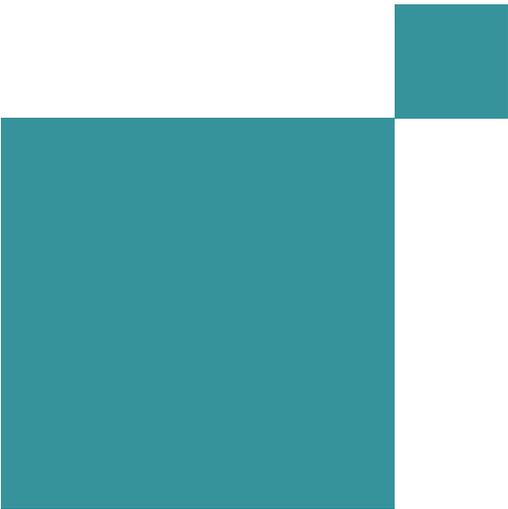


Cincom Smalltalk[™]



Tool Guide

P46-0147-01

Copyright © 1995–2007 by Cincom Systems, Inc.

All rights reserved.

This product contains copyrighted third-party software.

Part Number: P46-0147-01

Software Release 7.5

This document is subject to change without notice.

RESTRICTED RIGHTS LEGEND:

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

Trademark acknowledgments:

CINCOM, CINCOM SYSTEMS, the Cincom logo and Cincom Smalltalk logo are registered trademarks of Cincom Systems, Inc. ParcPlace and VisualWorks are trademarks of Cincom Systems, Inc., its subsidiaries, or successors and are registered in the United States and other countries. ObjectLens, ObjectSupport, Cincom Smalltalk, Database Connect, DLL & C Connect, COM Connect, and StORE are trademarks of Cincom Systems, Inc., its subsidiaries, or successors. ENVY is a registered trademark of Object Technology International, Inc. All other products or services mentioned herein are trademarks of their respective companies. Specifications subject to change without notice.

The following copyright notices apply to software that accompanies this documentation:

VisualWorks is furnished under a license and may not be used, copied, disclosed, and/or distributed except in accordance with the terms of said license. No class names, hierarchies, or protocols may be copied for implementation in other systems.

This manual set and online system documentation copyright © 1995–2007 by Cincom Systems, Inc. All rights reserved. No part of it may be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior written consent from Cincom.

Cincom Systems, Inc.

55 Merchant Street

Cincinnati, Ohio 45246

Phone: (513) 612-2300

Fax: (513) 612-2000

World Wide Web: <http://www.cincom.com>

Contents

About This Book ix

Overview	ix
Audience	ix
Conventions	x
Typographic Conventions	x
Special Symbols.....	x
Mouse Buttons and Menus	xi
Getting Help	xi
Commercial Licensees.....	xi
Non-Commercial Licensees	xii
Additional Sources of Information	xiii
Smalltalk Tutorial	xiii
Online Help	xiii
VisualWorks FAQ	xiv
News Groups	xiv
VisualWorks Wiki	xiv
Commercial Publications.....	xiv

Chapter 1 System Browser

Browser Navigator	1-4
Package View	1-4
Parcel View	1-4
Hierarchy View	1-4
Class / Name Space View	1-4
Instance, Class, and Variable Views	1-5
Icons in the Navigator	1-5
Working with the Browser	1-5
Editing Source Code	1-6
Missing Source Code	1-6
Searching	1-6
Drag and Drop	1-7
Controlling Visibility of Methods	1-7

Using Multiple Views	1-7
Source Code Formatting	1-7

Chapter 2 Code Rewrite Editor

Overview	2-1
Transformation Rules	2-2
Using Meta-variables and Modifiers	2-2
Rewriting Methods	2-4
Replacing Whole Methods	2-5

Chapter 3 Override Editor

Reviewing Overrides	3-1
Selecting Overrides	3-2
Restoring an Overridden Definition	3-3
Removing an Overridden Definition	3-3
Publishing Parcels and Packages with Overrides	3-3

Chapter 4 Change Sets

Working with Change Sets	4-1
Selecting a Current Change Set	4-2
Creating a New Change Set	4-2
Exploring Changes	4-3
Browse Methods	4-3
Edit	4-4
Inspect	4-4
Updating the Changes Display	4-5
Saving Changes	4-5
Creating Install and Remove Scripts	4-5
Change Initialization Ordering	4-5
Clearing a Change Set	4-6

Chapter 5 Change List Tool

Using the Change List	5-1
The Change List Tool	5-1
Browsing a Change List	5-3
Reordering Items in the Change List	5-3
Removing Items from the Change List	5-4
Resolving Conflicts with the System	5-4
Using the Conflicts Filter	5-4
Managing Conflicts	5-5

Change/Change Back Changes	5-6
Reverting to a Prior Version	5-7
Recovering from a Crash	5-7
Recovering Changes to a Clean Image	5-8
Condensing the Change List File	5-9
Changing the Change List File Name	5-9
Filing Out a Set of Changes	5-9

Chapter 6 Code Critic

Using the Code Critic	6-2
Filtering Results	6-2
Working with Rule Filters	6-2
Applying Transformation Rules	6-3
Limitations of the Code Critic	6-3
Code Critic Rules	6-4
Bugs	6-4
Possible Bugs	6-5
Unnecessary Code	6-7
Intention Revealing	6-9
Miscellaneous	6-11
Code Transformations	6-14

Chapter 7 Unit Testing

Overview	7-1
Unit Testing in VisualWorks	7-2
Running Test Cases	7-2
Creating a Test Case	7-3
Writing Test Methods	7-3
Using should: and shouldnt:	7-4
Defining Test Resources	7-4
Guidelines for Working with Test Cases	7-5
Criteria for Testing	7-6

Chapter 8 Trippy: Enhanced Inspector

Overview	8-1
Basic Inspecting	8-1
Inspection Views	8-2
Evaluating and Editing	8-3
Editing Variable Values	8-4
Copy and Paste	8-5
Add and Remove	8-5

Undoing an Edit	8-5
Editing with Drag-Drop	8-6
Protected Variables	8-6
Exploring Objects	8-7
Diving into Object References	8-7
Exploring	8-8
Siblings	8-8
Parts	8-9
History Views	8-9
Exploring a Window	8-10
Previewing a Visual Part	8-10
Exploring an Object Hierarchy	8-10
Viewing Related Objects	8-11
Customizing the Inspector	8-12
Define the Object printOn: Representation	8-12
Add Displayed Attributes	8-12
Add Menu Actions	8-12
Identify Hierarchies	8-13
Add an Inspector Page	8-13
Provide Custom Object Views	8-13
Prototype-based Programming in Trippy	8-13

Chapter 9 System Profilers

Loading the Profilers	9-1
Opening a Profiler Window	9-2
Profiling a Block of Code	9-2
Adjusting the Sample Size	9-3
Analyzing the Profiler Report	9-5
Tree Report View	9-6
Totals Report View	9-6
Adjusting the Cutoff Percentage	9-7
Contracting and Expanding the List	9-7
Spawning a Method Browser	9-9
Multi-process Profiling	9-9
Profiler Programmatic Interface	9-10

Chapter 10 Benchmarks

Using the Benchmark Interface	10-2
Assembling the Test Suite	10-2
Selection Techniques	10-3
Setting the Report's Granularity	10-3
Raw Measurements	10-3

Benchmark Results	10-4
Overall Suite Statistics	10-5
Choosing Types of Statistics	10-6
Setting the Report Destination	10-6
Setting the Number of Iterations	10-6
Creating a Benchmark Subclass	10-7
Benchmark Superclass	10-7
SystemBenchmark Subclass	10-7
BenchmarkTable Class	10-8
BenchDecompilerTestClass Class	10-8

Chapter 11 Class Reports

Overview	11-1
Creating Class Reports	11-2
Selecting the Target Classes	11-2
Locating Coding Errors	11-3
Class Report Options	11-4
Messages Sent but Not Implemented	11-4
Messages Implemented but Not Sent	11-4
Method Consistency	11-4
Subclass Responsibilities Not Implemented	11-5
Undeclared References	11-5
Instance Variables Not Referenced	11-5
Check Comment	11-5
Backward Compatibility Message Sends	11-6
Indefinite Backward Compatibility Message Sends	11-7
Backward Compatibility Class References	11-7
Estimating Memory Requirements	11-7
Documenting Your Code	11-8

Index

Index-1

About This Book

Overview

VisualWorks documentation is designed to help both new and experienced application developers create application programs effectively using the VisualWorks® application frameworks, tools, and libraries.

This document, the *VisualWorks Tool Guide*, provides detailed information about the development tools and how to get the most functionality out of them.

Audience

The *Tool Guide* makes very few assumptions about your level of knowledge about object-oriented programming, but does assume you have a basic knowledge of computer programming in some environment.

For additional help, a large number of books and tutorials are available from commercial book sellers and on the world-wide web. In addition, Cincom and some of its partners provide VisualWorks training classes. See [“Additional Sources of Information”](#) below for a listing of some of these resources.

Conventions

We have followed a variety of conventions, which are standard in the VisualWorks documentation.

Typographic Conventions

The following fonts are used to indicate special terms:

Example	Description
<i>template</i>	Indicates new terms where they are defined, emphasized words, book titles, and words as words.
cover.doc	Indicates filenames, pathnames, commands, and other constructs to be entered outside VisualWorks (for example, at a command line).
<i>filename.xwd</i>	Indicates a variable element for which you must substitute a value.
windowSpec	Indicates Smalltalk constructs; it also indicates any other information that you enter through the VisualWorks graphical user interface.
Edit menu	Indicates VisualWorks user-interface labels for menu names, dialog-box fields, and buttons; it also indicates emphasis in Smalltalk code samples.

Special Symbols

This book uses the following symbols to designate certain items or relationships:

Examples	Description
File → New	Indicates the name of an item (New) on a menu (File).
<Return> key <Select> button <Operate> menu	Indicates the name of a keyboard key or mouse button; it also indicates the pop-up menu that is displayed by pressing the mouse button of the same name.
<Control>-<g>	Indicates two keys that must be pressed simultaneously.
<Escape> <c>	Indicates two keys that must be pressed sequentially.
Integer>>asCharacter	Indicates an instance method defined in a class.
Float class>>pi	Indicates a class method defined in a class.

Mouse Buttons and Menus

VisualWorks supports a one-, two-, or three-button mouse common on various platforms. Smalltalk traditionally expects a three-button mouse, where the buttons are denoted by the logical names <Select>, <Operate>, and <Window>:

<Select> button	<i>Select</i> (or choose) a window location or a menu item, position the text cursor, or highlight text.
<Operate> button	Bring up a menu of <i>operations</i> that are appropriate for the current view or selection. The menu that is displayed is referred to as the <i><Operate> menu</i> .
<Window> button	Bring up the menu of actions that can be performed on any VisualWorks <i>window</i> (except dialogs), such as move and close . The menu that is displayed is referred to as the <i><Window> menu</i> .

These buttons correspond to the following mouse buttons or combinations:

	3-Button	2-Button	1-Button
<Select>	Left button	Left button	Button
<Operate>	Right button	Right button	<Option>+<Select>
<Window>	Middle button	<Ctrl> + <Select>	<Command>+<Select>

Getting Help

There are many sources of technical help available to users of VisualWorks. Cincom technical support options are available to users who have purchased a commercial license. Public support options are available to both commercial and non-commercial license holders.

Commercial Licensees

If, after reading the documentation, you find that you need additional help, you can contact Cincom Technical Support. Cincom provides all customers with help on product installation. For other problems there are several service plans available. For more information, send email to supportweb@cincom.com.

Before Contacting Technical Support

When you need to contact a technical support representative, please be prepared to provide the following information:

- The *version id*, which indicates the version of the product you are using. Choose **Help → About VisualWorks** in the VisualWorks main window. The version number can be found in the resulting dialog under **Version Id**.
- Any modifications (*patch files*) distributed by Cincom that you have imported into the standard image. Choose **Help → About VisualWorks** in the VisualWorks main window. All installed patches can be found in the resulting dialog under **Patches**.
- The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, select **copy stack** in the error notifier window (or in the stack view of the spawned Debugger). Then paste the text into a file that you can send to technical support.

Contacting Technical Support

Cincom Technical Support provides assistance by:

Electronic Mail

To get technical assistance on VisualWorks products, send email to:

supportweb@cincom.com.

Web

In addition to product and company information, technical support information is available on the Cincom website:

<http://supportweb.cincom.com>

Telephone

Within North America, you can call Cincom Technical Support at (800) 727-3525. Operating hours are Monday through Friday from 8:30 a.m. to 5:00 p.m., Eastern time.

Outside North America, you must contact the local authorized reseller of Cincom products to find out the telephone numbers and hours for technical support.

Non-Commercial Licensees

VisualWorks Non-Commercial is provided “as is,” without any technical support from Cincom. There are, however, on-line sources of help available on VisualWorks and its add-on components. Be assured, you are *not* alone. Many of these resources are valuable to commercial licensees as well.

The University of Illinois at Urbana-Champaign very kindly provides several resources on VisualWorks and Smalltalk:

- A mailing list for users of VisualWorks Non-Commercial, which serves a growing community of VisualWorks Non-Commercial users. To subscribe or unsubscribe, send a message to:

vwnc-request@cs.uiuc.edu

with the SUBJECT of "subscribe" or "unsubscribe".

- An excellent Smalltalk archive is maintained by faculty and students at UIUC, who are long-time Smalltalk users and leading lights in the Smalltalk community, at:

<http://st-www.cs.uiuc.edu/>

- A Wiki (a user-editable web site) for discussing any and all things VisualWorks related at:

<http://wiki.cs.uiuc.edu/VisualWorks>

- A variety of tutorials and other materials specifically on VisualWorks at:

<http://wiki.cs.uiuc.edu/VisualWorks/Tutorials+and+courses>

The Usenet Smalltalk news group, comp.lang.smalltalk, carries on active discussions about Smalltalk and VisualWorks, and is a good source for advice.

Additional Sources of Information

This is but one manual in the VisualWorks library. The Cincom Smalltalk publications website:

<http://www.cincom.com/smalltalk/documentation>

is a resource for the most up to date versions of VisualWorks manuals and additional information pertaining to Cincom Smalltalk.

Smalltalk Tutorial

A new VisualWorks Smalltalk tutorial is available online at:

<http://smalltalk.cincom.com/tutorial/index.ssp?content=tutorials>

The tutorial information is growing, so revisit this site.

Online Help

VisualWorks includes an online help system. To display the online documentation browser, open the **Help** pull-down menu from the VisualWorks main menu bar and select one of the help options.

VisualWorks FAQ

An accumulating set of answers to frequently asked questions about VisualWorks is being compiled in the VisualWorks FAQ, which accompanies this release and is available from the Cincom Smalltalk documentation site.

News Groups

The Smalltalk community is actively present on the internet, and willing to offer helpful advice. A common meeting place is the comp.lang.smalltalk news group. Discussion of VisualWorks and solutions to programming issues are common.

VisualWorks Wiki

A wiki server for VisualWorks is running and can be accessed at:

<http://wiki.cs.uiuc.edu:8080/VisualWorks>

This is becoming an active place for exchanges of information about VisualWorks. You can ask questions and, in most cases, get a reply in a couple of days.

Commercial Publications

Smalltalk in general, and VisualWorks in particular, is supported by a large library of documents published by major publishing houses. Check your favorite technical bookstore or online book seller.

1

System Browser

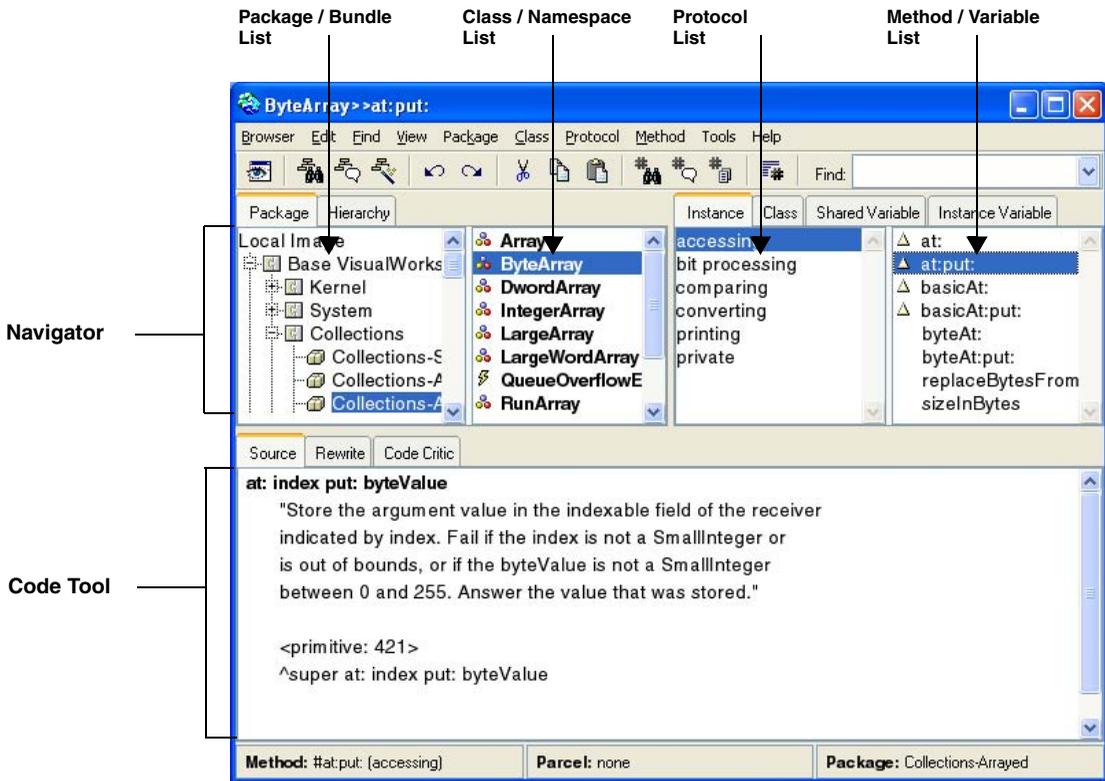
The principal programming tool in VisualWorks is the System Browser. You use it for “browsing” the code library, writing, editing, organizing, and other source code related operations. The browser also provides special-purpose tools for refactoring, rewriting, checking, and testing code, some of which are described in other chapters.



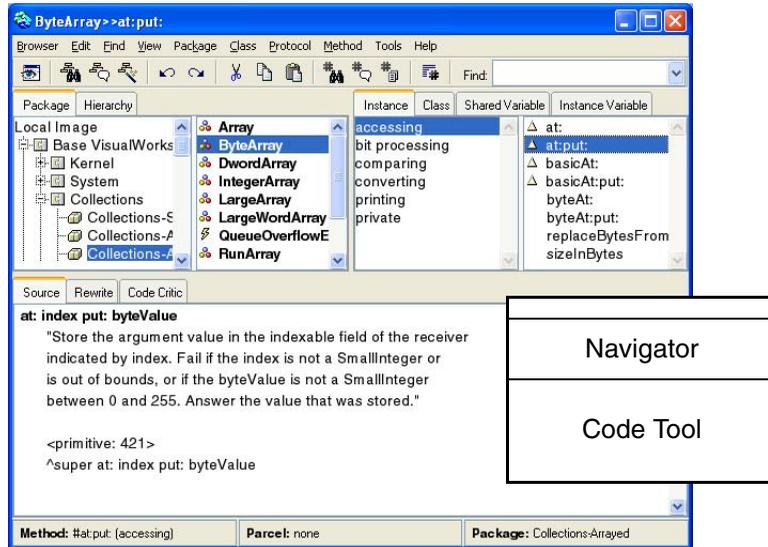
To open a browser, choose **Browse → System** or click on the Browser icon in the VisualWorks Launcher.

