alphabetical order. If this is not appropriate, you may turn sorting off.
- **Rebuild Tree On Refresh**

  If you are snooping a data structure that is constantly being mutated by other processes, you will want to rebuild it frequently.This command causes the tree to be rebuilt in detail whenever it is manipulated by the Snooper. It is, of course, fairly slow.
- **Mark Appearances Of Variables**

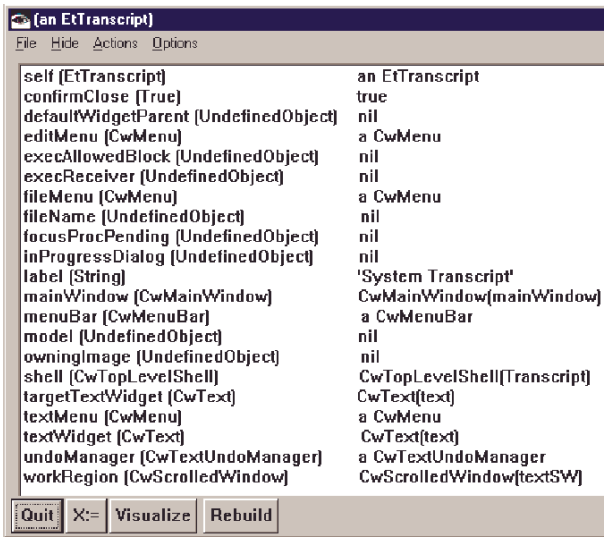
This puts markers on the tree whenever a value is the same object as one of the global variables `X, Y, Z, S`. This is useful for telling when two values that print the same are actually the same object. In conjunction with **S := Single Selection**, it can be used to probe the sharing structure of objects.

- **Save Options**

  Saves the current settings of the options, as the defaults for new Snoopers.

Example: Setting Options

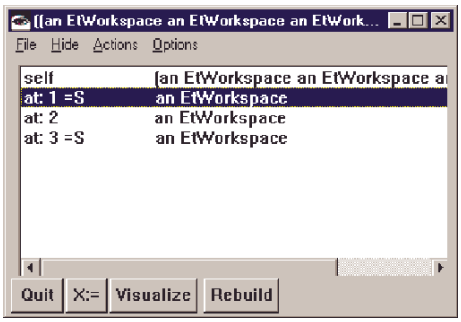
1. Open a Snooper on the Transcript.
2. From **Options** select **Show Classes** (or press Alt+S).
3. Note that all the classes of objects are now visible.



4. Press Alt+S again to return to the original state.
5. In the Transcript, type

```
| a b |
a := EtWorkspace new.
b := EtWorkspace new.
(Array with: a with: b with: a) snoop
```

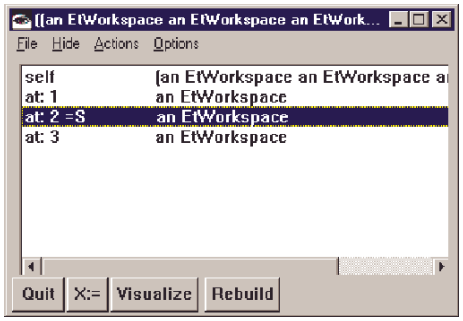
6. From **Options** select **S := Single Selection** (or, press control+alt+S).
7. From **Options** select **Mark Appearance Of Variables** (or, press alt+A.
8. Select the *at: 1* line. (This has the effect of assigning that value to S.)



Note that a marker **=S** appears next to the *at: 1* and *at: 3* line. This indicates that the values in those lines are identical to the global variable S. In particular, they are the same *an EtWorkspace*.

Also note that no **=S** marker appears on *at: 2*. This indicates that its value is a different *an EtWorkspace*.

9. Now, select the *at: 2* line. (This has the effect of assigning its *CwText(text)* value to S).



Note that an **=S** marker appears next to the *at: 2* line, but not the *at: 1* and *at: 3* lines.

10. Press control+alt+S to turn off **S :=** mode.
11. Press alt+S to turn off **Mark Appearances of Variables** mode.