

SilverMark's SMALLTALK Test Mentor™

©1996-2002 SilverMark, Incorporated. All rights reserved

Sixth Edition (May, 2002)

This edition applies to of Smalltalk Test Mentor products released in May of 2002., and to all subsequent releases and modifications until otherwise indicated in new editions. Please make sure you are using the correct edition for the level of the product.

This manual, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. The content of this manual is furnished for informational use only and is subject to change without notice and should not be construed as a commitment by SilverMark, Incorporated.

For defense agencies: Restricted Rights Legend. Use, reproduction or disclosure is subject to restrictions set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at 252.227-7013.

For civilian agencies: Restricted Rights Legend. Use, reproduction or disclosure is subject to restrictions set forth in subparagraphs (a) through (d) of the commercial Computer Software Restricted Rights clause at 52.227-19 and the limitations set forth in the SilverMark standard commercial agreement for this software. Unpublished rights reserved under the copyright laws of the United States.

If you have comments about this manual, please direct them to:

SilverMark Inc.
5511 Capital Center Drive
Suite 510
Raleigh, NC 27606
Attention: Publications

E-mail: publications@silvermark.com

Printed in the U.S.A

Table of Contents

RELEASE NOTES	8
<i>Trademarks</i>	8
PREREQUISITES	8
SUPPORT	8
WHAT IS THE SILVERMARK'S SMALLTALK TEST MENTOR?	9
Key Concepts	9
ABOUT THIS BOOK	10
<i>Who this Book is for</i>	10
<i>Product Updates Not In This Book</i>	10
<i>Vendor Variations</i>	10
<i>Organization of this book</i>	10
<i>How to read this book</i>	10
<i>Conventions used in this book</i>	11
<i>Required Skills</i>	11
GETTING STARTED	12
THE BASICS	12
<i>Installation</i>	12
For platforms using ENVY/developer	12
For platforms using VisualWorks parcels	12
VisualAge Smalltalk	13
VisualWorks	14
SMALLTALK VENDOR VIEW DIFFERENCES	14
<i>Test Browser</i>	15
VisualAge	15
VisualWorks	16
<i>Test Editor</i>	17
VisualAge	17
VisualWorks	18
<i>Test Results Browser</i>	19
VisualAge	19
VisualWorks	20
<i>Test Differences browser</i>	21
VisualAge	21
VisualWorks	22
<i>Method Coverage Analysis view</i>	23
VisualAge	23
VisualWorks	24
Creating and Running Your First Test Case	24
Creating and Running Your First Test Case	25
<i>Using the Test Editor</i>	25
<i>Using the Quick Runner</i>	33
<i>Viewing the Results of Running Your Test</i>	35
<i>Conclusion</i>	38
SILVERMARK'S TEST MENTOR USER'S REFERENCE	40
TEST CASE STRUCTURE	40
<i>Overview</i>	40
<i>Test representation</i>	40
<i>The Test Step</i>	40
Test Step Implementation	41
Test Step Properties	41
Class Method Step	42
Instance Method Step	42
Suite Step	42
Scenario Step	42