The System Browser allows you to select either a Package (the default) or Parcel view of the system components. From either of these views, you can also select a Hierarchy view, either of the selected component or of the entire system. To change the primary view for a browser, select the view from the **Browser** menu. To change default view for a new browser, use the **Browser** page of the Settings Tool (**System → Settings** in the Launcher window, and set the default to either **Package** or **Parcel**.

Depending on the current view, the lists show different items. It will take some experimentation and experience to get comfortable with the browser, but the following comments will guide your learning.



The browser window is composed of a *navigator* and a set of *code tools*. You select a view in the navigator by clicking on upper row of tab control buttons. Code tools are selected using the lower row of tab controls. The tab label indicates which view or tool it selects, and its current focus.



The VisualWorks system is organized as a class library. Classes are defined in an inheritance hierarchy, which you can browse by selecting the navigator's **Hierarchy** tab.

For organizational purposes, classes are grouped into packages, and packages can be grouped into bundles. Packages and bundles can be saved, or “published,” as parcels, which are essentially an external file based representation of a package or bundle. This organization is described more fully in “Managing Smalltalk Source Code” in the [Application Developer's Guide](#).

You use the navigator to traverse the VisualWorks class library, viewing definitions for classes, namespaces, methods, and variables.

The Package, Parcel, and Hierarchy views each has its own <Operate> menu, offering commands that are appropriate to its contents. Many of the commands are obvious. Specific commands are explained throughout this document as the operation is discussed. For details on individual menu functions, view the online help available from the browser's **Help** menu.

Browser Navigator

The different parts of the browser's navigator provide different views of the system. Here is a brief summary of their function and use:

Package View

The VisualWorks library is organized into packages and bundles. Each code definition is contained in a package, and can be viewed by selecting the package. Packages can also be grouped into bundles and the contained definitions browsed. The browser displays packages when **Package** tab is selected in the **Browser**.

When Store is loaded, packages and bundles support code revisioning and related mechanisms to assist in source code management. For information about working with packages, refer to the [VisualWorks Source Code Management Guide](#).

Note that the use of packages has now replaced the use of class categories and parcels as units of organization, as they were used in previous versions.

Parcel View

Parcels are a deployment artifact, providing an external file container for the code defined in packages and bundles. At this time, it is still useful to be able to browse code in terms of the parcel that contains it, which you do using the Parcel view (select **Browser** → **Parcel** in the system browser).

Hierarchy View

Occasionally it is useful to explore a class in terms of the other classes from which it inherits behavior, or that inherit behavior from it. The navigator allows you to do this by displaying the hierarchy of the selected class.

To view the entire class hierarchy, start by selecting class Object. You can then find and browse a class by navigating through the hierarchy to it. Although this is seldom very useful, it can be instructive.

Class / Name Space View

Classes and name spaces are defined in packages, so the contents of the Class / Name space view depend upon the selected Package.

In addition to having a superclass, each class is defined in a name space. A name space is a name resolution scope for name space, class, and shared variable names. Typically, you create your own name space and then create your applications within that name space.

When the class hierarchy view is selected, this view shows the containing package for the selected item.

Instance, Class, and Variable Views

The **Instance**, **Class**, **Shared Variable** and **Instance Variable** tabs toggle the contents of the method category and method/variable views, selecting whether the categories and definitions of instance methods, class methods, shared or instance variables are shown. In some situations, such as when a namespace is selected that has only shared variables defined in it, only one of the buttons, in this case **Shared Variables**, is shown. Usually, any of the buttons can be selected, even though there may be no entries for that view.

Icons in the Navigator

The browser's navigator uses a number of special icons to distinguish code components, special system classes, as well as the condition of individual methods. The following table offers a brief summary:

Icon	Description
	Package
	Bundle
	Name space
	Subclass of Model
	Subclass of ApplicationModel
	Subclass of Collection
	Subclass of Exception
	Method redefined by at least one superclass.
	Method redefined by at least one subclass.

Working with the Browser

The System Browser separates code tools from the navigator so that a variety of code tools may be used with each navigator. Generally, you use the **Source** tool to examine class, namespace and variable definitions, and to browse and edit source code.

The browser includes features for automated code refactoring (refer to “Refactoring,” Chapter 15 in the *Application Developer’s Guide*, for details). For advanced development, the browser also provides special tools for code checking, rewriting, and unit testing, which are described in other chapters.

To encourage learning and experimentation, each operation in the browser can be reversed with the **Undo** function (on the **Browser** menu).

Editing Source Code

The **Source** code tool in a System Browser is where you do most writing and editing of your application’s class and method definitions. Common editing operations, such as cut, paste, find and replace, are available on the <Operate> menu for this pane.

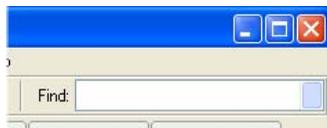
When you select a package but no class, a class definition template is displayed. Similarly, when you select a protocol but no method, a method definition template is displayed. To create a new class or method, edit the template with the appropriate definition. When you have edited a definition, you need to save, or *accept*, your changes. Select **Accept** from the code pane <Operate> menu.

Missing Source Code

Your Smalltalk image is associated with a sources file, as described in the *Application Developer’s Guide*. If the sources file is not correctly identified in the Settings Tool, or your VisualWorks home directory is not correctly set, or if the sources simply are not available, you may see code in the browser with a comment explaining that it is decompiled code. If you see this comment, set the home directory and/or edit the **Source Files** page of the Settings Tool, making sure the **.sou** file name agrees with the image name. (To open the Settings Tool, choose **System → Settings** in the Launcher window.)

Searching

The navigator tool bar includes an entry field to do a quick search by name for classes, variables, or methods:



To find a class, simply enter its name and select **Accept** from the <Operate> menu, or press the <Return> key. To find a method, enter its name, preceded by the # (pound) character. Wildcard searches are possible using the * (asterisk) character.

Drag and Drop

To reorganize code, you can drag and drop methods on classes or protocols; protocols on other classes or on protocols; classes on other categories; and categories on other categories.

Controlling Visibility of Methods

By default, the browser's method list only displays those methods belonging to the currently selected class and protocol. Several options are provided for controlling and expanding the visibility of methods.

When a class is selected, the browser may optionally be set to show all methods in the class when no protocol is selected. To enable this option, select **Show all Methods when No Protocols Selected** on the **Browser** page of the Settings Tool.

Just as it is often useful to see class inheritance using the **Hierarchy** view, so too it is often useful to see inherited methods. To expand the visibility of the Method List to include inherited methods located in a superclass, select the name of the superclass from the **Method → Visibility** menu. This setting remains active until you navigate to another class.

To fix the initial visibility setting so that it remains active while viewing different classes, select **Show All Inherited** or **Show All Inherited Except for Object**. To disable the expanded visibility, choose **Show No Inherited**.

Using Multiple Views

The System Browser can have with multiple active “views” on a method. For example, while editing one method, you can switch to a new view to look up some value in another method, and then return back to your edited method without opening a new browser.



To create a new view, use **View → New** or corresponding icon in the browser's tool bar. Select the entries on the **View** menu to toggle rapidly between the different views you've created. Use **View → Remove** to delete the current view.

Source Code Formatting

To format a method using the browser's integrated code formatter, select **Format** from the source code tool's <Operate> menu.

Many of the browser's refactoring commands also invoke the code formatter, so you should expect a formatting change any time you refactor a method.

The formatting rules are user-accessible and may be changed. The rules are located in class `RBConfigurableFormatter`, and they may be changed using a special tool. To set the browser to use the configurable formatter by default, evaluate:

```
RBProgramNode formatterClass: RBConfigurableFormatter
```

To open configuration the tool, evaluate:

```
FormatterConfigurationTool open
```

The Configuration Tool presents about 20 separate rules. When changing a rule, you must **Accept** the changed value using the <Operate> menu in the value's input field. To examine the effects of the rules on a test method, click on the tool's embedded **Format** button. To save any changes you make to the rules, click on the **OK** button.

Method source in the browsers may also be color coded. To enable color coding, load the **RBCodeHighlighting** parcel (it can be found in the Parcel Manager's **Environment Enhancements** category).

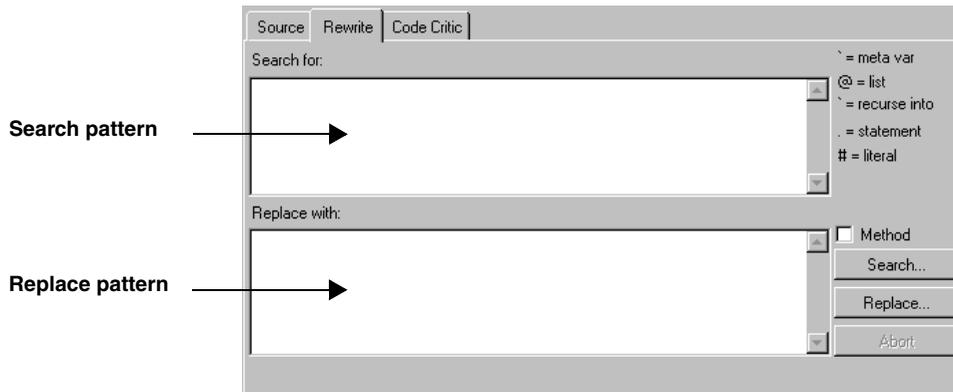
2

Code Rewrite Editor

Overview

The rewrite rule editor, which is integrated into the System Browser, enables you to create search and replace patterns that work at the method's structural level. Unlike simple string matching, these patterns are applied to the method's parse tree.

The rewrite tool uses a special syntax to specify a *transformation rule*. When a transformation rule is applied, it affects the method(s) selected in the browser's navigator. You may specify a single method, or any number of methods, protocols, or classes as the target of a single transformation.



The rewrite editor is available whenever you select the **Rewrite** tab of the browser's code tool. Specify a search pattern in the upper input area of the tool, and a replacement pattern in the lower area.

Use the **Search...** button to locate all occurrences of the search pattern in the method, protocol, or classes selected in the browser navigator. Results are displayed in a new browser. No code is changed.

Use the **Replace...** button to locate all occurrences of the search pattern in the specified code, and then open a transformation editor on all matching methods. The transformation editor allows you to apply the rewrite rule.

The rewrite editor enables you to write your own transformation rules. A set of pre-defined transformation rules are also available, but as part of the Code Critic; for details, see [“Applying Transformation Rules” on page 6-3](#).

Transformation Rules

A transformation rule is specified using a pattern that is iteratively applied to each expression in each method this is selected for rewriting.

Pattern-matching is performed against each node in the method's parse tree. When the rewrite editor finds a node in the method's parse tree that matches the node specified in the parse tree generated from the search pattern, it applies a transformation.

You may also specify a pattern for a whole method, rather than just a single node (for details, see [“Replacing Whole Methods” on page 2-5](#)).

When the rewrite tool scans for parse nodes that match the search pattern, it first converts the pattern into a collection of *meta-variables*. Each meta-variable is identified in the search pattern using a ``` character.

Meta-variables allow pattern matching without having to specify specific variable names. In general, a meta-variable is specified by one or more special characters followed by a valid variable name. For example:

```
`receiver printOn: `variable
```

specifies a pattern with two meta-variables named receiver and variable. This pattern would match the expression:

```
super printOn: aStream
```

Using Meta-variables and Modifiers

The ``` character for specifying a meta-variable may be accompanied by other special characters called “modifiers” that are used to specify the type of node that the meta-variable can match. Modifiers are entered immediately after the ``` character.

For example, since it is often impractical to specify every possible match exactly, the modifier `@` may be added after the ``` character to specify a match for any type of node in the method's parse-tree.

Suppose, to sustain the example, we want to replace every occurrence of the message `printOn:` with the message `print:.` Using `@`, we can specify a general pattern:

```
`@receiver printOn: `@variable
```

would now match the expression:

```
self name printOn: aStream
```

Slight variations of this pattern can be used to replace keyword messages with several arguments. Messages taking block arguments can be easily manipulated in this manner.

For example, to change:

```
maybeNil isNil ifTrue: [trueBlock] ifFalse: [falseBlock]
```

to:

```
maybeNil ifNil: [trueBlock] ifNotNil: [falseBlock]
```

We would use the following rules for search and replace:

```
`@maybeNil isNil ifTrue: `@trueBlock ifFalse: `@falseBlock
`@maybeNil ifNil: `@trueBlock ifNotNil: `@falseBlock
```

Note that the `@` modifier is polymorphic, i.e., it matches any subtree in the node. Depending upon where it appears in the pattern, it can specify anything from an individual node to a collection of statements.

Similarly, a list of temporary variables can be matched with ``@Temps`, e.g.:

```
| `@Temps |
```

When a match is found, it is often necessary to search inside the node for more matches. The ``` character may be used twice (e.g., ```@variable`) to specify this pattern-matching behavior.

The `.` (period) character may be used to match a statement node, thus a list of statements may be matched by using ```@.Statements`.

Four different modifiers are currently supported by the rewrite editor:

Character	Type	Comment	Examples
'	recurse into	Whenever a match is found, look inside this matched node for more matches.	"`@object foo" — matches foo sent to any object, plus for each match found look for more matches in the ``@object part
.	statement	Matches a statement in a sequence node.	".Statement" — matches a single statement
#	literal	Matches only literal objects.	"#literal" — matches any literal (#(), #foo, 1, etc.)
@	list	When applied to a variable node, this will match a literal, variable, or a sequence of messages sent to a literal or variable. When applied to a keyword in a message, it will match a list of keyword messages (i.e., any message send). When applied with a statement character, it will match a list of statements.	" `@Temps ..." — matches list of temporary variables. "@.Statements" — matches list of statements "@@object" — matches any message node, literal node or block node "foo `@message: `@args" — matches any message sent to foo

Rewriting Methods

To rewrite methods using a transformation rule:

- 1 Select the method or methods you wish to rewrite in the browser's navigator and then open the rewrite editor by clicking on the code tool's **Rewrite** tab.
- 2 Enter a search pattern in the upper input field of the rewrite editor.
- 3 Enter a replacement pattern in the lower input field.
- 4 To browse a list of methods that match the search pattern, click on the **Search...** button.
- 5 Open a list of methods that are ready to be transformed, by clicking on the **Replace..** button. A transformation editor appears, showing a list of methods and highlighting the code that will be transformed.
- 6 To actually transform the method(s) that match the search pattern, select **Execute** or **Execute All** from the <Operate> menu in the transformation editor.

Replacing Whole Methods

The rewrite tool also allows you to match and replace an entire method, not just single expressions. To enable entire-method transformations, select the **Method** check-box.

For example, you can search for methods that just return a super message, use the following pattern:

```
`@msg: `@args
  ^super `@msg: `@args
```

To eliminate ifTrue: guard clauses, you might use this search pattern:

```
`@methodName: `@methodArgs
  | `@Temps |
  `@Condition ifTrue:
  [ ` .Stmt1.
    ` .Stmt2.
    `@.Statements]
```

with this replacement pattern:

```
`@methodName: `@methodArgs
  | `@Temps |
  `@Condition ifFalse: [^self].
  ` .Stmt1.
  ` .Stmt2.
  `@.Statements
```


3

Override Editor

The Override List tool provides a view on overrides in the system. It is very much like the Change List tool, and most of the operations the same, so will not be repeated here (refer to “[Change List Tool](#)” in [Chapter 5](#)). There are differences, however, in command behavior that we will cover.

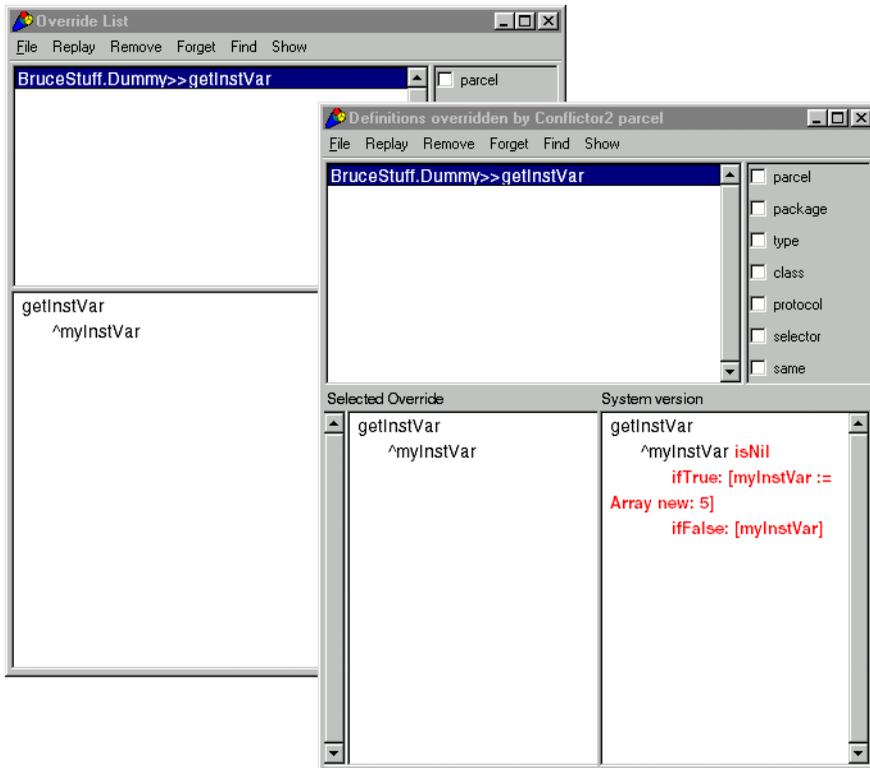
Reviewing Overrides

To open the an Override List showing only the overridden definitions, select **Changes → Open Override List** or **Changes → Browse System Overrides** from the **System** menu in the Launcher. **Browse System Overrides** opens a list of all overrides currently in the system. **Open Override List** opens an empty list to which you can selectively add parcels and/or packages containing overridden definitions.

To compare the overridden and overriding definitions, select the parcel or package to check in a browser and select:

- **Package (or Parcel) → Browse → Overrides of others**, to browse method definitions that have been overridden, or
- **Package (or Parcel) → Browse → Overridden by others**, to browse any methods defined in the parcel/package that have been overridden by another parcel or package.

These options open comparison browser versions of the Override List, putting the overridden and overriding definitions side-by-side for easy comparison.



Selecting Overrides

You can select which overrides are displayed, by selecting the relevant parcels and packages. To list overrides related to a specific parcel or package, select **File → Display Parcel...** or **File → Display Package...** in the Override List. Then select the parcel or package to display in the displayed list and click **OK**.

To list all overrides in the system, select **File → Display System Overrides**. All definitions overridden by a parcel or package are then included in the list.

The check boxes at the right provide filters on the list, to help focus on specific sets of conflicts. With all boxes unchecked, all conflicts are shown. When any boxes are checked, only the items checked are shown.

To show conflicts, select **Show → Show Conflicts**. The different versions of the selected item are then shown in separate panes, with conflicting code shown in red.

Restoring an Overridden Definition

If a definition has been overridden, and you want to restore it as the current definition in the system, use the **Replay** menu options. You have the option to replay a single definition, all displayed definitions, or all from the selected definition to the end of the list.

Once restored, the overridden package/parcel now “owns” the current definition, and competing definitions are removed from all the overbidding components. The parcels can now be saved, without the conflicts blocking the operation.

Removing an Overridden Definition

Alternatively, the overridden definition may be the one that should be removed.

To remove a single overridden definition from a parcel or package, and so to remove the conflict between defining parcels or packages, select the definition in the list and choose **Forget → Purge selection**. The overrider now owns the definition, and the components can be saved. Note that if the overriding parcel/package is unloaded, the overridden definition will not be restored.

Other options are available for purging blocks of definitions. For example, marking definitions using the **Remove** menu items, and then selecting **Forget → Purge these** removes all of the selected definitions from their components.

Publishing Parcels and Packages with Overrides

Parcels and packages behave differently when publishing with overrides. The issue is how to publish code that has been overridden. What happens is:

- If a parcel contains an overridden definition, an attempt to publish will fail, and a notifier is displayed.
- If a package contains an overridden definition, an attempt to publish will succeed, although publishing binary is not allowed, and the package will include its overridden code.