Workspace Step.....	43
Manual Intervention Step.....	43
Collection Step.....	43
Conditional Collection Step.....	43
File Iteration Step.....	43
Script Step.....	44
User Interface Step.....	45
User Interface Verification Step.....	45
User Interface Recording Step.....	45
Unspecified Step.....	45
<i>The Test Scenario</i>	45
<i>The Test Suite</i>	46
TEST MENTOR VIEWS.....	47
<i>The Test Editor</i>	47
VisualAge:.....	47
VisualWorks:.....	47
VisualWorks:.....	48
The List View.....	49
The Details View.....	50
Test Editor Tool Bar.....	59
Test Editor Menus.....	60
Test Editor Operations.....	62
<i>The Test Browser</i>	68
<i>The Quick Runner</i>	68
<i>The Test Runner</i>	70
Test Runner List View.....	70
Test Runner Execution View.....	70
Test Runner Tool Bar.....	73
Test Runner Menus.....	73
Test Runner Operations.....	74
<i>The Test Results Browser</i>	77
The List View.....	77
The Details View.....	78
Test Results Tool Bar.....	80
Test Results Menus.....	80
<i>The Test Results Differences Browser</i>	84
Test Results Differences Browser List View.....	84
Test Results Differences Browser Details View.....	85
Test Results Differences Browser Tool Bar.....	86
Test Results Differences Browser Menus.....	86
<i>Method Coverage Browser</i>	88
Method Coverage Browser List View.....	89
Method Coverage Browser Details View.....	91
Method Coverage Browser Tool Bar.....	91
Test Results Differences Browser Menus.....	91
<i>The Preferences View</i>	93
Time values.....	93
Prompters.....	94
Recording.....	95
Miscellaneous.....	96
RECORDING AND PLAYING BACK PROMPTERS AND SUSPENDED VIEWS.....	98
(VISUALAGE SMALLTALK ONLY).....	98
<i>System prompters</i>	98
<i>Suspended views</i>	98
SILVERMARK'S SMALLTALK TEST MENTOR USER'S GUIDE.....	100
CREATING AND EDITING TESTS.....	100
<i>How to open the Test Editor</i>	100
<i>How to close the Test Editor</i>	100
<i>How to create a new suite</i>	101

<i>How to create a new scenario</i>	102
<i>How to add a step or steps to a scenario</i>	102
<i>How to change the ordering of steps within the Test Editor</i>	103
<i>How to change step types</i>	106
VisualAge:	106
VisualWorks:	106
<i>How to change the number of times a step is executed</i>	107
<i>How to temporarily disable execution of a step</i>	107
<i>How to delete suites, scenarios, or steps</i>	108
<i>How to record user interface interactions</i>	108
How to capture user interface interactions during execution.....	108
How to record directly into a User Interface Recording step	113
<i>How to specify whether UI playback code is generated as widget or VisualAge part interactions (VisualAge only)</i>	115
<i>How to verify widget state</i>	115
<i>How to view which widgets are in a window</i>	120
<i>How to apply a test data file to a set of recorded user interface interactions</i>	122
RUNNING TESTS	123
<i>How to view existing tests</i>	123
<i>How to open the Test Browser</i>	123
<i>How to run a test</i>	123
How to use the Quick Runner	123
How to use the Test Runner	124
<i>How to stop a test from executing</i>	125
<i>How to run steps in a particular order (VisualAge only)</i>	125
<i>How to pause and resume the execution of a test</i>	127
What you can do when a test is paused	127
How to pause execution for any step.....	129
How to pause execution for specific steps.....	129
<i>How to terminate execution of a test while it is running</i>	130
<i>How to set delays and wait times</i>	130
RESULTS ANALYSIS	132
<i>How to open a Test Results Browser</i>	132
<i>How to calculate step statistics</i>	132
<i>How to compare steps for differences</i>	134
<i>How to analyze method coverage metrics</i>	138
<i>How to store results into the results archive (ENVY only)</i>	140
<i>How to store results into a file</i>	140
<i>How to load results from the results archive</i>	141
<i>How to view results from an external tool</i>	141
<i>How to load test results from a file</i>	145
TESTING STRATEGIES	146
<i>How to design test cases</i>	146
Test Case Structure	146
<i>How to reuse tests</i>	149
Reuse through reference.....	149
Reuse through inheritance.....	150
<i>How to use invariants</i>	150
<i>How to keep track of objects under test</i>	151
<i>How to design for step failures</i>	153
How to allow a test to proceed after a failed step.....	154
How to abort a test after a failed step	155
How to 'clean up' after a failed step	156
<i>How to test asynchronous systems</i>	157
<i>How to make the best use of metrics</i>	158
Execution time	158
Code coverage.....	158
Executed/Passed/Failed steps.....	159

Combining metrics	159
MISCELLANEOUS	160
<i>How to Search</i>	160
<i>How to adjust the proportional size of the upper and lower views within a window (VisualAge only)</i>	161
THE SMALLTALK TEST MENTOR PROGRAMMER'S GUIDE.....	162
TEST CASE SPECIFICATION	162
<i>Specifying a Suite</i>	162
<i>Specifying a Scenario</i>	163
The scenario spec	163
Scenario Steps.....	164
Adding Base Steps	166
Adding User Interface Playback Steps	169
<i>Steps</i>	169
Class method step - StmClassMethodStep	169
Instance method step - StmInstanceMethodStep.....	169
Suite Step - StmSuiteStep	170
Scenario Step - StmScenarioStep.....	170
Workspace step - StmWorkspaceStep.....	170
Manual intervention step - StmManualStep	171
Collection step - StmCollectionStep	171
User interface recording step - StmUIRecordedStep.....	171
File Iteration step - StmCollectionOnTextFileStep	171
Script step - StmBlockStep	172
User interface step - StmRecordedStep.....	172
User interface verification step - StmVerificationStep.....	172
<i>Changing the execution time behavior of a step</i>	172
<i>Suite Operations</i>	173
#performTest.....	173
#performTestStoreResultsInto: <fileName>	173
#report.....	173
#scenarioNamed: <aString>.....	174
#scenarioNamed: <aString> ifAbsent: <failBlock>	174
#scenarioNames	174
#scenarios.....	175
#scenarioSpecNamed: <aString>	175
#scenarioSpecNamed: <aString> ifAbsent: <failBlock>	175
#scenarioSpecs	175
#spec	175
<i>Test Case Structure</i>	175
STEP EXECUTION	178
EXECUTION APIS.....	179
<i>Test Variables</i>	179
#varNamed: <name> put: <object>.....	179
#var: <object> named: <name>.....	179
#varNamed: <name>.....	179
<i>Execution Control</i>	179
#performTest: <aBlock>	179
#performTest: <aBlock> name: <nameString> description: <descriptionString>.....	179
#failStep.....	180
#verify: <aBlock>	180
#verify: <aBlock> name: <nameString> description: <descriptionString>.....	180
<i>State</i>	180
#currentStep	180
#currentIteration.....	180
#currentResults.....	180
#currentStackDump.....	180
#previousResults	180
#previousValue	180
#pass: <aBoolean>.....	180

Iteration control.....	181
#repeatIteration	181
#repeatIterationIf: <conditionBlock>	181
<i>Message logging</i>	<i>181</i>
#logMsg: <aString>	181
#log: <aString> toFileNamed: <fileNameString>	181
<i>Waiting on condition</i>	<i>181</i>
#checkCondition: conditionBlock	181
interval: intervalInMilliseconds	181
timeout: timeoutInMilliseconds	181
TEST RESULTS	182
<i>Persistent Test Results</i>	<i>185</i>
Adding New Types of Persistent Repository	185
#materializeResults: uniqueIdentifier	186
#storeResults: anStmTestResults	186
RECORD AND PLAYBACK LIMITATIONS AND NOTES	187
INDEX	188

Release Notes

You should read the README.TXT file that is included with distribution CD or download for the latest information on problems and limitations.

Trademarks

The following terms are trademarks of SilverMark, Inc. in the United States or other countries or both:

Test Mentor *Visual Verification Wizard*

The following terms are trademarks of other companies:

VisualAge, ENVY/developer *IBM Corporation*

VisualWorks *Cincom, Inc.*

Excel *Microsoft Corporation*

Prerequisites

For operation on VisualAge Smalltalk, the Smalltalk Test Mentor must be installed on version 3.0a or later.