In a parcel, the result would be to publish the overriding code, and the overridden code would be lost. Rather than publish under these conditions, the operation is cancelled. To republish the parcel, you must remove the override condition, either by removing the overridden definition from the parcel, or by copying or moving the overriding definition into the parcel.

In a package, the mechanism allows keeping the overridden and the overriding code separate, and so the package can be published while retaining its original (overridden) code. To keep the original code, simply publish the package. To update the package with the overriding code, you must copy or move the code into the package.

Since publishing a package in binary creates a parcel format file, which cannot contain overridden definitions, the binary option is disabled if the code contains an overridden definition.

Due to differences in how parcels are constructed, this difference is unlikely to be removed in the future.

4

Change Sets

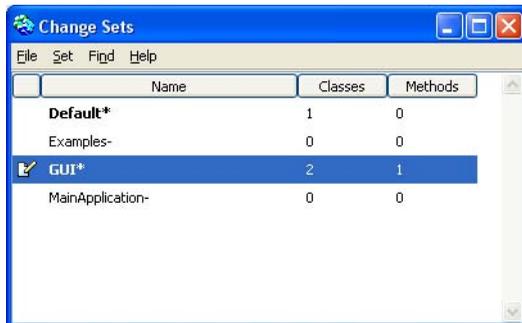
Named change sets (or simply *change sets*) provide a “project-based” view of changes you have made to the system. By using multiple change sets, you can keep the changes made for different applications or subsystems separate, while maintaining a single development environment. This is particularly useful if you work on multiple small projects at the same time, but do not want to maintain separate images for each.

Working with Change Sets

Change set entries represent either new or changed class definitions and their methods, or individual methods that you create or change without modifying the class itself. These define a set of definitions that you can then file out as a group.

Unlike the Change List (see “[Change List Tool](#)” in [Chapter 5](#)), change sets do not record the evolution of those changes. Instead, a change set contains the current definitions of classes and methods in the system.

You manage change sets by using the Change Sets Manager. In this tool, you set the *current* change set and access operations on change sets, using the menu options. To open the Manager, select **System → Changes → Change Sets** in the VisualWorks Launcher.



In addition to the list of change set names:

- The **Classes** column lists the number of classes in each change set that have changes to the class definition itself; filing out will include all methods.
- The **Methods** column indicates the number of loose methods that will be included when filing out (methods changed without changes to their classes).

Selecting a Current Change Set

The Change Set Manager always has the **Default** change set, plus any change sets that are defined in the image. If no change set is selected, or if **Default** is selected, all changes go to the default change set. Otherwise, they go to the selected change set.

To make a change set active or current, double-click on the name in the change set list, or select it and pick **Set → Make Current**. All changes you make to the system will then be saved in that change set.

You can also change the current change set by clicking on the change set icon on the status bar of the Launcher.

Creating a New Change Set

To add a new change set, select **Set → New**, or select **New** in the change set list <Operate> menu. Enter a name for the change set in the prompter, and click **OK**.

To make this the current change set, double-click on its name.

Alternatively, click on the change set icon on the status bar of the Launcher, and select **New Change Set**.

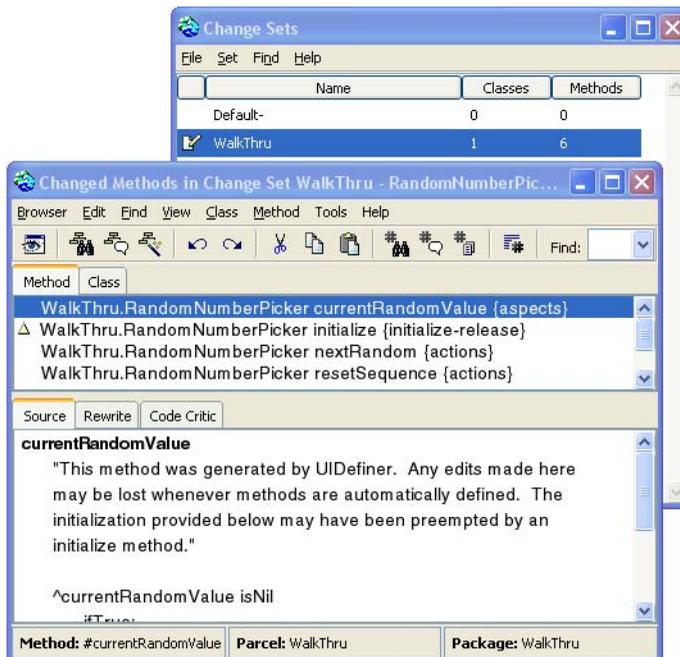
Exploring Changes

Having made changes that are assigned to a change set you can review them. Three menu options in the Change Set Manager **Set** (or <Operate>) menu to allow you to review your change set's contents.

Change sets do not separately report changes to methods when the class that contains them is already in the change set. When you file out the new class, its methods are included. However, if you empty the change set or “forget” the class addition, successive method changes are recorded.

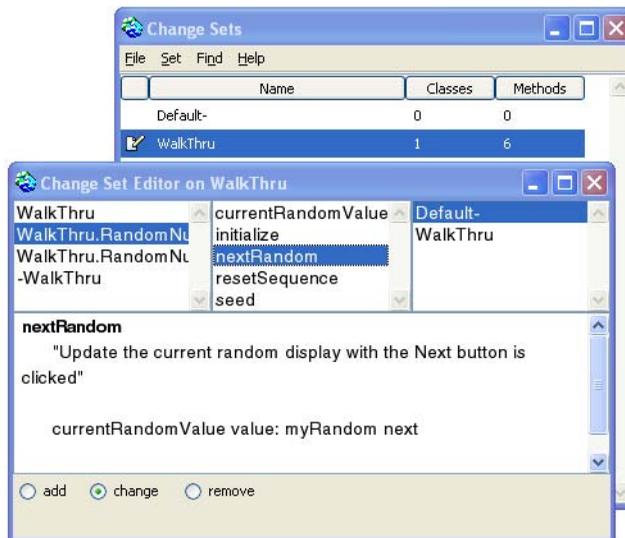
Browse Methods

This menu pick opens a Method Browser on methods changed and recorded in the change set.



Edit

This menu pick opens an editor browser on the current change set. You can change the selected change set in the Change Set List, and the editor will update to show the changes for that change set.



The top-left pane lists classes that either have changed or contain loose methods that have changed, recorded in this change set. The top-right pane lists the named change sets, for information only; it is inactive.

The top center pane lists methods that have changed for the selected class, if any are recorded. Methods for classes whose definitions are in the change set are not listed, since a file-out will include them anyway.

You can edit the definitions in this editor, but the edits do not survive, either in the system or in the change set. To edit a definition, select it and then pick **Browse** or **Spawn** in the <Operate> menus to open a browser on the item. To remove just the one item from the change set, select **Forget** in its <Operate> menu.

The check boxes and radio buttons (depending on what is selected) at the bottom of the window indicate the kind of change recorded. You can change these annotations, and they are saved with the change set, but there is little value in doing so in most cases.

Inspect

This menu option opens an inspector on the change set. Here you can perform the usual inspector options.

Updating the Changes Display

To update an open Change Set browser after making a change to the system, select **update** in the <Operate> menu for a browser pane.

Saving Changes

Change sets are typically used to identify sets of changes that can then be distributed as file-out format files. Change sets are saved in source code format, and so can be browsed in the Changes List.

To write out all the changes in a change set, select the change set and select **File → File Out...** . You will be prompted for a file name.

As a shortcut, to file out all save sets, select **File → File out All...** . You will be prompted for a directory name. The directory will be created, if necessary, and a separate file-out file for each change set is written to it.

You can file out a single method by selecting it in the Change Set Editor (**ChangeSet → Edit**), then selecting **File out as...** in the <Operate> menu.

Note that, when filing out a change set that includes defining a class, all subsequent changes made to methods in that class are also (implicitly) assigned to the change set. This is true even if a different change set is “current” when those method changes are made. A file-out the first change set will include the method definitions.

Creating Install and Remove Scripts

To assist in installing and removing the code filed-out from a change set, you can create import and removal scripts. Simply select the change set in the Change Set List, and select either **ChangeSet → Import Script** or **ChangeSet → Remove Script**.

To be effective, the scripts must be created from the change set while it is exactly the same as when the file-out was created.

Change Initialization Ordering

Change Sets have an `initializationOrder` instance variable which can be used to override the default class initialization ordering derived from the class hierarchy. This is useful in some complex change sets where initialization order is important.

Clearing a Change Set

When a particular change or collection of changes is secure, so that you do not need to continue to hold it in the change set, you can remove it. For example, after filing-out a set of changes, you can purge the whole set, since you can restore them from the file-out file.

To empty all changes from the Change Set for the active project, select **ChangeSet → Empty** in the Change Set List.

To remove a single change from the change set, select the change in the Change Set Editor, and select **forget** in the item's <Operate> menu.

5

Change List Tool

Using the Change List

VisualWorks maintains and records a running list of changes made to the system, in the changes file. By default, the file has the same name as your image but with a `.cha` extension. The changes file is saved in source-code format.

The changes file records anything that changes the state of the system, such as: changes resulting from loading parcels and filing in code; added and modified class and method definitions; special dolts and related operations.

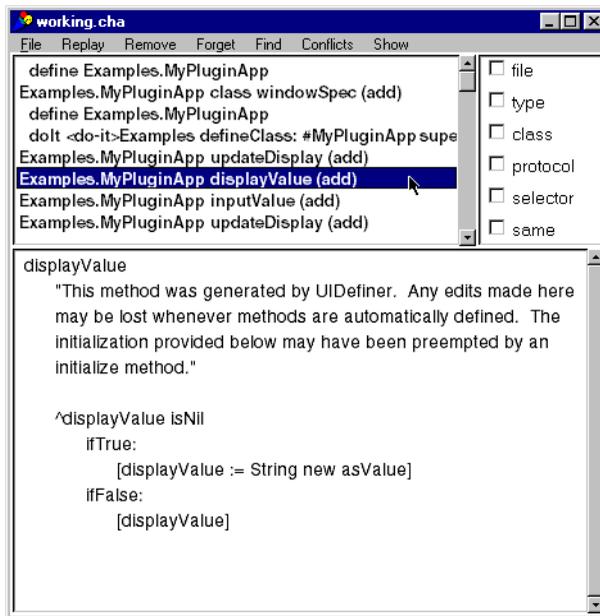
Changes are recorded as they are made, preserving a record of changes even if you exit VisualWorks without saving or if the system crashes. For this reason, the changes file provides a sure way of recovering lost work.

To work with the changes file, VisualWorks has a Change List Tool that allows you to build and manipulate a *change list*, which is based on the contents of the changes file, change sets, and other file-in format files containing descriptions of changes.

The Change List Tool

The Change List tool allows you to work with a change list. It provides a wide variety of operations for reading changes files, comparing the contents of files to the system, filtering the display, and installing changes into the system.

To open the Change List tool, select **Tools → Change List** in the Launcher window.



The Change List window has three views. The view at top left displays a list of the changes. Entries in the Change List generally identify the affected object and the nature of the change, such as **NotifierController menu (add)**. When you select an entry, the affected class or method displays in the text view as it existed after the change.

The top right-hand view provides on/off switches for filtering the contents of the change list. Any combination of filter switches can be selected. The switches filter the list *based on the currently selected list item*. The filters have no effect if no item is selected, and so cannot be selected.

For example, to display only changes that affect the same class as the one affected by the *currently selected* change list entry, click on the **class** switch. To further restrict the listing to identical entries, such as **NotifierController menu**, click on the **same** switch.

Several operations using the Change List are described in this section. For descriptions of menu items not covered here, refer to the online VisualWorks Tools help.

Browsing a Change List

The Change List Browser is initially empty when it opens. This allows you to select what set of changes you want to view, whether in the current changes file or in some other file. To display changes, use one of the following options in the **File** menu:

Read File/Directory

This option reads into the browser the contents of a changes file you specify, or from all changes files in a directory you specify. If you specify a directory, the contents are added to the browser in the order read. To add files in a specific order, read them individually.

Recover Last Changes

This option reads in to the Change List Browser all changes to the system since the last image save. Use this option to recover lost work, such as from a system crash.

Display System Changes

This option appends any changes in the current Change Set (project) to the list of changes in the browser. Unlike the Change Set browser, which displays only a summary, this shows the history of changes.

Display All System Changes

This option adds all changes in all Change Sets to the browser display.

Parcels

This option adds changes from a given parcel that is present in the system. This can be used to examine a parcel's unloaded code and its overridden extension methods as well as normal code.

Reordering Items in the Change List

Some errors may be caused by the order in which changes were made in the system. For example, one operation may require that an object be initialized to a state, but the initialization was neglected or performed too late. Rather than repeat the series of operations manually, the Change List can be used to reorder and then replay the operations.

To change the order of operations, display system changes. Select an operation item to move, click and hold it using the <Select> button, drag the item up or down in the list to an appropriate position, then release (drop) it.

You can now replay the operations to execute them in the new order.

Removing Items from the Change List

The **Remove** and **Forget** menus provide a large number of options for selectively excluding items in the Change List for processing. For brief descriptions of each of these, refer to the VisualWorks Tools help topics.

The **Remove** options mark items for removal from the current list of changes. Marked items are shown in strike-out type style. Options allow you to mark either individual items or large groups of items.

Once a collection of items are marked for removal, you can remove them from the list. In the Forget menu, select either **Forget these** or **Forget all**, to remove the marked items from the change list. The difference is that **Forget all** removes even any marked items that are not showing at the moment due to the filtering selections; **Forget these** only removes those currently showing.

To clear removal markings, use the **Restore...** menu items in the **Forget** menu.

Note that removing items only removes them from the current change list, not from the change list file. You can always get back by re-reading the changes file.

Resolving Conflicts with the System

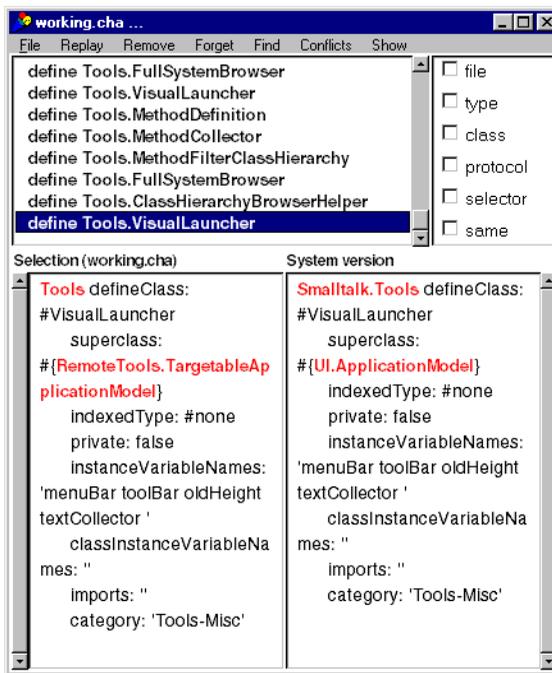
Several options in the Change List tool help you assess the impact of a set of file-ins on the current system. These facilities filter changes based on their similarity or dissimilarity to the current system.

A major use of the conflicts view is to merge changes made by a collection of files, and so construct a single file containing only the desired changes. It can also be an aid in crash recovery, by filtering older changes from a changes file.

Using the Conflicts Filter

Selecting the **Show → Show conflicts** splits the lower text view into two adjacent text views (vertically or horizontally, set by **Show→Conflicts→ Vertical view** or **Horizontal view**). The left-hand or upper view shows the text for the selected change. The right-hand or lower view shows the text of

the corresponding system entity (method or class definition, class comment or organization, etc.) or an explanatory message if this doesn't exist.



The differences between the two texts are high-lighted. This gives you a quick, graphic indication of what would be changed by filing-in a specific change.

Turning off the show conflicts filter hides the conflict text view and returns the change list view to its usual appearance.

Managing Conflicts

Several items on the **Conflicts** menu allow adding or otherwise processing conflicts between the changes list and the system. The full set of menu items are briefly described in the VisualWorks Tools Help. Here we comment on a few of the more interesting options.

Add system conflicts

For each displayed change that has a version in the system with which it conflicts, this option adds the corresponding system version of the change to the change list.

Add original versions

This option scans the system's source files (excluding the current changes file) and, for each displayed change for which a corresponding change exists in the sources file, add the sources file version to the change list. This is useful comparing your changes against the original sources.

Add to change set/Remove from change set

These options update the current change set to include or exclude the changes in the change list, without filing in the changes. This is useful when you have an old file-in representing a component that you wish to extract from the system. This can be used together with the System Browser's **Parcel → Build → Add Changes** and **Parcel → Build → Remove Changes** options.

Change/Change Back Changes

Method changes check to see if they're filing in from the sources files. If so, they set the new compiled method's source pointer to the sources file, and remove the method from the current change set. This enables a technique for working with methods that you frequently change and then change back.

To revert changed-then-changed-back methods do the following:

- 1** Once you have a set of system changes, use **Add originals** to pull-in all corresponding changes from the sources files. These appear after the current changes.
- 2** While holding down the Shift key, choose **Remove → Exact Duplicates** to remove the changes in the sources files that are duplicates of the changes further up the list. Holding the Shift key down causes removal to happen at the end of the list rather than at the beginning.
- 3** Choose **Remove → Exchange removed** to select the set of source file changes that match the current system.
- 4** Select **Forget → Forget these**, and turn on **Show → Show conflicts** and **Show → Show file** to make sure that these changes are indeed on the sources files and identical to the current versions in the system.
- 5** Select **Replay → All from the top**, and watch the transcript to see that each filed-in method change says "in sources file."

Reverting to a Prior Version

During the course of development, a class or method may undergo several changes. The Change List tool makes it easy to see the evolution of, and to examine the details of, the code at any stage in its development. This is particularly useful when you need to see a prior version so you can change the code back.

To display the changes that have occurred since the last snapshot was taken, select **Recover last changes** in the <Operate> menu of the list view at the top. If you want to display changes that are in the Change Set, select **Display system changes** instead.

Once you have displayed the change you want to revert to, select that change and then select **Replay selection** in the <Operate> menu (or **Replay → This Change**).

To revert to a whole collection of changes, read in the necessary changes and set the filters to show exactly the changes you want to load. Then choose **Replay all**.

Recovering from a Crash

If some change you made to the system causes it to crash, the Change List provides a way to recover changes up to, but excluding the change causing the crash. In this way it provides a powerful crash recovery tool.

To recover from a crash:

- 1 Launch the last saved image.
- 2 Open a Change List, and select **File → Recover last changes**.
- 3 Using a combination browsing and editing operations on the displayed list, remove unneeded items that may have contributed to the system crash.

This may involve a good deal of work, browsing the changes first and understanding what ultimately caused the crash, which was probably an interaction between several changes.

Dolts in particular are not usually necessary to recovering changes, and may easily contribute to system instability. To remove all dolts, select on, and then click the **type** checkbox. This filters the list to show only the dolts. You can then select **Remove → All** to clear all dolts from the list.

- 4 Once the list contains just those operations you wish to recover, select **Replay → All** from the top, or another appropriate replay option.

Recovering Changes to a Clean Image

If your image file is damaged in a crash, you may need to recover your changes into a clean image. By “clean image” we mean a copy of the original `visual.im` that shipped with VisualWorks. If you have modified this image file, you will need to start with one from the distribution media.

The technique described here uses the changes file (*myimage.cha*) related to the damaged image. You should back up this file before proceeding. Then:

- 1 Backup your changes file.
- 2 Launch VisualWorks with the clean `visual.im` image file.
- 3 Load any parcels that were loaded in the lost image.

Parcel loading is not included in the `.cha` file, so they must be loaded to ensure that code required by the changes is available.

- 4 Save this image to a *new name*, different from the name of the damaged image.

If you use the same name as the damaged image, you will overwrite the changes file you need for recovering. You will be able to rename the original name later, after you have recovered your work.

- 5 Open a Change List (select **Tools** → **Change List** in the Launcher window), and load the changes file using **File** → **Read file(s)...** .

If your changes file is large, reading the file may be slow. Be patient.

- 6 Remove dolts from the change list by selecting a **dolt** line, clicking the **Type** check box to filter the list, then selecting **Remove** → **All**. Then, select **Forget** → **Forget these** to remove the dolts from the list.

Remove → **All** marks all of the dolts for removal. To unmark one, select it and choose **Forget** → **Restore selection**.

Remove at least dolts that you invoked from a workspace or browser, since this might fail. In general, you may be able to remove all dolts.

- 7 Uncheck **Type**, to show all the remaining changes.
- 8 Examine the list, especially near the end, to see if there is a change that might have caused the damage. If so, remove it from the list using **Remove** → **Selection** and **Forget** → **Forget These**.

There are a variety of changes you might wish to remove. For example, a method may be defined several times, which is okay as long as the last definition is the one you want.

9 Select **Replay** → **All from the top**, to restore all of your changes.

10 Save the resulting image.

At this point you have recovered your changes into the new image. Test it, and if you are satisfied that it is stable, you may save it to the original image name. Note that the old changes file will then be overwritten, so you will not be able to repeat the process using it.

It frequently takes a few tries to get exactly what you want into the restored image, so repeat the procedure until you have just what you want.

Condensing the Change List File

In a large development effort, spanning months or years of programming, the changes file can become very large. To condense it so that it contains only the most recent change for each method, select **System** → **Changes** → **Condense Changes** in the Visual Launcher or evaluate the expression `SourceFileManager default condenseChanges`. Changes involving anything other than a method—such as a class addition or redefinition—will also be purged from the file permanently. VisualWorks will assist you by making a backup copy of the changes file before condensing it.

Changing the Change List File Name

By default, the change list is written to a file with the same file name as the image file, but with a `.cha` extension. It is seldom necessary to use a different file name. If you do need to change the file name, edit the file name in the Settings Tool, **Source Files** page.

Filing Out a Set of Changes

When the code you want to share consists of fragments from many different classes and categories, it may be more convenient to use the Change List to **Write file** with the desired code. Begin by loading all changes into the Change List tool, as described in “[Browsing a Change List](#)” above.

Next, remove the irrelevant changes. For example, dolts are likely candidates for removal because they rarely affect the image in a lasting way. Also, use **Remove** → **Old Versions** to remove duplicate entries, as when a method has undergone several changes, and leave only the last entry in each case. Use **Remove** → **Selection** and **Remove** → **All** to mark one or more changes for deletion, then use **Forget** to erase them from the list. Use the filter switches to control the affected range of entries.

For example, to remove all dolts, begin by selecting any dolt. Then turn on the **type** switch so all of the dolts are listed. Select **Remove all** in the <Operate> menu to mark them for deletion, then **Forget** to erase them. Then turn off the **type** switch to see the remaining entries.

When the displayed list of changes is the desired set, select **Write file** in the <Operate> menu and supply the name of a file in which to store the code. That file can then be loaded into another image via the **File in** command in a File Editor or File List.

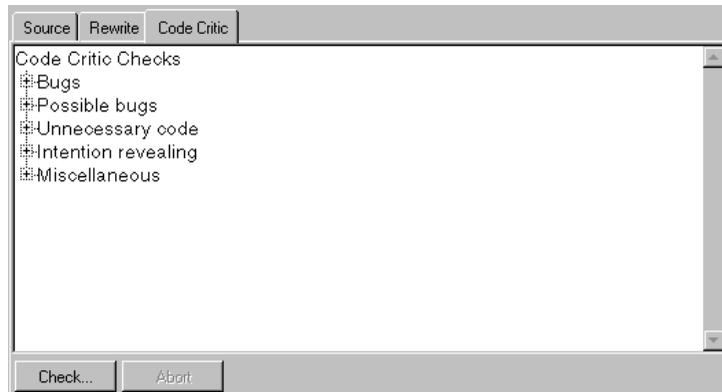
Only the displayed changes are included in a **Write file** operation, so if it is possible to define the minimum set of changes by using the filter switches alone, it is not necessary to **Remove** and **Forget** the nondisplayed entries.

Note: When you write selections to a file, be sure to choose a file name that is different from any file that has been read into the change list. The change list maintains pointers to the code in the files that are read in, and these pointers become invalid when you overwrite a file.

6

Code Critic

The VisualWorks browser includes a Code Critic tool that may be used to screen application code for over 60 common types of bugs.



The Code Critic also provides a mechanism for applying a set of pre-defined transformation rules to your application code. These rules express “best practices” for code development, and are almost always safe to apply.

To write your own transformation rules, refer to the discussion of the [“Code Rewrite Editor” on page 2-1](#).

For a detailed description of all the rules and transformations used by the Code Critic, see [“Code Critic” on page 1](#).

Using the Code Critic

To check a class, a protocol, method or methods:

- 1 Use the browser navigator to set the scope of the test. Select multiple classes, protocols or methods by holding down the **<Shift>** key.
- 2 Select the tool using the **Code Critic** tab control in the lower part of the browser.

The Code Critic tool presents a hierarchical list of rules. You may select the entire list, or individual rules that you wish to check against. Select multiple rules by holding down the **<Shift>** key.

- 3 With both code and the rules chosen, run the critic by clicking on the **Check...** button.

Once the critic finishes checking your code, it presents a list of classes/methods that failed a check. If all checks pass without error, the message **No results found** appears in the lower portion of the browser.

All methods that fail a check are gathered together and displayed in a results window. Results are categorized by type, with a list of rules that failed to pass followed by the number of methods that failed to pass each rule (shown inside square brackets).

To open a Method List browser on all the methods that failed to pass a given rule, select a rule in the results window and click on the **Browse...** button.

Use the **Remove** button to remove items from the results list. This feature may be helpful for keeping track of items that have been fixed.

Filtering Results

The Code Critic includes a mechanism for creating and applying special rule filters. These enable you to ignore a particular rule for a particular class or method.

For example, during a session with the Critic, once problems associated with certain rules have been fixed, we may want to ignore those rules during subsequent checks with the Code Critic.

Working with Rule Filters

To filter out an item visible in the Critic's results list, select it and choose **Add Filter** from the **<Operate>** menu or choose **Class → Add Filter** from the browser menu bar.

To save the current filter set, or load another one, select **Save Filters...** or **Load Filters...** from the browser's **Edit** menu. To reset the filters used by the Critic, select **Clear Filters**.

Applying Transformation Rules

To view the available rules, select **View → Transformation Rules** from the browser's **Edit** menu.

Follow the same general steps for applying the rules to your code:

- 1 Use the browser navigator to set the scope of the transformation. Select multiple classes, protocols or methods by holding down the **<Shift>** key.
- 2 Select the individual transformation rules you wish to apply. You may select multiple rules by holding down the **<Shift>** key.
- 3 With both code and the rules chosen, run the critic by clicking on the **Check...** button.

Once the critic finishes checking your code, it presents a list of classes/methods that may be transformed. If all checks pass without error, the message **No results found** appears in the lower portion of the browser.

To preview the effects of a code transformation, select a method in the upper portion of the results window. The current version of the method and the transformed result are displayed side-by-side in the lower half of the results window.

To apply the code transformation to the highlighted method, select **Execute** from the **<Operate>** menu. To apply all the transformations in the results window, select **Execute All**.

Limitations of the Code Critic

As with all automated tools, not everything flagged by Code Critic is necessary a bug, rather, they are potential problems that might merit further attention. Be aware that for some rules, the Code Critic may return false positives.

Code Critic Rules

This section summarizes the rules used by the browser's Code Critic tool. The rules are organized in five groups:

- Bugs
- Possible Bugs
- Unnecessary Code
- Intention Revealing
- Miscellaneous

Bugs

Rule	Description
Messages sent but not implemented	Checks for messages that are sent by a method, but no class in the system implements such a message. These will certainly cause a <code>doesNotUnderstand:</code> message when they are executed.
Self/Super sends not implemented	Similar to the "Message sent but not implemented" check, but only checks messages sent to self or super since these can be statically typed.
Overrides a "special" message	Checks that a class does not override a message that is essential to the base system (e.g., <code>Object>>class</code>).
References an undeclared variable	Checks for references to a variable in the Undeclared dictionary. If you remove a variable from a class that is accessed by a method, you will create an undeclared variable reference for those methods that accessed the variable.
Subclass responsibility not defined	Checks that all <code>subclassResponsibility</code> methods are defined in all leaf classes.
Uses <code>A B = C</code> instead of <code>A (B = C)</code>	Checks precedence ordering of <code>&</code> and <code> </code> with equality operators. Since <code> </code> and <code>&</code> have the same precedence as <code>=</code> , there are common mistakes where parenthesis are missing around equality operators.

Rule	Description
Uses True/False instead of true/false	Checks for uses of the classes True and False instead of the objects true and false.
Variable used, but not defined anywhere	Similar to the "References an undeclared variable" check, but looks for variables that are not defined in the class or in the Undeclared dictionary.

Possible Bugs

Rule	Description
Assignment inside unwind blocks should be outside	Checks assignment to a variable that is the first statement inside a value block that is also used in an unwind block.
Defines = but not hash	Checks that all classes that define = also define hash. If hash is not defined then the instances of the class might not be able to be used in sets since elements that are equal must have the same hash.
Has class instance variable but no initialize method	Checks that all classes that have class instance variables also have an initialize method. This ensures that all class instance variables are initialized properly when the class is filed-in to a new image.
Instance variable overridden by temporary variable	Checks for methods with block temporary variables that override an instance variable. This causes problems when using the instance variable inside the method.
Missing super sends	Checks that some methods contain a send to super. For example, the postCopy method should always contain super postCopy.
Modifies collection while iterating	Checks for sends to remove: from inside of collection iteration methods such as do:. These can cause the do: method to break since it will iterate beyond the end of the collection. The common fix for this problem is to copy the collection before iterating over it.
More addDependent: messages then removeDependent:	Check that the number of addDependent: message sends in a class is less than or equal to the number of removeDependent: messages. If there are more addDependent: sends, it is possible that some dependents are not being released, which may lead to memory leaks.

Rule	Description
Possible missing "; yourself"	Checks for missing "; yourself" cascaded message send for cascaded messages. This helps locate common coding mistakes such as: anArray := (Array new: 2) at: 1 put: 1; at: 2 put: 2. I.e., anArray would be assigned the value 2 rather than the array object.
Possible three element point; E.g., x @ y + q @ r	Checks arithmetic statements for possible three element points (i.e., a point that has another point in its x or y part).
References an abstract class	Checks for references to classes that have subclassResponsibility methods. Such references might create instances of the abstract class or might be used as the argument to an isKindOfClass: message (the latter is considered bad style).
Returns a boolean and non boolean	Checks for methods that return a boolean value (true or false) and return some other value such as (nil or self). If the method is supposed to return a boolean, this suggests there is a path through the method that might return a non-boolean. If the method doesn't need to return a boolean, you should probably rewrite it to return some non-boolean value since other programmers reading your method might assume that it returns a boolean.
Returns value of ifTrue:/ifFalse: without ifFalse:/ifTrue: block	Check for methods returning the value of an ifTrue: or ifFalse: message. These statements return nil when the block is not executed. For example, the following code will return nil when aBoolean is false: <pre>methodName ^aBoolean ifTrue: [0]</pre> If the code should return nil when aBoolean is false, then it should probably be written as: <pre>methodName ^aBoolean ifTrue: [0] ifFalse: [nil]</pre>

Rule	Description
Sends different super message	<p>Checks for methods whose source sends a different super message. A common example of this is in creation methods. You might define a method such as:</p> <pre>createInstance ^super new initialize</pre> <p>If the new method is not defined in the class, you should probably rewrite this to use <code>self</code> instead. Also, if the new method is defined, you might question why you need to use the superclass' new method instead of new method defined in the class.</p>
Subclass of class that has instance variable but doesn't define <code>copyEmpty</code>	<p>Checks that all subclasses of <code>Collection</code> classes that add an instance variable also redefine the <code>copyEmpty</code> method. This method is used when growing the collection. It copies over the necessary instance variables to the new, larger collection.</p>
Temporaries read before written	<p>Checks that all temporaries are assigned before they are used. This can help find possible paths through the code where a variable might still be unassigned when it is used.</p>
Uses the result of an <code>add:</code> message	<p>Check for possible uses of the result returned by the <code>add:</code> or <code>addAll:</code> messages. These messages return their arguments not the receiver. As a result, many uses of the results are wrong.</p>

Unnecessary Code

Rule	Description
Block immediately evaluated	<p>Check for blocks that are immediately evaluated. Since the block is immediately evaluated, there is no need for the statements to be in a block.</p>
Check for same statements at end of <code>ifTrue:ifFalse:</code> blocks	<p>Checks for <code>ifTrue:ifFalse:</code> blocks that have the same code at the beginning or end. Instead of having the same code in two places, it should be moved outside the blocks.</p>
Class not referenced	<p>Check if a class is referenced either directly or indirectly by a symbol. If a class is not referenced, it can be removed.</p>

Rule	Description
Instance variables not read and written	Checks that all instance variables are both read and written. This check does not work for data model classes since they use the <code>instVarAt:put:</code> messages to set instance variables.
Method just sends super message	Check for methods that forward the message to its superclass. These methods can be removed.
Methods equivalently defined in superclass	Check for methods that are equivalent to their superclass methods. Such methods don't add anything to the computation and can be removed, since the superclass's method will work just fine.
Methods implemented but not sent	Check for methods that are never sent. If a method is not sent, it can be removed.
Unnecessary = true	Check for an <code>=</code> , <code>==</code> , <code>~=</code> , or <code>~~</code> message being sent to <code>true/false</code> or with <code>true/false</code> as the argument. Many times these can be eliminated since their receivers are already booleans. For example, <code>anObject isFoo == false</code> could be replaced with <code>anObject isFoo</code> not if <code>isFoo</code> always returns a boolean. Sometimes variables might refer to <code>true</code> , <code>false</code> , and something else, but this is considered bad style since the variable has multiple types.
Variable referenced in only one method and always assigned first	Checks for instance variables that might better be defined as temporary variables. If an instance variable is only used in one method and it is always assigned before it is used, then that method could define that variable as a temporary variable of the method instead (assuming that the method is not recursive).
Variables not referenced	Check for variables not referenced. If a variable isn't used in a class, it should be deleted.

Intention Revealing

Rule	Description
Assignment to same variable at the end of ifTrue:iffalse: blocks	<p>Checks for ifTrue:iffalse: blocks that assign the same variable at the end of the block. Instead of having the assignment being in both blocks, we can instead assign the variable the result of the ifTrue:iffalse: message. For example, this code:</p> <pre>aBoolean ifTrue: [foo := true] ifFalse: [foo := anotherBoolean]</pre> <p>could be rewritten as:</p> <pre>foo := aBoolean ifTrue: [true] ifFalse: [anotherBoolean]</pre> <p>Once we have simplified the expression by pulling the assignment out of the blocks, then we could see that the code is equivalent to:</p> <pre>foo := aBoolean or: [anotherBoolean]</pre>
Guarding clauses	<p>Checks for ifTrue: or ifFalse: conditions at the end of methods with two or more statements inside their blocks. Such methods might be more comprehensible if they returned self instead. For example, the following code:</p> <pre>someMethod a isNil ifFalse: [self doSomething. self doAnotherThing]</pre> <p>might be better represented as:</p> <pre>someMethod a isNil ifTrue: [^self]. self doSomething. self doAnotherThing</pre> <p>In the first method, a not being nil looks like the exception, but most likely a being nil is the exception which is more obvious in the second method.</p>
ifTrue:iffalse: returns instead of and:/or:'s	<p>Checks for common ifTrue: returns that could be simplified. For example,</p> <pre>foo aCondition ifTrue: [^false]. ^true</pre> <p>can be simplified as:</p> <pre>foo ^aCondition not</pre>

Rule	Description
Method defined in all subclasses, but not in superclass	Checks classes for methods that are defined in all subclasses, but not defined in self. Such methods should most likely be defined as subclassResponsibility methods to help document the class. Furthermore, this check helps to find similar code that might be occurring in all the subclasses that should be pulled up into the superclass.
Sends add:/remove: to external collection	Checks for methods that appear to be modifying a collection that is owned by another object. Such modifications can cause problems especially if other variables are modified when the collection is modified. For example, CompositePart must set the containers of all its parts when adding a new component.
Unnecessary size check	Check for code that checks that a collection is non-empty before sending it an iteration message (e.g., do:, collect:, etc.). Since the collection iteration messages work for empty collections, the method does not need to be cluttered with the extra size check.
Uses "size = 0" or "= nil" instead of "isEmpty" or "isNil"	Checks for methods using equality tests instead of the message sends. Since the code aCollection size = 0 works for all objects, it is more difficult for someone reading such code to determine that aCollection is a collection. Whereas, in the expression aCollection isEmpty, it is clear that aCollection must be a collection since isEmpty is only defined for collections.
Uses at:ifAbsent: instead of at:ifAbsentPut:	<p>Checks for uses of at:ifAbsent: in place of the shorter at:ifAbsentPut: message. For example:</p> <pre>aDictionary at: aKey ifAbsent: [aDictionary at: aKey put: anObject]</pre> <p>should be rewritten as:</p> <pre>aDictionary at: aKey ifAbsentPut: [anObject].</pre> <p>You may also use one of the Code Critic's transformation rules to convert these methods.</p>
Uses detect:ifNone: instead of contains:	<p>Checks for the common code fragment:</p> <pre>(aCollection detect: [:each 'some condition'] ifNone: [nil]) ~= nil</pre> <p>which can be simplified and clarified as:</p> <pre>aCollection contains: [:each 'some condition'].</pre>

Rule	Description
Uses do: instead of collect: or select: 's	Checks for methods using do: instead of collect: or select:. The collect: and select: variants are preferred for clearly expressing intention.
Uses do: instead of contains: or detect:	Checks for methods using do: instead of using contains: or detect:.
Uses ifTrue:/ifFalse: instead of min: or max:	Checks for uses of ifTrue:/ifFalse: when it could use min: or max:. For example: <code>a < b ifTrue: [a] ifFalse: [b]</code> may be rewritten as: <code>a min: b</code>
Uses to:do: instead of do:, with:do:, or timesRepeat:	Checks for methods using to:do: when a do:, with:do: or timesRepeat: should be used.
Uses whileTrue: instead of to:do:	Checks for methods using whileTrue: when the shorter to:do: would work. For example, this common C-like code: <pre>i := 1. [i <= size] whileTrue: ["self do something with i". i := i + 1]</pre> can be written as: <code>1 to: size do: [:i "self do something with i"]</code>

Miscellaneous

Rule	Description
Doesn't use the result of a yourself message	Check for methods sending the yourself message unnecessarily. For example, the following statement doesn't need yourself, since it is not used: <code>aCollection addAll: #(a b c); yourself</code> If this statement were assigned to a variable, then the cascade with yourself would be needed to get the value of aCollection.
Inspect instances of "A + B * C" might be "A + (B * C)"	Checks for methods that might have precedence problems. Developers who are used to other languages often make mistakes when writing Smalltalk code since in Smalltalk all binary operations are performed left-to-right.