For operation on VisualWorks™, the following configurations are supported:

VisualWorks version	Source control or code repository supported
2.51	<i>ENVY/developer</i>
3.0	<i>ENVY/developer</i>
5i.2	<i>ENVY/developer</i>
5i.4	Parcels or STORE
7.0	Parcels or STORE

Support

For questions, problems or general assistance regarding the Smalltalk Test Mentor, send e-mail to

support@silvermark.com

What is the SilverMark's Smalltalk Test Mentor?

SilverMark's Smalltalk Test Mentor is a framework that enables you to efficiently create, manage, execute and analyze the results of test cases for systems developed in Smalltalk. The Smalltalk Test Mentor records user interface interactions at the widget level, for replay during testing, enabling you to rapidly create user interface tests. The Smalltalk Test Mentor also provides facilities for testing your system model (domain object) classes and encourages you to test early and often throughout the development process.

Unlike many black-box user interface automation tools, the Smalltalk Test Mentor operates within the Smalltalk image. This gives it the power to operate on any class in the Smalltalk image, whether it is a user interface class or a model class. Because the Smalltalk Test Mentor operates within the image, it has full access to the state of any instances of user interface classes - even extended widgets.

The Smalltalk Test Mentor is fully integrated with ENVY®/developer. When you create test cases, they are saved as classes and methods. Any changes you make are automatically managed by ENVY®/developer. Test execution results may be stored within the ENVY®/developer database. Test execution results are always stored under the control of the test case class whose execution created them. You can always be certain about which version of a test class was responsible for the creation of a particular set of stored test results.

In summary, you can use the Smalltalk Test Mentor to help you raise the quality level of the Smalltalk systems you produce while reducing the cost of quality assurance. The Smalltalk Test Mentor helps you do this by introducing structure and automation to your testing process.

Key Concepts

Product variations

SilverMark's Smalltalk Test Mentor comes in two variations: **Smalltalk Test Mentor - runner**, which provides functions related to executing existing tests and analyzing the results of their execution, and the **Smalltalk Test Mentor - developer** which provides all the above functions plus all functions related to developing tests, including user interface interaction recording.

Vendor Variations

SilverMark's Test Mentor is supported for IBM's VisualAge Smalltalk and ObjectShare's VisualWorks. In general, Test Mentor operates equivalently in both environments. Any differences will be noted in the appropriate sections in this manual.

Testing units

The Smalltalk Test Mentor views tests in terms of *suites*, *scenarios* and *steps*. A step is the smallest atomic unit of test execution. You have complete control over the granularity of the steps you define. Steps are grouped by scenario. A scenario is a particular usage or *use case* of the system or a part of the system under test. Scenarios are grouped by suite. A suite is a set of scenarios specific to a particular subset of the system under test.

Interface

The Smalltalk Test Mentor provides the following views for performing tasks:

Task	View
Creating tests	Test Editor*
Viewing tests	Test Browser
Executing tests	Test Runner, Quick Runner
Viewing execution results	Results Browser

* Not available in the Smalltalk Test Mentor - runner product.

Viewing differences between results	Results Differences Browser
Viewing method coverage metrics	Method Coverage Browser
Adding user interface verification	Visual Verification Wizard**

About this Book

The purpose of this book is to introduce you to basic through advanced usage of the Smalltalk Test Mentor. It will provide you with the information you need to start creating and managing test cases for your Smalltalk system.

Who this Book is for

This book is for anyone preparing to use the Smalltalk Test Mentor to manage and automate their testing of Smalltalk systems.

Product Updates Not In This Book

SilverMark is constantly improving its products. In some cases, the product itself may include features and improvements that are not yet included in this manual. You should always check the [readme.txt](#) file that is included with the product to note any differences.

Vendor Variations

As stated above, variations between vendor implementations (VisualAge and VisualWorks) are minimal. Because the Test Mentor version supporting VisualAge was produced first, most screen captures are from a VisualAge development image. VisualWorks screen captures are presented when there are notable differences.