Rule	Description
Instance variables defined in all subclasses	Checks classes for instance variables that are defined in all subclasses. It is often better style to move the instance variable up into the class so that all the subclasses don't have to define it.
Long methods	Checks for methods that have more 10 statements (this check counts statements, not lines.)
Methods with full blocks	Checks for methods that contain full blocks or create a context with the <code>thisContext</code> keyword. These methods are a place where inefficiencies can creep in. For example, a common reason why a full block is created is because a block assigns a temporary variable that is not defined inside the block. If the temporary variable is only used inside the block, then the definition of the temporary should be moved inside the block. The "move to inner scope" refactoring can be used to correct this.
Non-blocks in <code>ifTrue:/ifFalse:</code> messages	Checks for methods that don't use blocks in the <code>ifTrue:/ifFalse:</code> messages. Developers new to Smalltalk may write code such as: <code>aBoolean ifTrue: (self doSomething).</code> instead of the correct version: <code>aBoolean ifTrue: [self doSomething].</code> Even if such expressions are correct, they cannot be optimized by the compiler.
Redundant class name in selector	Checks for the class name in a selector, e.g.: <code>openHierarchyBrowserFrom:</code> , which is a redundant name for <code>HierarchyBrowser</code> .
Refers to class name instead of "self class"	Checks for classes that have their class name directly in the source instead of <code>self class</code> . Using <code>self class</code> allows you to create subclasses without needing to redefine the method.
Sends "questionable" message	Check for methods that send messages which perform low level actions. For example, using <code>become:</code> throughout an application should be avoided. Also, messages such as <code>isKindOf:</code> suggest a lack of polymorphism.

Rule	Description
String concatenation instead of streams	<p>Check for methods that use string concatenation inside an iteration message. Since string concatenation is $O(n^2)$, it is better to use streaming since it is $O(n)$ - assuming that n is large enough.</p>
Unnecessary assignment or return in block	<p>Checks <code>valueNowOrOnUnwindDo:</code>, <code>valueOnUnwindDo:</code>, <code>ensure:</code>, and <code>showWhile:</code> blocks for assignments or returns that are the last statement in the block. These assignments or returns should be moved outside the block since they return the value of the block.</p> <p>For example, the code:</p> <pre>someMethod bos bos := BinaryObjectStorage onOld: 'test' asFilename readStream. [^bos next] valueNowOrOnUnwindDo: [bos close]</pre> <p>can be rewritten as:</p> <pre>someMethod bos bos := BinaryObjectStorage onOld: 'test' asFilename readStream. ^[bos next] valueNowOrOnUnwindDo: [bos close]</pre> <p>Having the assignment or return inside the block runs much slower than copying or optimizing blocks.</p>
Utility method	<p>Check for methods that have one or more arguments and do not refer to self or an instance variable. These methods might be better defined in some other class or as class methods.</p>
Variable is only assigned a single literal value	<p>If a variable is only assigned a single literal value then that variable is either nil or that literal value. If the variable is always initialized with that literal value, then each variable reference could be replaced with a message send to get the value. If the variable can also be nil, then it might be better to replace that variable with another that stores true or false, depending on whether the old variable had been assigned.</p>

Code Transformations

The following predefined code transformations are provided by the Code Critic (the patterns are defined in class ParseTreeTransformationRule):

Transformation	Description
"a >= b and: [a <= c]" -> "a between: b and: c"	Transform: a >= b and: [a <= c] to: a between: b and: c
= nil -> isNil AND ~= nil -> notNil	Transform = nil to isNil and transform ~= nil to notNil.
at:ifAbsent: -> at:ifAbsentPut:	Transform: aDictionary at: aKey ifAbsent: [aDictionary at: aKey put: anObject] to: aDictionary at: aKey ifAbsentPut: [anObject].
detect:ifNone: -> contains:	Transform: (foo detect: [:a a test] ifNone: [nil]) isNil to: (foo anySatisfy: [:a a test])
Eliminate guarding clauses	Transform methods ending with an ifTrue: or ifFalse: that have multiple statements inside the block, replacing them with ifFalse: [^self]. followed by straight-line code that was inside the block. For example: someMethod a isNil ifFalse: [self doSomething. self doAnotherThing] is transformed to: someMethod a isNil ifTrue: [^self]. self doSomething. self doAnotherThing
Eliminate unnecessary not	Transform: aTest not ifTrue: to: aTest ifFalse:

Transformation	Description
Move assignment out of showWhile: blocks	<p>Transform:</p> <pre>Cursor busy showWhile: [x := self someLongCalc].</pre> <p>to:</p> <pre>x := Cursor busy showWhile: [self someLongCalc]</pre> <p>This eliminates a full block.</p>
Move assignment out of ensure: blocks	<p>Transform:</p> <pre>[x := self aCalc] ensure: [self close]</pre> <p>to:</p> <pre>x := [self aCalc] ensure: [self close]</pre> <p>This eliminates a full block. Includes the ifCurtailed: variation.</p>
Move variable assignment outside of single statement ifTrue:ifFalse: blocks	<p>Transform:</p> <pre>aTest ifTrue: [x:=1] ifFalse: [x:=2]</pre> <p>to:</p> <pre>x := (aTest ifTrue: [1] ifFalse: [2])</pre>
Rewrite ifTrue:ifFalse: using min:/max:	<p>Transform:</p> <pre>a < b ifTrue: [a] ifFalse: [b]</pre> <p>to:</p> <pre>a max: b</pre> <p>(includes many variations)</p>
Rewrite super messages to self messages when both refer to same method	<p>Transform:</p> <pre>Singleton class>>default ^super new initialize</pre> <p>to:</p> <pre>Singleton class>>default ^self new initialize</pre> <p>if Singleton class does not define new.</p>
Use cascaded nextPutAll: instead of #, in #nextPutAll:	<p>Transform:</p> <pre>aStream nextPutAll: 'any ', 'time ', 'now '.</pre> <p>to:</p> <pre>aStream nextPutAll: 'any '; nextPutAll: 'time '; nextPutAll: 'now '.</pre>

7

Unit Testing

Overview

The VisualWorks browser includes a testing tool for using the SUnit framework. SUnit (short for Smalltalk Unit Testing Framework) is the *de facto* industry standard for developers to build test suites and attach them to their application classes.

SUnit supports the XP Unit Test testing practices. It is currently available for all Smalltalk dialects and is maintained by Camp Smalltalk. The SUnit distributed with VisualWorks is the most recent Camp Smalltalk version.

Additional information can be found at the following sites:

- *SUnit Camp Smalltalk*
<http://wiki.cs.uiuc.edu/CampSmalltalk/SUnit+Camp+Smalltalk>
- *Simple Smalltalk Testing: With Patterns*
<http://www.xprogramming.com/testfram.htm>
Kent Beck's original paper on SUnit.
- Various documents and papers are available at:
<http://sunit.sourceforge.net/manual.htm>

The SUnit testing framework makes two general structures available to developers:

Test Case

A *test case* is a collection of unit tests that are generally applied to a single class. the smallest unit of testing available, checking for specific conditions. Individual test cases are constructed by creating concrete subclasses of the abstract framework class TestCase.

Test Resource

A *test resource* represents a resource needed to perform a repeatable unit test (e.g., a temporary file or database instance). Test resources are associated with specific test cases, and may need to be prepared or initialized before testing can begin.

The unit testing methodology involves a novel approach to code development: test cases are defined first, *then* the application code is written and tested until the test passes.

The test case defines what the code should do or be like (the invariants of the program execution), thereby integrating test-writing into the heart of the development process.

Unit Testing in VisualWorks

The VisualWorks browser provides a simple, integrated interface for building tests and running them against your applications.

To begin using the unit testing framework, you must first load the `RBSUnitExtensions` parcel.

Using the Parcel Manager (select **Tools** → **Parcel Manager...** in the Launcher window), open the “Suggestions” category and click on **RBSUnitExtensions**; then select **Load...** from the <Operate> menu.

To view some example test classes, you may also wish to load the `SUnitTests` parcel.

Running Test Cases

To run the test cases defined in a class, use the browser to examine the test class’s definition. When examining any subclass of class `TestCase`, the browser displays the unit testing interface in the lower-right corner of the code tool:



To gather all tests in the selected class into a suite and then run them, click **Run**. The test results are displayed by the testing interface:

Passed: 6 run, 0 failed, 0 errors

You may run selected tests, by selecting only a single method, protocol, or a combination of protocols or methods. Similarly, you can gather a number of test cases into a single suite by selecting several classes (using the <**Shift**> key) before clicking on **Run**.

Although the unit test framework includes the notion of a *test suite*, developers using the VisualWorks testing tool don't need to worry about suites. The testing tool automatically bundles all selected test cases into a test suite before running the suite.

Creating a Test Case

Each new test case is represented using a separate class, each being a subclass of class `TestCase`. Any resources the test case uses are represented using instance variables (for details on resources, see [“Defining Test Resources” on page 7-4](#)).

For example, to create a simple test case:

- 1 Create a subclass of `TestCase` (e.g., `MyTest`)
- 2 In `MyTest`, create a protocol called `'tests'`
- 3 Create a test method (we'll call it `testExistence`), e.g.:

```
testExistence
  self should:
    [#MyNewClass] ifDefinedDo: [:value | true] elseDo: [false]].
```

The method `testExistence` now defines a single test case.

To run this single test, select the method `testExistence` and press the **Run** button located at the bottom of the browser code tool.

If `MyNewClass` does not exist, an error is shown in the lower-left corner of the browser code tool. To follow the practice of unit testing, you would next define the class and then run the test again to ensure it passes.

To run every test defined by a single test case, select the class (with no method selected) and press the **Run** button (you can also select a protocol to run all tests defined in that protocol).

Writing Test Methods

Class `TestCase` provides six basic methods for checking the specific conditions that allow a test to pass:

should: aBlock

Requires a 0 argument block that must answer a boolean. To pass the test, `aBlock` should answer `true`.

shouldnt: aBlock

Requires a 0 argument block that must answer a boolean. To pass the test, `aBlock` should answer `false`.

should: aBlock raise: anExceptionSubclass

Requires a 0 argument block. To pass the test, aBlock should raise an error of class anExceptionSubclass, i.e., a subclass of Exception.

shouldnt: aBlock raise: anExceptionSubclass

Requires a 0 argument block. To pass the test, aBlock should not raise an error of class anExceptionSubclass, i.e., a subclass of Exception.

assert: anExpression

To pass the test, *anExpression* should answer true. This is a convenience method that behaves like should:.

deny: anExpression

To pass the test, *anExpression* should answer false. This is a convenience method that behaves like shouldnt:.

Using should: and shouldnt:

Test cases may use the methods should: and shouldnt: to test for specific conditions allowing a test to pass.

The method should: takes a block parameter and allows the test to pass if the block returns true. For example:

```
testSomeCondition
  self should: [someCondition == true].
```

Alternately, you may test using the method shouldnt:, which passes if the block returns false.

For example, a test that uses shouldnt:raise: (a variant of shouldnt:) might look like this:

```
testCreation
  self shouldnt: [MyNewClass new] raise: Error
```

In place of should: and shouldnt:, you may also use the two convenience methods assert: and deny:. For example:

```
self should: [empty include: 5]
```

is equivalent to:

```
self assert: (empty includes: 5)
```

Defining Test Resources

Individual test cases will vary in their complexity. A test case may check against a simple set of predictions (e.g., “Should this condition be true?”), or it may involve a *test resource* that is used during testing.

A resource may be a simple data structure, or something as complicated as a database instance or a server connection. A single test case may require a number of different resources, which may either be created and destroyed for each individual test, or used for a series of tests.

Resources that are created and destroyed for each test are represented using instance variables in the test case class. In practice, an instance variable is added for each condition predicted by the resource.

Resources that are created and then used for a number of tests are represented using subclasses of `TestResource`. These must be initialized once before any test is run, and destroyed after all tests have been run.

As an example of a resource used for a single test, we might test the behavior of a `Set` using two resources, one to represent the “empty” condition, and one to represent the “full” condition. Thus, we would add two instance variables, `empty` and `full`, to our test case.

To initialize these two resources, we override the method `setUp` defined in class `TestCase`:

```
setUp
  empty := Set new.
  full := Set with: 5 with: #abc
```

Each time a test case is run, it begins by sending `setUp` and ends by sending `tearDown`. To ensure that tests are repeatable and predictable, `setUp` and `tearDown` are run once for each test. Subclasses of `TestCase` should override these two methods to initialize and release resources.

An example method that would use a resource to test the `add:` behavior of a `Set` might look like this:

```
testAdd
  empty add: 5.
  self should: [empty includes: 5]
```

This test adds the number 5 to the `empty` resource, and then uses `should:` to test that the `empty` resource now include the number 5.

Subclasses of `TestCase` are instantiated once for each test that is run, and any resources are initialized, used once, and then discarded.

Guidelines for Working with Test Cases

The SUnit methodology distinguishes between *unit tests* and *functional tests*. Unit tests are always repeatable, without regard for anything in the external environment (e.g., a database instance). Functional tests differ in that they may not be repeatable, since they test behaviors that depend on external resources which may change from one test run to another.

Generally, a test case is a collection of unit tests that are applied to a single class in the target application. However, occasionally a common set of behavioral tests are grouped into a single test case, thereby crossing class boundaries. These test cases are also more appropriately categorized as functional tests.

Criteria for Testing

The criteria for passing unit tests differ from those for passing functional tests. As a rule, any code that fails a unit test isn't ready for system integration. Integration should *only* take place when all unit tests pass 100%.

Functional tests, on the other hand, may not all pass 100%, and code that fails certain functional tests may still be integrated into the system. Generally, functional tests and unit tests are written at the same time, with a greater proportion of functional tests passing as the project progresses.

8

Trippy: Enhanced Inspector

Overview

As described in the Application Developer's Guide, the inspector is used to examine the state of objects in the system. To this extent the inspector can be used just like the traditional inspector dating back to the earliest versions of Smalltalk. However, the new inspector, also called "Trippy," can do many things the classic inspector could not.

This chapter describes and illustrates how to use Trippy's basic and additional features.

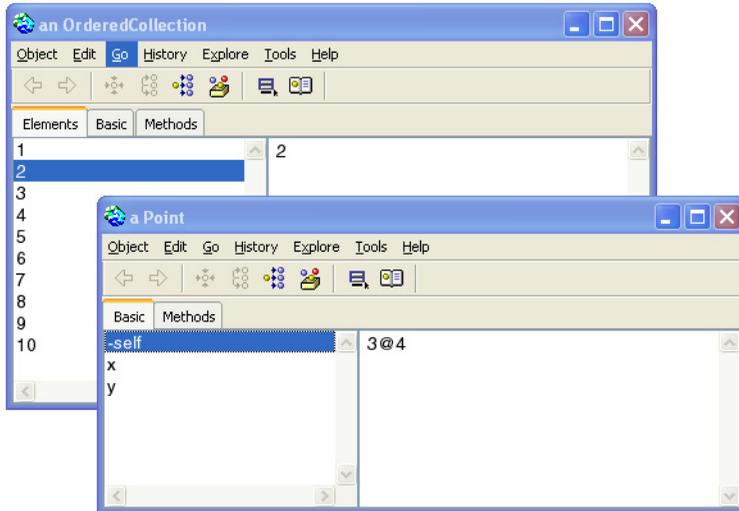
Basic Inspecting



To open an inspector, either send an inspect message to the object, or select the object and pick the **Inspect it** menu action (**Smalltalk** → **Inspect it in a workspace**) or click the Inspect toolbar button (shown at the left). For example, evaluate these expressions in a workspace to inspect a Point and an OrderedCollection:

(3 @ 4) inspect.
 (1 to: 10) asOrderedCollection inspect

The two windows that open have the normal inspector layouts.



The list on the left shows the component parts of the object. The parts of a Point object are the point itself and its x and y instance variables. Parts of an OrderedCollection are its elements, from first to tenth.

Selecting a part shows the printString representation of the part in the right hand side text view. Selecting multiple parts is possible (hold down **Ctrl** or **Shift** while clicking). Selecting all parts at once is very handy to get a quick overview of an object, so there is a toolbar button (shown at the left) and a keyboard shortcut **Ctrl+A** to do that.



Inspection Views

Above the parts list there is a set of tabs for selecting various views of the object being inspected.

All objects have a **Basic** view that shows all instance variables.

The OrderedCollection we look at also has the **Elements** view, which is initially selected. It is a higher-level "logical" view showing all elements of the collection but ignoring such implementation details as the firstIndex and lastIndex instance variables and unused indexed variables. These can be viewed in the **Basic** view

In relation to the traditional inspectors, the **Elements** view works like the `OrderedCollectionInspector` one would get when inspecting an `OrderedCollection`, while the **Basic** view is the same as doing the `basicInspect` to the `OrderedCollectionInspector`. A similar distinction applies to other objects that had special inspectors, such as `Dictionary`.

All objects also have a **Methods** view that displays a methods browser on the object's class. The **Inheritance** menu allows you to control how far down the inheritance tree inherited methods are shown, just like you control the display in the system browser.

For some objects, the **Basic** view may include extra parts which are not its instance variables. In fact, **-self**, which is shown as a part, is always shown but is not an instance variable. For a less obvious example, inspect a compiled method:

```
(Object compiledMethodAt: #printString) inspect
```

The **Basic** view includes **-bytecode** and **-source**. These are not really parts of the receiver, but they are included in the basic view as "virtual" attributes. For other interesting examples are an `Integer`

```
1234 inspect
```

or a `Character`

```
Character space inspect
```

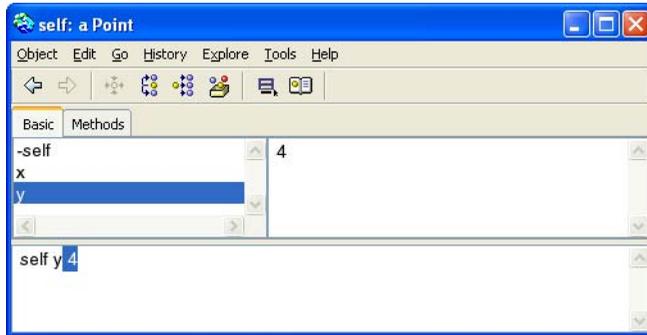
Evaluating and Editing

The text pane on the right of the inspector views is a normal `VisualWorks` text pane, so you can type a `Smalltalk` expression into the text view pane and evaluate it there. The evaluation context, the value of `self`, is the receiver object being inspected. However, whatever you type in this pane is lost as soon as you switch to another field.



Trippy provides a better way, an evaluation pane, that allows the option to save any expression you wish to evaluate. Click the “Toggle Evaluation Pane” button (shown to the left) on the toolbar to open an extra text area to the window.

Evaluation Pane



This area stays unchanged regardless of what is going on in the rest of the inspector. It can keep a set of expressions to be used over and over again.

In addition, if you **Accept** the contents of the text pane (<Operate> → **Accept**), the contents will be saved and displayed in all other inspectors. This saves the text in a shared variable within the image. Because that variable is used as the text model by all inspectors, the accepted text will appear in all open inspectors, as well as in those that will be open in future.

Editing Variable Values

You can change the value of any instance variable of an object you are inspecting. Select the variable, enter a Smalltalk expression in the right text pane, and **Accept**. The result of evaluating the expression is saved in the variable.

For example, select the `y` variable in the parts list, replace its value in the text pane, then do <Operate> → **Accept**. Select `-self` to see that the `y` coordinate of the point has changed.

The same happens if you change the value of an element of the `OrderedCollection`; The old element at the selected index is replaced with a new one.

You can even change the values of several variables or collection elements all at once. Use multi-select to select all of the parts you want to change, enter the new value into the right-hand text view, and **Accept**.

Copy and Paste

The **Edit** menu contains the usual **Copy** and **Paste** items. These actions copy and paste the object held by the instance variable or element in the selected parts list item.

Objects are copied to or pasted from Trippy's own clipboard, and instance of class Clipboard. To inspect the clipboard, evaluate:

```
Clipboard default inspect
```

Also, the `printString` representation of the object is copied to the system clipboard.

Note that these menu items *do not* copy or paste text from the text pane or evaluation pane. Copy and Paste operations on text in these panes can be performed using the usual `<Ctrl><C>` and `<Ctrl><V>` commands.

Add and Remove

The **Add** and **Remove** actions on the **Edit** menu add and remove parts in the list.