Organization of this book

This book is organized into four sections:

Section Title	Purpose
Getting Started	Read this chapter first to get a brief introduction to working with the product and what it can do for you, as well as how to install it.
User's Reference	Read this chapter to find out what the different parts of the product are, and to some degree, how to use those parts.
User's Guide	Read this chapter to find out how to use the product to perform tasks. Each section in this chapter is presented in terms of how to perform a particular task.
Programmer's Guide	Read this section to understand the programming APIs used for test automation.

How to read this book

If this is your first exposure to the Smalltalk Test Mentor, you should read the *Creating and Running Your First Test Case* section of this chapter. This will briefly introduce you to hands-on operation of the product.

** Not available in the Smalltalk Test Mentor - *runner* product.

You should then lightly read the chapter titled *Smalltalk Test Mentor User's Reference* to get a general impression of what the components of the product are, then move on to the chapter titled *Smalltalk Test Mentor User's Guide* for step-by-step instructions on how to perform specific tasks. Refer to the chapter titled *Smalltalk Test Mentor Programmer's Guide* for details on creating test case scripts and other technical issues.

Conventions used in this book

This book uses the following text highlighting conventions:

Highlight Style	Used for
Boldface	Used to indicate special emphasis, or text that you can enter.
<i>Italics</i>	Used to introduce new terms the first time they are used.
<u>Underscore</u>	Used to show a label, title or the name of a widget within a window.
Example font	Used to show executable Smalltalk code.

Required Skills

Basic navigation skills - you only need to have basic navigation skills. For example, you would have to know how to find the Smalltalk Tools menu in the Smalltalk System Transcript in order to open the views required to perform your tasks.

If you plan to create test cases, **you do not need to understand how to program in Smalltalk**, however a little knowledge is helpful if you plan to create intelligent scripts.

A very brief introduction is also helpful in order to understand how to load and save test case configurations.

Getting Started

The Basics

Installation

Follow the specific directions in the readme.txt file that is provided with the product CD or download. The general concept is to:

For platforms using ENVY/developer

1. Extract the product files from the compressed installation format and copy them to a directory accessible to the file system visible to the code repository.
2. Import the code contained in the product files into the code repository
3. Load the product code from the code repository into a development image

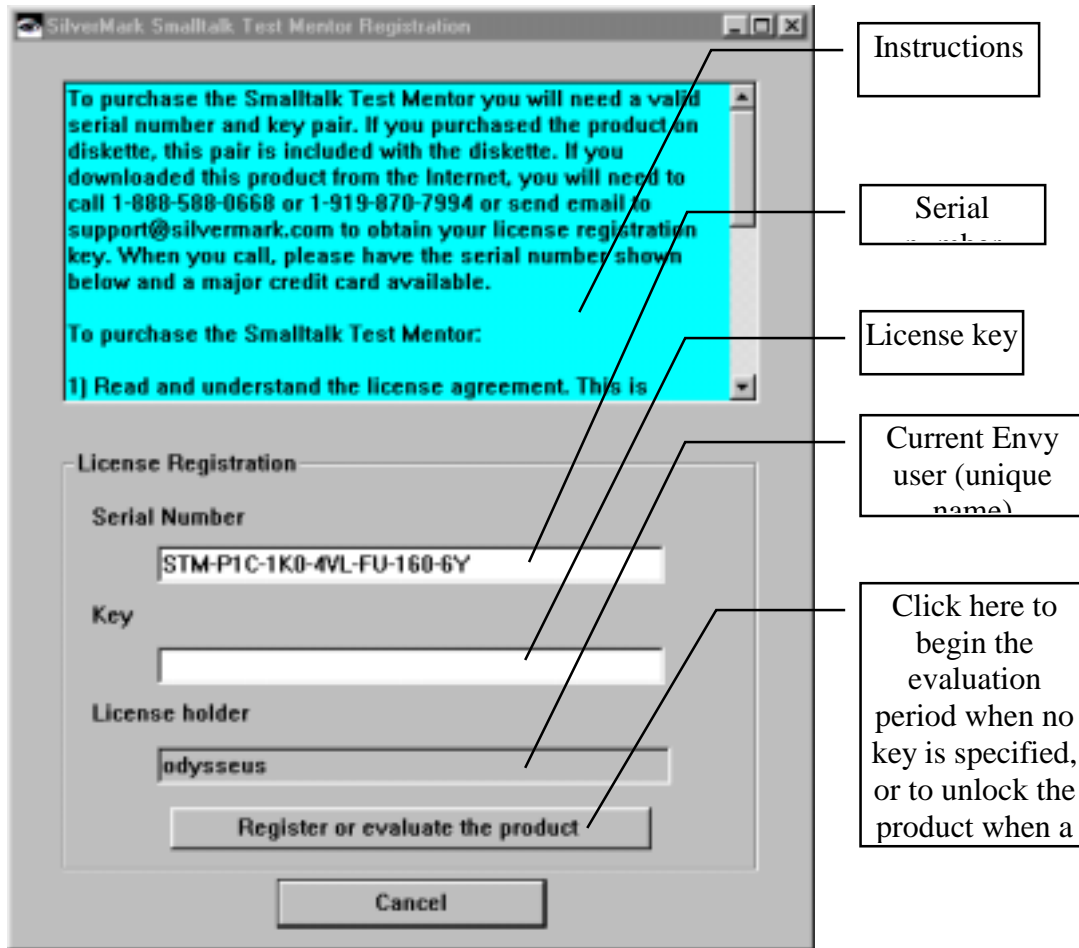
For platforms using VisualWorks parcels

1. Extract the distributed product parcel files from the compressed installation format and copy them to the \$(VISUALWORKS)\parcels directory.
2. Load the TestMentor.pcl parcel. This will pull in any required parcels.

Registering or evaluating the product

When you load the Smalltalk Test Mentor into your image and open one of the views you are presented with a product registration dialog.

VisualAge Smalltalk



Note: If the above dialog is shown when a view opens, the view will immediately close.

If your license for the product came with a key and a serial number, enter your key in the field marked Key and then replace the serial number in the Serial Number field with the one that came with your key.

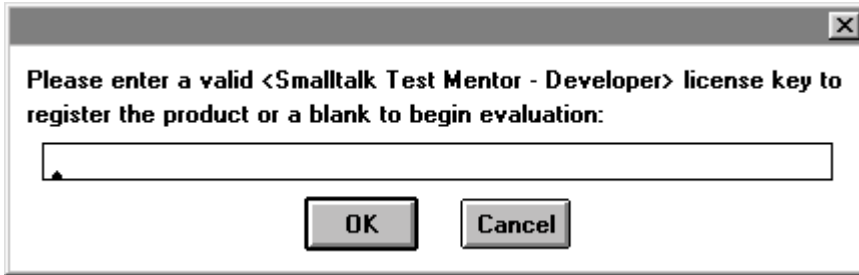
If you are simply evaluating the product, you may press the Register or evaluate the product button with no key specified to start your thirty-day evaluation period. During the evaluation period you will be reminded each day with a message in the system transcript of how many days you have remaining:

odysseus has 29 days remaining on the Smalltalk Test Mentor evaluation.

Please contact SilverMark, Inc. at info@silvermark.com or www.silvermark.com or (888) 588-0668 or (919) 870-7994 if you would like to register your copy.

At any time during the product evaluation period, you may choose to register (unlock) the product by opening the above dialog and following the instructions. This process will be discussed in the *Smalltalk Test Mentor User's Guide*.

VisualWorks



Note: If the above dialog is shown when a view opens, the view will immediately close after you dismiss this dialog.

The VisualWorks licensing scheme is slightly simpler than that provided with VisualAge. To temporarily enable the product for evaluation, simply press OK in the registration dialog with the key entry field empty. Once you have purchased the product, enter the key in the key entry field and press OK.