When using these commands bear in mind that, depending on context, the change might affect either the instance or the class. For example, if you are inspecting a collection, adding a part (element) adds to the instance. On the other hand, if the new part is a named instance variable, it is added to the class definition of the object being inspected. In the latter case, adding an instance variable changes the structure (shape) of all current and future instances of the class. Also, adding and removing instance variables is not allowed for some objects.

The same kind of editing is possible with an Array. Even though strictly speaking, an Array is not resizeable, Trippy makes it appear resizeable, allowing you to insert elements into an Array, or remove elements and make the Array smaller.

Undoing an Edit

If you make a value change to a part and then change your mind, you can undo the edit. Select **Edit** → **Undo**.

Undo is multiple-level, remembering and allowing you to undo a sequence of edits. Suppose you had replaced the fifth element of an `OrderedCollection` with 0, and then replaced multiple element values with 555. If you do an **Undo** at this point, the multiple replacement will be undone, reverting to the previous values. But, because we replaced the fifth element with 0 before that, **Undo** is still enabled. Doing another **Undo** will restore the fifth element to its original value.

Editing with Drag-Drop

The inspector is also enabled to allow editing using Drag-and-Drop actions, to either change the order of elements or to copy values of elements.

For example, open another inspector on this OrderedCollection:

```
(OrderedCollection with: 1 with: 2 with: 3 with: 4) inspect
```

To use drag-and-drop to change the order of elements in this collection, select the first two elements of the collection, drag them and drop *after* the last element in the list. Now select all elements to see the new order of elements. You can also drop the selections *between* elements.

To copy the value of one element to another, select one element, then drag and drop it *onto* another element. Dropping on an element replaces the element with the dropped value.

Drag-and-drop also works between inspectors. Open an inspector on another collection, such as:

```
(OrderedCollection with: 5 with: 6 with: 7 with: 8) inspect
```

If you select several elements in one OrderedCollection, then drag and drop them into the other collection, the elements are inserted, making the collection bigger.

The drop target can only be a single element or between elements. If you select several elements and attempt to drag and drop them onto a single element, whether in the same inspector or in another, a dialog prompts you to select the intended element from a list. Multiple-select targets are not supported.

Drag-and-drop is supported as widely as possible: between collections; between regular objects; between collections and regular objects.

Drag and drop is also extended to workspaces. You can add any object to a workspace as a local variable by dropping it onto the workspace or on the workspace's **Variables** page.

Protected Variables

Trippy allows any class to declare some or all of its instance variables as "protected," and indicates a protected variable with a hash mark.

For example, evaluate.

```
Object inspect
```

Notice that all of the instance variables are marked as protected. (A class is itself an instance of its metaclass, so can have instance variables.) Accidentally changing the value of, for example, its `methodDict` variable, can crash the system.

Protected variables can be changed, but only after answering **Yes** to a confirmation dialog.

To protect a class's instance variables, define a class method named `protectedInstVarNames` in the class that returns a collection of the names of the instance variables to collect.

Note that `protectedInstVarNames` methods that are defined throughout the superclass chain of a class are used to determine the variables of a given instance that need protection. Because of that, a method in a subclass can only add, but not remove, protection defined in a superclass. The method should answer a collection of variable names (Strings), possibly empty. Two useful ways to implement it are:

```
^self instVarNames
```

to protect all variables defined in this class, but not those inherited from superclasses, and

```
^self allInstVarNames
```

to protect all variables an instance of this class has, including those inherited from superclasses.

Exploring Objects

Most objects are complex, holding either other objects or references to other objects in instance variables or collection elements. The inspector is an object exploration tool that helps you examine these further objects, all in the context of the original object.

Diving into Object References

A common action is to follow a reference to another object. Double-clicking on a variable or element in the parts list *dives* into that part. The inspector is then refocused on the selected object.

For example, inspect something more complex than a simple collection, such as:

```
Window activeController inspect
```

This inspects the controller, an `ApplicationStandardSystemController`, of the currently active window, which is a workspace. Double-click on model. The inspector is now inspecting the workspace, an instance of `Workbook`, which is the controller's model. The title bar now shows the role of the new object—the name of the instance variable it was stored in or an index of an element—and the class of the object (**model: a Workbook**).



Notice that the first of two toolbar arrow buttons is now available. These buttons work like a web browser, moving the inspector focus forward and back along the trail of visited objects. Click the first one, the left-pointing arrow, to go back to the original controller. The right-pointing arrow button becomes available, because the model has been visited and so is now “ahead” on the visit trail. Click this button to return to scheduled controllers.

Now double-click the `uiSession` part to dive into it. The object under inspection is now a **ControlManager**. Open the **History** menu to see the trail of visited objects, described by their roles and class membership. Use this history list to jump to any object on the visit trail.

Exploring

Diving and traversing along the visit trail are important navigation aids but they are limited for exploring the wider network of relationships between objects. For example, if we wanted to inspect the window of the `VisualLauncher`, we would have to inspect something like

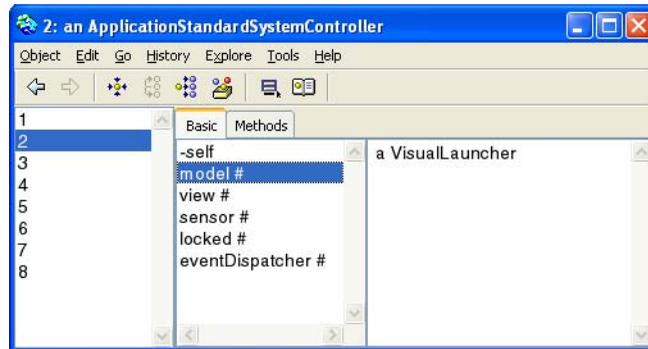
```
ApplicationStandardSystemController allInstances inspect
which inspects a collection. Then we would have to repeatedly dive into
elements of the collection, look at the models, back out to the collection,
then dive back in on another element, until we find one with a
VisualLauncher as its model.
```

Trippy includes additional views allowing us to explore the world outside any given object.

Siblings

In the case mentioned above, for example, we want to be able to easily inspect all controllers in the collection.

Inspecting the collection above, dive into one of the collection elements, so the inspector is focused on an `ApplicationStandardSystemController`. Select the **Explore → Siblings** menu item. The inspector transforms, adding an extra list on the left that shows the parts of the previous object, in this case the collection we were inspecting before diving into the current controller.



When you select to explore siblings, the added list box lists the parts of the object of which the current object is a part; those are its “siblings.” Selecting an element in the list refocuses the inspector on the right so that it shows details of the newly selected element. Now, to find the launcher, select `model #` in the inspector and go through elements of the collection until you find the one with a `VisualLauncher` as the model.

Once the desired object is located, you can focus the inspector on that object, closing the siblings list, by selecting **Explore → Focus**.

Parts

Similar to the Siblings view, is the Parts view (**Explore → Parts**). This view adds a list of the parts of the current object’s parts. So, instead of adding a list to the left containing the current object’s siblings, it adds a list to the right listing the parts of the selected part of the object.

For example, instead of diving into one of the elements in the collection of `ApplicationStandardSystemController` instances and then displaying siblings, you can start with the list and display the parts. The same view is displayed. The difference is that instead of going through the siblings of the original object, we want to go through its parts.

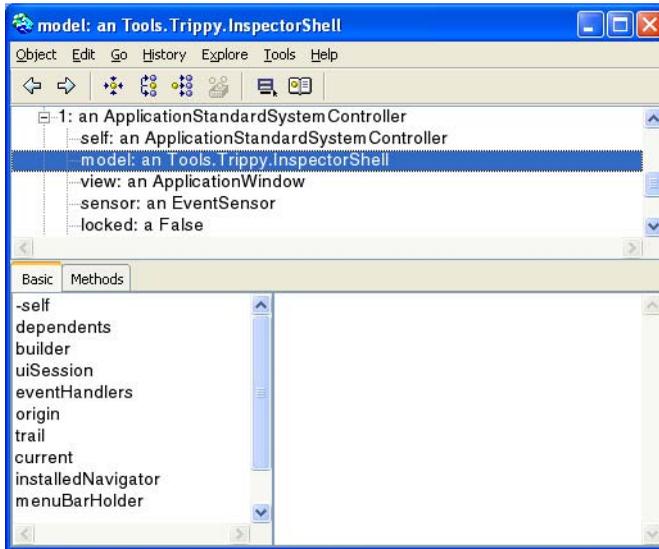
History Views

The forward and back navigation buttons have already been described.

As you dive into objects, go back, dive into other objects, and so on, these visits form a tree.

The **History** menu provides a list of the current branch of that tree. Rather than stepping forward and back one node at a time, use the **History** menu to select a specific node.

To view the entire tree, select **Explore → Visited**.



This adds a tree view to the top of the inspector showing the various objects you have inspected and the path you took getting to them. Select any of the nodes and the inspector displays it. You can then continue your explorations from that point.

Exploring a Window

When you are exploring a window, some additional inspection options are available.

The **Object** menu (and the part list popup menu when **-self** is selected) includes two extra items: **Raise** and **Close**. These options make the selected window active, or close and release it, respectively.

Previewing a Visual Part

A preview pane is added to the inspector, whenever a visual component or image is inspected. For example, try

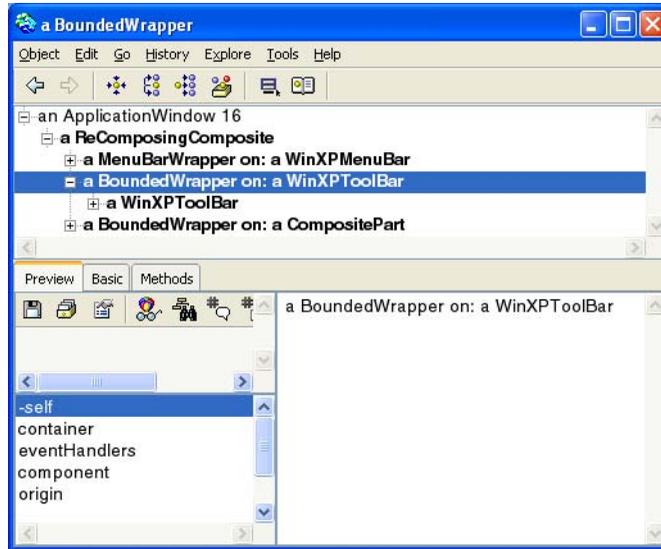
```
Image cincomSmalltalkLogo inspect
```

The preview pane shows the graphic, the window, the menu, or whatever the visual component is.

Exploring an Object Hierarchy

For objects that form hierarchies, an extra item is added to the **Explore** menu: **Component Hierarchy**. For example, visual parts (e.g., windows and widgets), classes, exceptions (with exception classes being members of

two hierarchies at the same time), parse tree nodes, UI specs, all form hierarchies. This command adds a tree list to the inspector showing the entire contained/containing hierarchy of objects.



This view greatly helps understand the structure of such objects. Using this view together with the **Methods** view provides a powerful tool for exploring complex object structures.

The inspector shows the component tree of the window, containing only the classes of those components. Expand the branches to expose more of the structure.

Viewing Related Objects

All objects have a **Go → To Class** menu item that focuses the inspector on the class of the current object. In addition, an object can tell the inspector about other important objects somehow related to it and add them to the **Go** menu.

For example, if you inspect an `ApplicationWindow`, the **Go** menu includes items that can take you to the important objects related to that window: the model, the controller, the main visual component, and the application. All of these objects are in fact instance variables of the window, though is much easier to use the menu than to find them in a list of more than variables.

Customizing the Inspector

Any object can publish actions to be added to the inspector menus.

By default, an object will have two views in the inspector: **Basic** and **Methods**, with **Basic** view showing **-self** and all named and indexed variables of the instance.

Trippy is a flexible tool, and allows you to provide additional representations of objects, as described in this section.

Define the Object printOn: Representation

Objects often reimplement the `printOn:` method to specify its print display. The inspector uses this method as well.

Add Displayed Attributes

You can add virtual attributes to the **Basic** view of an object by defining an instance-side method, `inspectorExtraAttributes`. The method should return a sequence of instances of either `DerivedAttribute` or `TextAttribute` (both classes are in the `Tools.Trippy` namespace).

A `DerivedAttribute` has an object value. Such a value can, for example, be dragged and dropped on a variable to store its value in that variable. An example of a `DerivedAttribute` is the `asInteger` attribute of a `Character`. **-self**, which shows up in any basic view, is a `DerivedAttribute` added by the inspector itself.

A `TextAttribute` is an attribute without an object value, but that displays informational text in the text view of the inspector. For example, the various radix print strings of an `Integer` are text attributes.

See implementors of `inspectorExtraAttributes` for examples.

Add Menu Actions

Adding selectable object actions to the **Object** and the part list `<Operate>` menus is done in a similar way: define an instance-side method `inspectorActions` answering a sequence of instances of class `Action` (defined in the `Tools.Trippy` namespace). See implementors of `inspectorAction` for examples.

To define objects to jump to using the **Go** menu, define a method `inspectorCollaborators` answering a sequence of instances of `Collaborator`.

Identify Hierarchies

To display an object as part of a hierarchy (or several hierarchies) in the hierarchy tree view, define a method `inspectorHierarchies`, answering a sequence of instances of `Hierarchy`. Browse implementors of `inspectorHierarchies` for examples in the system.

Add an Inspector Page

You can create your own inspector views to show special object properties and add them as pages in Trippy. Several system objects do this, such as `Dictionary`, `Array`, and visual components.

To add your inspector to Trippy, define an `inspectorClasses` method that returns a collection of inspector classes that can meaningfully display the object. Browse implements of this method for examples.

Provide Custom Object Views

To define your own view of an object, create a subclass of `Tools.Trippy.Inspector` and include it into the list of classes returned by `inspectorClasses` of your object.

Prototype-based Programming in Trippy

Besides being an enhanced inspector, Trippy is a good tool for prototype-based programming.

In the following example, we create a `Library` object to managing a collection of `Book` objects. We create it by modifying prototypical instances of `Book` and `Library` and testing the functionality as we proceed.

To begin, create two new classes, `Library` and `Book`, as usual.

To work on `Book`, inspect an instance of it. You can do this either by evaluating in a workspace:

```
Book new inspect
```

or, if an inspector is already open on the `Book` class, select **Go → To New Instance** menu item. The menu command is available whenever you are inspecting a class, and it creates and focuses on a new instance of the class.

So far the Book is just an empty shell. It needs, to start with, instance variables to store information like the title and the author. To add these, select **Add...** on the part list <Operate> menu of the variable list (or use **Edit → Add...**). In the dialog, enter the name “title” and click **OK**. The variable is added and selected. Repeat the process for “author.”

The variables are now created but their values are nil. You can add values to the variables in the inspector, as described earlier. For example, select title and type the String expression

```
'Moby Dick'
```

in the text view, then **Accept** it. Similarly, assign the String 'Herman Melville' to the author variable. (Normal setter methods can be added later.)

When you select **-self** in the inspector, the text just shows “a Book.” We can change that to show the title and author by reimplementing the #printOn: method for Book, and we can do this in Trippy. Switch to the **Methods** tab, add a “printing” protocol, and write a reasonable printOn: method, such as:

```
printOn: aStream
    aStream nextPutAll: (title, ', by', author)
```

Switch back to the **Basic** tab and see the change to the display.

Changing the value of an instance variable of a prototype “by hand” as we did above is fine for testing, but real objects will need a proper API, with proper accessor methods. Switch back to the **Methods** tab, add an “accessing” protocol, and create setter and getter methods for the instance variables. These will be simple methods to set or return the variable value, such as:

```
title
    ^title

and

title: aString
    title := aString
```

To test one of them, switch back to the **Basics** tab and select the title instance variable. Open the evaluation pane (**Options → Evaluation Pane** or hit <Ctrl>+<E>) and evaluate:

```
self title: 'Tempest'
```

The value of title that the inspector shows does not change automatically, because the inspector does not know the code we have evaluated has changed it. To update the display, select **Object → Refresh**, or press <Ctrl>+<R>.

The implementation of Book is now reasonably complete. But before moving on to Library, create another Book prototype to add to our library. In the evaluation pane of the current inspector, type “self copy” and **Inspect It**. This second inspector now holds onto a copy of the original book, initially with the same title and author, but we can change that.

Now open a third inspector on a fresh instance of Library. In a workspace or the evaluation pane of an inspector, evaluate

```
Library new inspect
```

So the library can hold onto its books, add an instance variable, books. It is added and its value is nil.

Because the library will hold more than one book, its value should be a collection. We could create the value in the inspector, like we did before, but that is only temporary. Any instance of Library will have a collection in that variable. What we really need is an initialization method.

Switch to the **Methods** tab of the Library and add a “initialize-release” protocol at the instance side. Then write an initialize method to initialize books to hold an OrderedCollection, such as:

```
initialize  
  books := OrderedCollection new
```

Instead of using the evaluation pane to try the new initialization logic, simply select the initialize method, open its <Operate> menu, and select **Send It**. Switch to the instance side and make sure the instance was initialized properly.

To finish with the initialization logic, add the usual new method with ^{^super}new initialize on the class side of Library. This might already exist, if you had **Initializer** checked in the class creation dialog.

To add our books to the Library, dive into the books collection, so the empty collection is displayed. Now, drag and drop **self** from both Book inspectors into the collection. Switch back to Library and verify that the books are where they should be.

This ends this simple demonstration of using Trippy for protocol programming. As you continue your explorations, you will find additional ways to use this inspector to simplify your work.

9

System Profilers

Profilers are tools that report system resource use by a block of code. The Time Profiler is useful for identifying portions of your code that consume large amounts of processing time. The Allocation Profiler performs a similar service for memory usage. Both single-process and multi-process profiling is supported.

All profilers rely upon a statistical sampling heuristic to estimate, rather than on instrumentation to directly measure, the resources consumed by a process. Multiprocess profilers distribute the probes that are used to estimate resource consumption over several processes, and the distribution may be uneven. Running multiprocess profilers does cause garbage collection and other maintenance processes to run more frequently than otherwise. These facts should be kept firmly in view when setting up multiprocess profiling runs and when estimating the reliability of their results. Within these limitations, multiprocess profilers have proven useful in tuning web applications involving many hundreds of processes.

Loading the Profilers

The profiler tools are contained in two parcels: AT Profiling Core and AT Profiling UI. AT Profiling Core is a prerequisite for AT Profiling UI, so both parcels are loaded when you load AT Profiling UI (or when you load the All Advanced Tools).

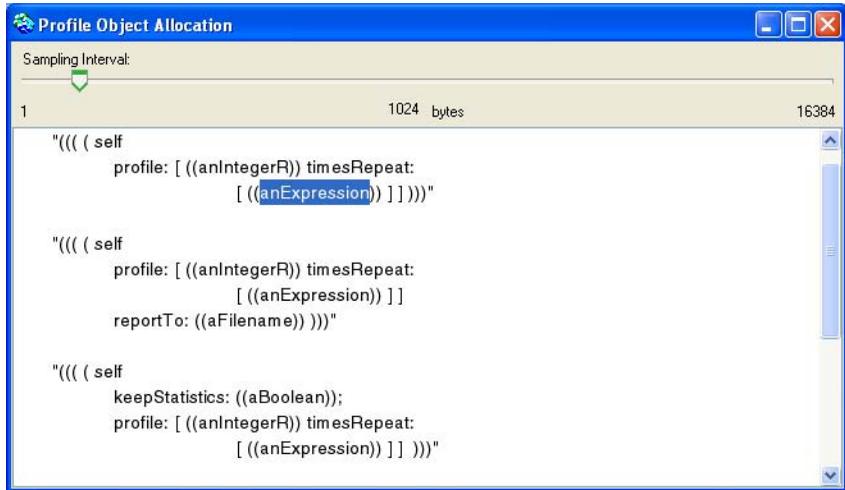
For most development environments, load AT Profiling UI to get the entire profiling tool.

In the future, a detachable, distributable profiler will be available, and only the core will need to be loaded in the image being profiled. Until then, the AT Profiling Core parcel is not independent.

Opening a Profiler Window

Several Profiler UIs are available as submenus of **Tools → Advanced → Profiles** in the Visual Launcher. For example, to open an Allocation Profiler, select the **Allocations** submenu item.

Each profiler window has a code view for entering the code to be analyzed, and a slider control for adjusting the sample size.



By default, the window shows explanatory text, as a guide to usage. To display only profiling code templates, evaluate in a workspace:

```
Profiler showTemplates: true
```

The templates provide schematic expressions. Replace the place holders, for iterations, the expression to profile, and others as needed. The remainder of this section will assume the templates are displayed.

Profiling a Block of Code

To profile either the time or memory usage of a block of code, open the appropriate profiler and enter the Smalltalk expressions in the code view of the profiler in a profile block. Templates are provided to help you.

For example, suppose you wanted to find out what proportion of the memory allocated by the Date today method. Open an Allocations profiler (**Tools → Advanced → Profiles → Allocations**). Several templates are displayed:

```
"((( self
  profile:[ ((anIntegerR)) timesRepeat:
    [ ((anExpression)) ] ] )))"

"((( self
  profile:[ ((anIntegerR)) timesRepeat:
    [ ((anExpression)) ] ]
  reportTo: ((aFilename)) )))"

"((( self
  keepStatistics: ((aBoolean));
  profile:[ ((anIntegerR)) timesRepeat:
    [ ((anExpression)) ] ] )))"

"((( self
  keepStatistics: ((aBoolean));
  samplingInterval: ((anIntegerS));
  yourself )
  profile:[ ((anIntegerR)) timesRepeat:
    [ ((anExpression)) ] ]
  onExitDo: ((aBlock)) )))"
```

Because the profiler employs statistical sampling, several iterations should be used to produce results, so replace `anIntegerR` with an integer value sufficiently large to produce good results (some experimentation may be necessary). Then, replace `anExpression` with the expression to be profiled. That is sufficient for the first template; the others provide additional options.

To complete the example, then, you might use the first template with the following substitutions:

```
"((( self
  profile:[ (( 1000 )) timesRepeat:
    [ (( Date today )) ] ] )))"
```

Then select the expression and evaluate it with **Do it** in the <Operate> menu. After the expression is executed, the results of the analysis are displayed in a new window.

For an explanation of the report, see [“Analyzing the Profiler Report.”](#)

Adjusting the Sample Size

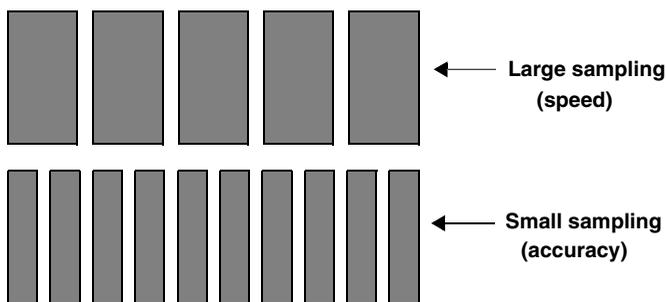
Repeating the code to be profiled, as shown above, increases the accuracy of the sampling. An additional mechanism to control accuracy is to adjust the sampling size, using the slider control in the Profiler.

A profiler typically provides only an approximation of the time or memory being used by each method that is called. It does so, in effect, by monitoring the process at a regular interval, called the *sampling interval*. For example, if a baby-sitter checks in on children playing in their room every half hour, the sampling interval is 30 minutes.

At each 30 minute check point, the babysitter has to assume that the behavior of the moment has been going on for the past half hour. By reducing the sample size to 15 minutes, the babysitter will get a more accurate picture of the children's activities, though it will cost more time and effort.

The sample size can affect the accuracy of the results dramatically. Reducing the sample size improves the accuracy, but may slow down the profiling run disproportionately. Setting the sample size to zero, for example, causes the profile to be updated after each indivisible chunk of time or memory is used, which can be very time-consuming. In most situations, the default sample size provides adequate accuracy without slowing things down unnecessarily.

To reduce the sample size (for brief processes), move the slider to the left until the desired numerical size is shown below the slider. To increase the sample size (for time- or memory-intensive processes), move the slider to the right. (To move the slider, place the cursor on the dark bar, press and hold the <Select> button on the mouse, then move the mouse to position the slider.)

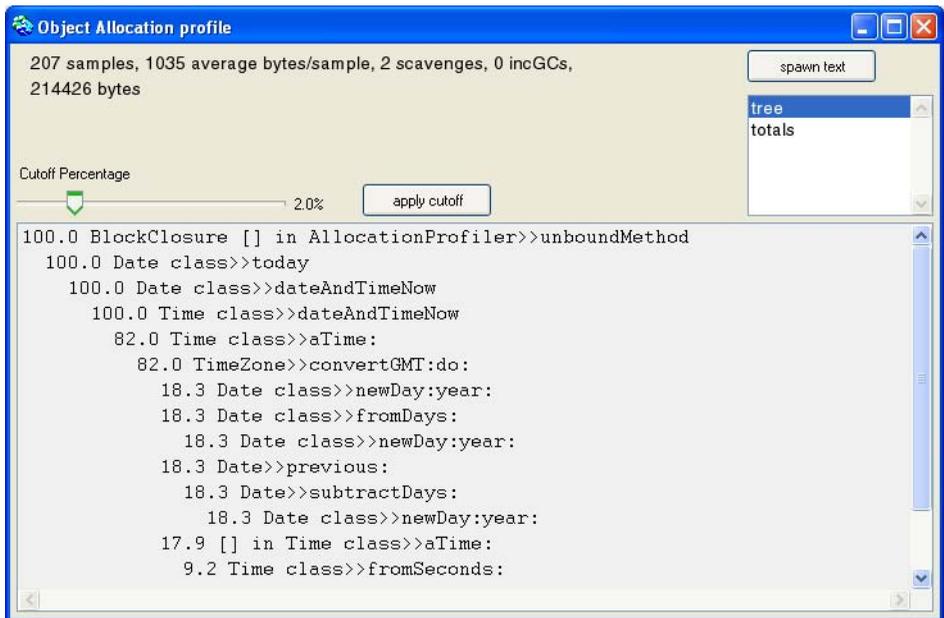


In the example used above, printing today's date in the transcript, the process is so light in its memory usage that the default sampling interval of 1024 bytes is inappropriate. The process is only monitored a few times, resulting in misleading allocation statistics. The obvious solution is to reduce the sample size so the process is checked more frequently.

Analyzing the Profiler Report

After the process that you are profiling has finished executing, the profile report is displayed in a window having the following components:

- A record of the sampling parameters.
- A slider for changing the cutoff percentage and a button for applying a new percentage.
- A text view for displaying the statistics.
- A list providing selections of a **totals** view or a **tree** view, for selecting the type of statistics to be displayed in the text view.



At the top of the profile window, a set of statistics display useful information about the profiling run, which include:

- Number of samples
- Sample size
- Total bytes consumed (allocation profile)
- Total milliseconds consumed, in both elapsed and processor time (time profile)

This information is useful in judging whether a change in the sampling interval will prove fruitful—because relatively few samples were taken, for example. This information also serves to label the profile, distinguishing it from profiles generated by other sampling runs on the same code.

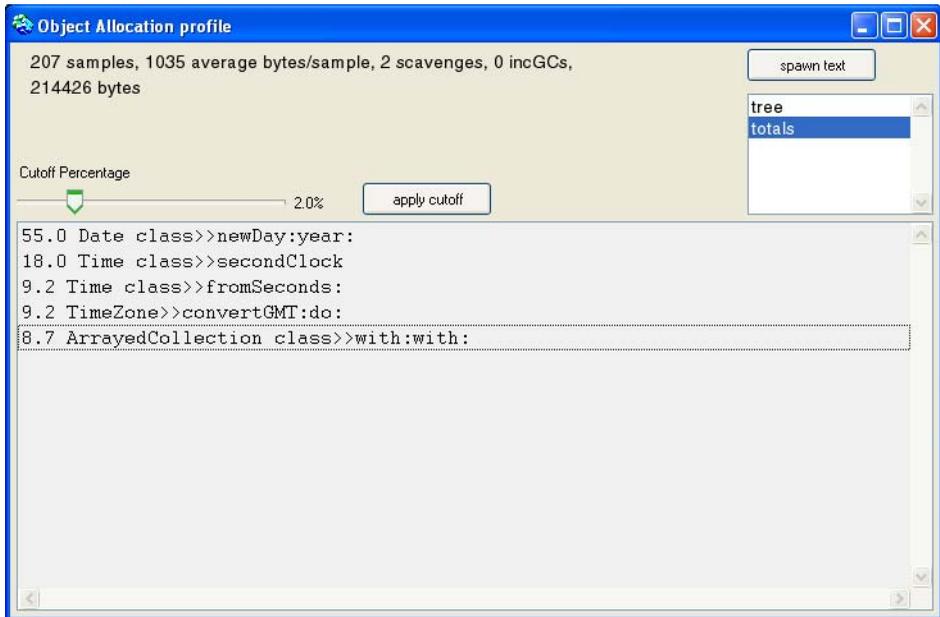
Tree Report View

When the **tree** view is selected, the text view displays a listing of consuming methods that were called during the process. This listing is useful for locating the places in your code that consume the most time or memory, and therefore merit your optimizing attention.

Each method selector is preceded by a number representing the percentage of system resource (bytes or milliseconds) consumed by that method. The tree is displayed in the form of an indented list—each method is indented under its calling method.

Totals Report View

When the **totals** switch is selected, the text view displays a list of the fundamental object-creating methods that were called, with the percentage of system resource consumed by each.



For example, a process that deals with graphics might make many calls to the `x.y:` method in the `Point` class. That activity would be summarized here. If you felt `Point` was taking an inordinate amount of time or memory to get the job done, you might investigate alternative coding paths that would generate fewer messages to `Point`.

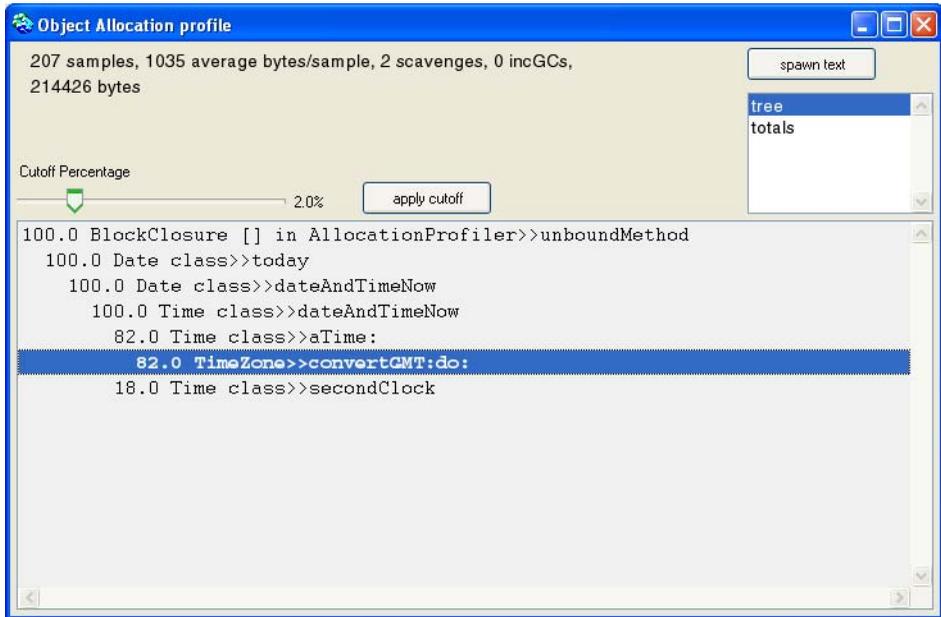
Adjusting the Cutoff Percentage

Only those methods that consumed more than a threshold percentage of time or memory are shown. The default is 2 percent, meaning any method that consumed less than 2 percent of the time or memory is excluded from the listing. In effect: “If it’s smaller than this, don’t bother me with it.”

To get finer detail in the profile, reduce the cutoff percentage by moving the slider to the left. To restrict the profile to the methods that consumed larger chunks of time or memory, move the slider to the right. After you have changed the position of the slider, apply the new cutoff by clicking on the **apply cutoff** button.

Contracting and Expanding the List

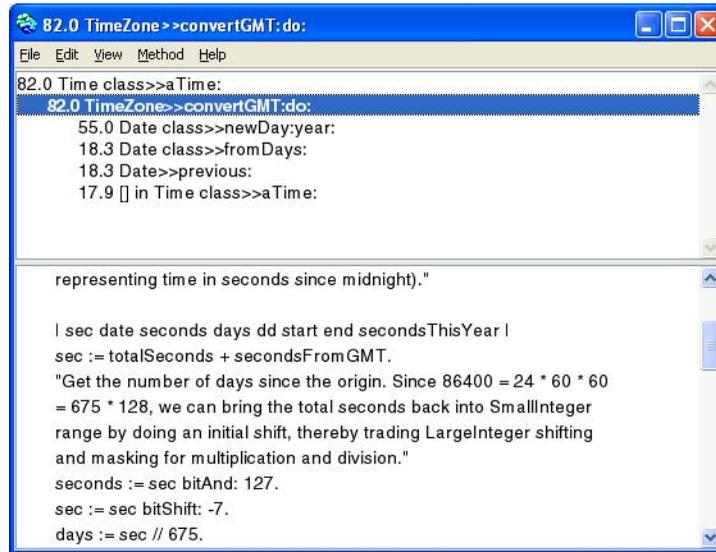
Another means of making the list more manageable in size is to temporarily remove selected subhierarchies from the display. To do so, select the method above an unwanted subhierarchy and then use the **contract fully** command in the <Operate> menu. The selected method redisplayed in boldface, indicating that it can be expanded to show more detail.



To restore detail under a contracted method, use either **expand** (for a single level of called methods) or **expand fully** (for the entire subhierarchy) in the <Operate> menu.

Spawning a Method Browser

To examine the body of a method in the tree, select the desired method and then use **spawn** in the <Operate> menu. A method browser will be opened in a separate window. Besides the selected method, which is listed in boldface in the new window, the browser will list parent and child methods when appropriate.



While the browser offers most of the features of a code view, including text editing, you cannot recompile an edited method (via **accept**) in this window, because that could cause confusion about the state of the code at the time of the profile.

You can also browse **senders** of the selected message, **implementors** of the method, and implementors of **messages** contained in the selected method. These operations are the same as in the System Browser.

Multi-process Profiling

Multi-process profiling provides profile reports for an expression being evaluated in multiple process rather than a single process. The multi-process profilers distribute the probes among several processes to evaluate the resources used by an expression running in a multi-processes context.

The expressions provided by the templates for MultiTime and MultiAllocations profiling reflect the difference between multi-process profiling and single-process profiling. Rather than evaluating the

expression within a single profile block, the profiler is started, then the expression is evaluated, possibly repeatedly, and then the profiler is stopped:

```
"(((self startProfiling )))"  
"((( ((anIntegerR)) timesRepeat: [ ((anExpression)) ] )))"  
"((( self stopProfiling )))"
```

The report window opens once the profiler is stopped. The resulting report provides one more view option, **tree grouped by priority**.

Profiler Programmatic Interface

At times it may be useful to profile larger blocks of code within the context of an application. The Profiler API allows you to do this, invoking the Profilers apart from the Profiler windows.

The interface classes are Profiler and its subclasses:

```
Profiler  
  AllocationProfiler  
    MultiAllocationProfiler  
  TimeProfiler  
    MultiTimeProfiler
```

The primary messages for invoking a profiler on code are the same as shown in the templates in the Profiler UI. The main difference is that, outside of the UI, you cannot simply refer to self, but have to send messages to either the appropriate Profiler class or an instance, depending on the message.

For example, to run the profile used earlier, an Allocation Profile on Date today, you would send:

```
AllocationProfiler profile:[ 1000 timesRepeat: [ Date today ] ]
```

However, there is no class method for setting the sampling interval. To change the interval, do:

```
| profiler |  
profiler := AllocationProfiler new.  
profiler samplingInterval: 2056.  
profiler profile: [ 1000 timesRepeat: [Date today ] ]
```

For the complete API, browse the **public api** protocol on both the instance and class side of these classes.

10

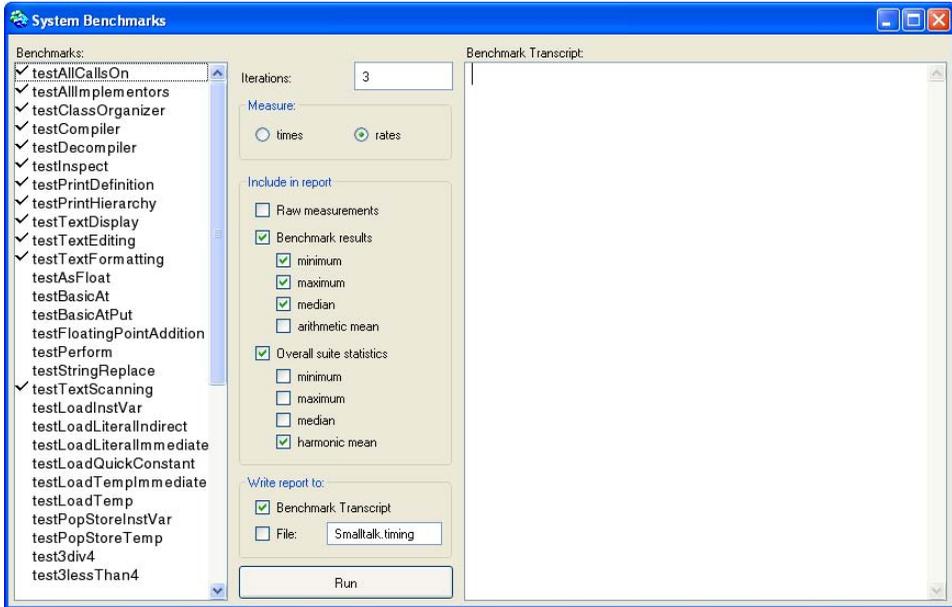
Benchmarks

The Benchmark class provides a framework and a convenient interface for running benchmarks to compare your application's performance across versions and in various operating environments. A simple subclass of Benchmark can be built to run the benchmarking tests. As an example, we have provided a subclass called SystemBenchmark, which contains updated versions of the historic test suite we at Cincom use to compare system performance on different platforms.

This chapter describes the reusable interface and related mechanisms provided by the Benchmark class, using the SystemBenchmark subclass as an example. The final section then explains how to implement your own benchmarks.

Using the Benchmark Interface

To open the example System Benchmarks window, select **Tools**→**Advanced**→**Benchmarks**.



The System Benchmarks window has three views, arranged side by side. The benchmarks view, on the left side, lists the available benchmark tests. The parameters view, in the center, contains a variety of buttons and fill-ins for controlling report attributes. The **Run** button located below the list view begins execution of a test suite. The benchmark transcript, on the right, displays execution progress and the final benchmark report.

Assembling the Test Suite

Although a benchmarking run can be limited to a single type of test, such as adding 3 + 4 thousands of times, a run frequently involves a suite of several related tests. You can use the benchmarks view to select the tests you want to include in a run. To select an individual test, just click on it with the <Select> button; click again to deselect it. A check mark appears in the margin next to each selected test.

Selection Techniques

To select multiple adjacent tests, hold down the <Shift> key while dragging the cursor through the desired tests (the check marks will appear after you release both the mouse button and the <Shift> key). To deselect multiple adjacent tests, hold down the <Control> key while dragging through the test names.

To cancel all selections, use **clear selections** in the <Operate> menu; use **select all** to include all of the tests. The subclass can define a default suite of tests—in our example, SystemBenchmark uses as defaults the tests used by VisualWorks development for standard comparisons of platform performance. You can reset the test suite to the defaults at any time by selecting **reset to default** in the <Operate> menu. To summarize these operations:

Selection techniques for system benchmarks

Operation	Description
click <select> button	Select and deselect a single test
<Shift> + drag <select>	Select multiple tests
<Control> + drag <select>	Deselect multiple tests
select all	Select all tests
clear selections	Deselect all selected tests
reset to default	Select default tests

Setting the Report's Granularity

At the end of each benchmarking run, a report is generated containing statistics accumulated during the tests. Three buttons at the top of the parameters view control the level of detail in the report, as follows:

Raw Measurements

Details about each iteration of each test method. This information can be used to discover significant variations among iterations. The first iteration of an operation, for example, might consume a disproportionate amount of time because it might not take advantage of compiled-code caching.

The following times, for example, were reported for three iterations of two tests in the SystemBenchmark suite: text displaying and text replacement.

Individual benchmark results (three iterations)

Benchmark	Minimum	Maximum	Median
TextDisplay	136.170	145.455	138.979
TextEditing	82.7451	84.7389	84.7389

Overall Suite Statistics

A summary for the entire suite, the purpose of creating a suite in the first place is to measure the performance of some subsystem. Benchmarking provides a weighted average for the performance of that subsystem, which you can then use to compare with an identical benchmarking run under different operating circumstances.

For the weighted average, the report displays the same columns as for the individual statistics. For example, if you elect to display only the median value for individual benchmarks, only the median value for the suite-wide statistic will be shown.

Benchmark suite results (three iterations)

Rating Type	Minimum	Maximum	Median
Minimum	118.539	126.309	125.558
Maximum	139.13	142.222	142.222
H-Mean	116.364	119.425	118.321
Median	118.539	126.309	125.558

Let's use the minimum H-Mean (harmonic mean) to illustrate the derivation of these statistics further. Each time the test suite is performed, the individual test results are converted to rates and then combined mathematically to arrive at the harmonic mean score for that iteration.

The suite was performed three times, in our example, so three such harmonic means are derived. The minimum H-Mean represents the lowest of the three scores. Similarly, the maximum H-Mean is the highest of the three, and the median H-Mean is the median (or middle value) of the three.

Choosing Types of Statistics

The two summary sections of the report can include different types of statistics. You control which types are included in the report by selecting buttons in the parameters view. The types of statistics are as follows (i represents the number of iterations):

- Minimum—the result from the best-performing iteration.
- Maximum—the result from the worst-performing iteration.
- Arithmetic mean—the average of all iterations; sum/i .
- Harmonic mean—The number of iterations, divided by the sum of the inverses of the weighted results for the separate iterations.

$$i/[(1/\text{result}_1) + (1/\text{result}_2) + \dots]$$

- Median—the value that is midway through a ranked list of the scores. For example, if you specify five iterations, the median is the third element in the sorted collection of scores.

The harmonic mean is only useful when summarizing overall performance, so it is only available under the **Overall suite statistics** check box. Under the heading **Benchmark results** check box, the arithmetic mean is only offered when you select the **times** switch; when the **rates** switch is selected, the harmonic mean is offered.

Setting the Report Destination

The report can be displayed in the benchmark transcript view, stored in a disk file, or both. Use the buttons under the heading **Write report to:** in the parameters view to select one or both destinations. You can provide the name of a file in the input field. The file will be created in the start-up directory unless you specify an absolute or relative pathname.

Setting the Number of Iterations

The test suite can be repeated as a means of improving the accuracy of the results. By default, the iteration count is set to three. To change the number of iterations, type the desired number in the input field labeled **Iterations**.

The number of iterations represents the number of times the test suite will be repeated—this is not to be confused with repetitions that are hard-coded into a given method. For example, the `test3plus4` method repeats the $3 + 4$ operation 100,000 times for each iteration, so three iterations would cause the operation to be repeated 300,000 times.

In some situations, a single iteration may produce more interesting results. For example, a method might take a relatively long time to execute on its first pass, but run much faster subsequently. However, if your application calls the method only infrequently, the first-iteration results might prove more illuminating.

To begin execution of the testing run, click on the **run** button. If your window manager is configured to prompt you for placement of windows, you might consider turning off that feature before running the default test suite or other suites involving window-displaying operations. However, prompt-for-placement can be left on without affecting the results.

Creating a Benchmark Subclass

The benchmarks are implemented via the following four classes, all of which are subclasses of `Object`:

- `Benchmark`, and its subclass `SystemBenchmark`
- `BenchmarkTable`
- `BenchDecompilerTestClass`

Benchmark Superclass

`Benchmark` is an abstract superclass whose protocol provides the interface we have been describing, as well as the timing and statistical analysis facilities for a benchmarking run. It has instance variables for remembering the report parameters as selected in the interface, and the test results as they are accumulated. `Benchmark` also provides the reporting protocol, making use of `BenchmarkTable` (described further below).

SystemBenchmark Subclass

Subclasses of `Benchmark`, such as `SystemBenchmark`, are responsible for providing the specific tests to be run. See the methods that begin with the word “test” in `SystemBenchmark` for examples.

In addition, subclasses must implement the following accessing messages:

`benchmarkLabelForSelector:`

`benchmarkSelectors`
`initiallySelectedBenchmarks`

Subclasses may also need to override Benchmark's weighting protocol, to establish relative weights for test methods and to convert the results to an appropriate rate; and the defaults protocol, which determines the default selections in the user interface.

BenchmarkTable Class

BenchmarkTable provides two-dimensional reporting capabilities that might well be useful to other applications, though the code requires extensions to make it more generally useful. It holds onto a report name, a collection of column labels, and a collection of rows. Each row is assumed to be a collection itself.

The protocol is tailored to the needs of the benchmark reports, though it provides a subset of a more generally useful set of behaviors.

BenchDecompilerTestClass Class

BenchDecompilerTestClass is a holder for methods that are decompiled during the `SystemBenchmark>>testDecompiler` benchmark. The code in the methods has no functional value—in fact, it is obsolete.

11

Class Reports

Overview

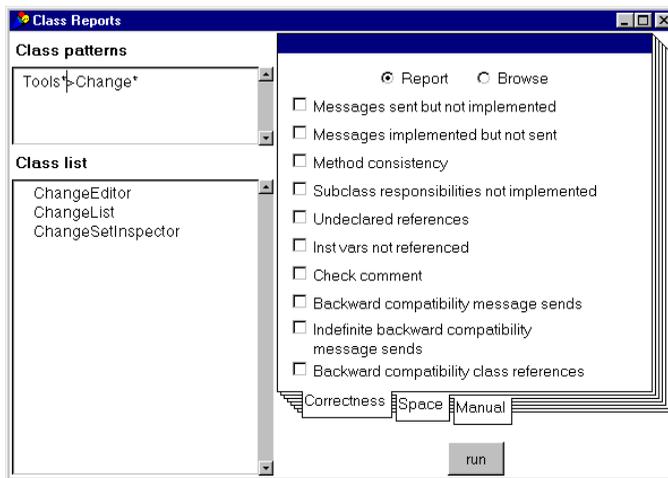
The Class Reports tool performs a variety of automated checks on specified classes and helps you:

- Repair common coding errors.
- Estimate memory requirements of your application.
- Document your code.

Class Reports is a specific tool that is built on top of a set of general system-analysis capabilities.

Creating Class Reports

To open a Class Reports window, select **Tools**→**Advanced**→**Class Reports**.



The Class Reports window contains the following components for defining the contents of the report:

- A Class Patterns view for roughly defining the classes to be checked.
- A Class List view for selecting individual target classes.
- Three switches for choosing a type of report.
- Depending on the type of report selected, two extra switches may be provided for choosing the output destination.
- Depending on the type of report and the output destination, additional options may be provided.
- A button labeled **run** for launching a scan-and-report sequence.

Selecting the Target Classes

You can generate a report for a single class, all classes or any list of classes. Keep in mind as you assemble your list that the amount of time required to produce a report increases with each added class.

Use the Class Patterns view to make a rough cut at the list. Enter one or more wildcard patterns, one per line. Each such entry can contain a class category component and/or a class component. If both components are

present, separate them with a greater-than symbol (>). Then choose **accept** in the <Operate> menu, or click the **run** button, to display all classes matching those criteria in the Class List view.

Wildcard patterns are not case sensitive; an asterisk (*) stands for any string, and a number sign (#) stands for any single character. You can also use the **paste** command to insert a list of patterns that you use frequently.

The following examples are all valid class patterns:

Valid class patterns

Tools*	Classes in categories beginning with 'Tools'
tools*	Same as above
Tools-Misc>*	Classes in the Tools-Misc category
Tools*>Changes*	Classes beginning with 'Changes' in categories beginning with 'Tools'
Changes*	Classes beginning with 'Changes'
ChangesList	The class name ChangeList

Then, in the Class List view, click on the desired class or classes to highlight them for inclusion in the report. Use the **add all** command in the <Operate> menu to select all of the classes in the list at once; use **clear all** to deselect all of them. To select a range of classes, hold down the <Shift> key while dragging through the desired class names; to deselect a range of classes, hold down the <Control> key while dragging.

Locating Coding Errors

To scan the selected classes for coding errors, select the **Correctness** switch in the upper left corner of the Class Reports window. Two new switches will appear, labeled **Report** and **Browse**. When the **Report** switch is selected, ten report options are displayed. Each option has a check box, and you can check any number of them to build up the desired report. When the **Browse** switch is selected, eight of the options are offered—the other two are only appropriate for report output.

Class Report Options

Messages Sent but Not Implemented

Each method in the class is checked to make sure that every message sent is implemented somewhere in the system. No attempt is made to assure the appropriateness of the implementor. For example, a self grok message is acceptable even if grok's implementor is not in the target class or its superclass hierarchy.

Methods that send an unimplemented message are reported or, in **Browse** mode, listed in a browser for examination and possible correction.

Messages Implemented but Not Sent

Each method in the class is checked to make sure that its selector is sent by at least one calling method.

Defining what it means for a message to be "sent" is problematic. As an extreme example, one could have code that says self perform: (a,b) asSymbol, where a and b are variables that hold 'foo' and 'bar', respectively. This code, then, sends the message foobar, but no practical analyzer can figure this out. So system tools have to take a particular stand as to what it means for a message to be sent.

In the case of this facility, the stance taken is exactly the same as that taken by the **senders** and **messages** facilities in the System Browser: a message is sent if some compiled code has the message selector as a literal. It will be a literal if the selector is used in code (e.g., self foobar), or if the symbol exists in literal form (e.g., self perform: #foobar). (The exception to this rule is a set of special selectors known by the compiler classes. These selectors are always considered to be sent, even if they do not appear as literals anywhere.)

As a result, the facility may falsely report that some implemented messages are not sent, so the report should be used as a guide. The above example is, of course, poor programming style.

Methods that are not sent are reported or, in **Browse** mode, listed in a browser for examination.

Method Consistency

When two messages sent to the same instance or class variable assume different object types, a conflict is reported.

Similarly, when a temporary variable is used to hold two very different kinds of objects (considered bad form) and thus is sent incompatible messages, a conflict is reported.

The current value of each class variable, pool variable, and global variable is also tested to be sure its class implements the messages that are sent to it.

Finally, an inconsistency is reported when a message is sent to self that is not understood by the self object.

When inconsistent methods are found, they are reported or, in **Browse** mode, listed in a browser.

Subclass Responsibilities Not Implemented

Each method that consists of a self subclassResponsibility message motivates a check of each leaf subclass to make sure it owns or inherits a reimplementations of that message.

Note that abstract subclasses need not implement these messages—in such cases, the report will falsely report errors, so use the report as a guide.

Offending methods are reported or, in **Browse** mode, listed in a browser.

Undeclared References

Each method in the class is checked to verify that no undeclared literals are used. Offending methods are reported or, in **Browse** mode, listed in a browser.

Instance Variables Not Referenced

Each instance variable is checked to make sure it is referenced by at least one method. Unreferenced variables are reported; this option is not available in **Browse** mode.

Check Comment

The class comment is checked to make sure it mentions all instance variables, class variables, and class instance variables that are in the class definition.

The comment is expected to follow a particular syntax:

- Any amount of plain text followed by a line that says “Instance Variables:”.
- After that line, there should be a line for each instance variable, containing the variable’s name followed by one or more spaces and tabs, followed by a “type” specification in angle brackets, followed by one or more tabs and spaces, followed by text describing the variable.

- If the class has indexed instance variables, include another line as described above, substituting "(indexed instance variables)" for the variable name.

The type specification is typically one or more class names, or nil, separated by vertical bars. In place of class name, you can also use "ClassName of: OtherClassName", for example "Array of: Boolean". The syntax allows more complicated descriptions; for more information, see the method comments in `Parser>>typeExpression` and `Parser>>simpleType`.

If the class defines any class variables, the comment should have a section similar to the instance variable section. The heading line is expected to say "Class Variables:".

Finally, if the class has messages defined as self subclassResponsibility, these messages should be listed in a section with "Subclasses must implement the following messages:" as its heading.

The parsing of class comments is somewhat rigid and sometimes what appears to be a valid comment will generate errors in this report, so use the report as a guide. For example, if a type description does not fit on one line, or if the variable description does not start on the same line, the facility will report these as errors.

For instance variables, the facility performs a protocol test:

- All messages sent to each instance variable are verified as being implemented for the named class (or, if more than one class is named, for at least one of them).
- If the class has existing instances, each variable is expected to hold an object of the named type.
- For each class variable, the current value is expected to be an object of the named type.

This option is not available in **Browse** mode. If a comment contains the words UNDER DEVELOPMENT (in capital letters), that fact is reported and no checking takes place for that class.

Backward Compatibility Message Sends

The methods are checked to see whether they send messages that exist (only) in a backward compatibility protocol.

Indefinite Backward Compatibility Message Sends

Similar to the preceding option, but the checker only pays attention to the ambiguous case, when a message send exists in both a backward compatibility category and another category. In this situation, static analysis cannot determine whether the message send is inappropriate, so it is reported as a candidate for your further investigation.

Backward Compatibility Class References

The methods are checked to see whether they refer to a class that is in a class category that contains the string 'backward compat' (without case sensitivity).

Estimating Memory Requirements

To receive an estimate of the memory requirements of the target classes, select the **Space** switch in the upper-right portion of the Class Reports window. Three new switches will appear. Each button provides a different perspective on the estimated memory requirements, as follows:

- **Class Size**—For each target class, the report shows the estimated number of bytes required for the class definition, variables, methods, and class organization.
- **Method Size**—For each method in a target class, the following measurements are reported:
 - **Code Bytes**—the memory occupied by the method's byte code, the portable compiled form of the method that is used to create native machine code.
 - **Literals**—the number of literal pointers created by the compiler to refer to such things as message selectors, arrays, strings, and floats. Each such literal pointer contributes 4 bytes to the total.
 - **Literal Bytes**—the number of bytes required by literal objects other than Symbols.
 - **Full Blocks**—the number of full blocks in each method. Full blocks are blocks that contain out-of-scope references to temps, or nonlocal (^) returns. Full blocks are nonoptimal because they are slower and use more dynamic memory. This is only of concern in methods that are used frequently.
 - **Total**—the estimated total number of bytes needed by each method, including overhead (20 bytes) not reported in the other columns. A total byte count for all methods is also displayed.

- **Instance Size**—For each target class, the following measurements are reported:
 - **Count**—the number of instances that exist.
 - **TotBytes**—the memory, in bytes, occupied by all instances.
 - **AveByte**—the average number of bytes for each instance.

A summary line reports the same measurements for all target classes.

These reports are intended to help you optimize memory usage by identifying places in your code where memory usage is disproportionate to the functional contribution of the code.

Documenting Your Code

To create a listing of some or all of the elements that make up the code in the target classes, select the **Manual** switch in the upper left portion of the Class Reports window. Two new switches will appear, labeled **Report** and **Print**. When the **Report** switch is selected, the documentation is displayed in a separate window. When **Print** is selected, the output is sent to a printer instead.

The following check-box options are provided for defining the code components to be included in the listing. The options are hierarchic and interconnected, as follows:

- **class definition**
- **class comment**
- **include metaclass**—include the metaclass definition.
- **protocol names**—instance protocol names are reported; class protocol names are included when the **include metaclass** check-box is selected.
- **include private protocols**—include any protocol beginning with the string “private.” Private protocols are made separable in this way because only public protocol is relevant for certain types of manuals.
- **methods**—list method selectors, including metaclass and private methods if those check-boxes are selected.
 - **method comments only**
 - **method bodies**—including method comments.

Various text emphases are used for the different components of documentation. For example, `#italic` is used for the class comment. To change one of these emphases, modify and recompile the appropriate method in the emphases protocol on the instance side of the `ManualWriter` class.

Index

Symbols

<Operate> button [xi](#)
<Select> button [xi](#)
<Window> button [xi](#)

A

adding

class definition [1-6](#)
method definition [1-6](#)

B

Benchmarks

Arithmetic mean [10-6](#)
BenchDecompilerTestClass [10-8](#)
Benchmark class [10-7](#)
BenchmarkTable class [10-8](#)
clear selections command [10-3](#)
creating a subclass [10-7](#)
Harmonic mean [10-6](#)
Maximum [10-6](#)
Median [10-6](#)
Minimum [10-6](#)
opening example [10-2](#)
Raw Measurements [10-3](#)
report components [10-3](#)
reset to default command [10-3](#)
run button [10-2](#), [10-7](#)
select all command [10-3](#)
suite statistics [10-5](#)
SystemBenchmark class [10-1](#), [10-7](#)
types of statistics [10-6](#)
window components [10-2](#)

buttons

mouse [xi](#)

C

Change List

See also changes
browsing [4-3](#)
condensing [5-9](#)

change list [5-1](#)

Change Set

See also changes
browsing changes [5-7](#)
clearing [4-6](#)

updating [4-5](#)

changes

See also Change List, Change Set
browsing [5-7](#)
managing [5-1–5-9](#)

class

creating [1-6](#)

class button [1-5](#)

Class Reports

accept command [11-3](#)
add all command [11-3](#)
Browse switch [11-3](#)
Check comment [11-5](#)
Class List view [11-3](#)
Class Patterns view [11-2](#)
Class Size [11-7](#)
clear all command [11-3](#)
Correctness reports [11-3](#)
finding coding errors [11-3](#)
Inst vars not referenced [11-5](#)
Instance Size [11-8](#)
Manual switch [11-8](#)
memory usage reports [11-7](#)
Messages implemented but not sent
[11-4](#)
Messages sent but not implemented
[11-4](#)
Method consistency [11-4](#)
Method Size [11-7](#)
opening [11-2](#)
Report switch [11-3](#)
Space switch [11-7](#)
SubclassResponsibilities not
implemented [11-5](#)
text emphases [11-9](#)
Undeclared references [11-5](#)
Wildcard patterns [11-3](#)
window components [11-2](#)

conventions

typographic [x](#)

crash recovery [5-7](#)

D

decompiled code [1-6](#)

E

editing
source code 1-6

F

filtering
change list 5-10

fonts x

G

getting help xi

I

instance button 1-5

M

method
creating 1-6
mouse buttons xi
 <Operate> button xi
 <Select> button xi
 <Window> button xi

N

named change sets 4-1
notational conventions x

O

overrides
packages 3-3
parcels 3-3

P

packages
overrides 3-3

parcels
overrides 3-3

Profilers

apply cutoff button 9-7
contract fully command 9-7
cutoff percentage 9-7
expand command 9-8
expand fully command 9-8
profile descriptors 9-5
profile window 9-5
spawn command 9-9
threshold percentage 9-7
totals switch 9-6
tree list expansion 9-7
tree switch 9-6
window components 9-2

project

managing 5-1–5-9
summarizing changes 4-3

R

recover, from crash 5-7

S

save source code 1-6
shared variables button 1-5
source code
 editing 1-6
 missing 1-6
 saving 1-6
special symbols x
symbols used in documentation x
System Browser 1-1

T

technical support xi
typographic conventions x

V

version control 5-1–5-9