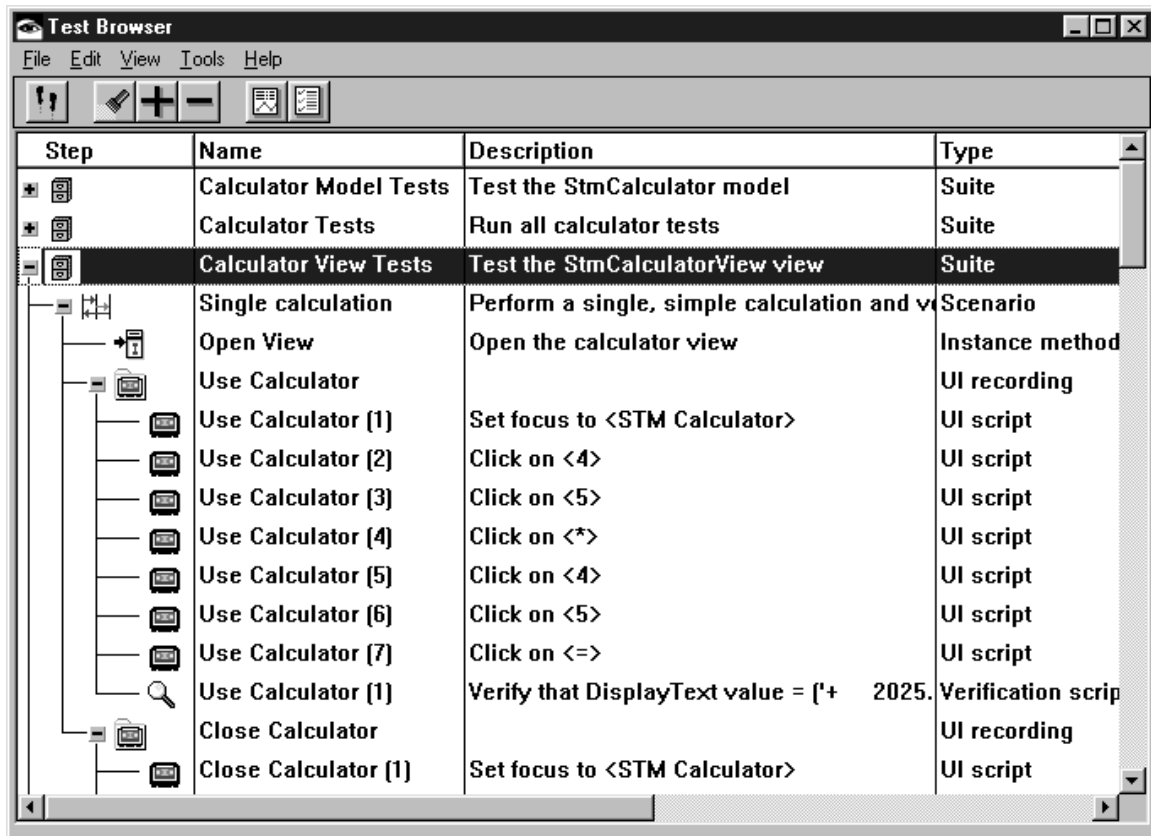
Smalltalk Vendor View Differences

In general, SilverMark's Test Mentor operates nearly identically in VisualWorks and VisualAge. Due to differences in base widgets available with the products, there are, however, some minor differences in some views. Some differences are mainly cosmetic. Other differences require different interactions with the views.

The chief difference is in the way the properties of test elements (suites, scenarios and steps) are presented. In VisualAge, a columnar table-tree is used in many views, where specific properties occupy individual columns. In VisualWorks, a separate region is used to display the properties of the selected step. Neither arrangement is particularly advantageous with respect to the other with each having its pros and cons. These differences should have little to no impact on your productivity. Any operational differences will be noted in the appropriate sections of this manual.

Test Browser

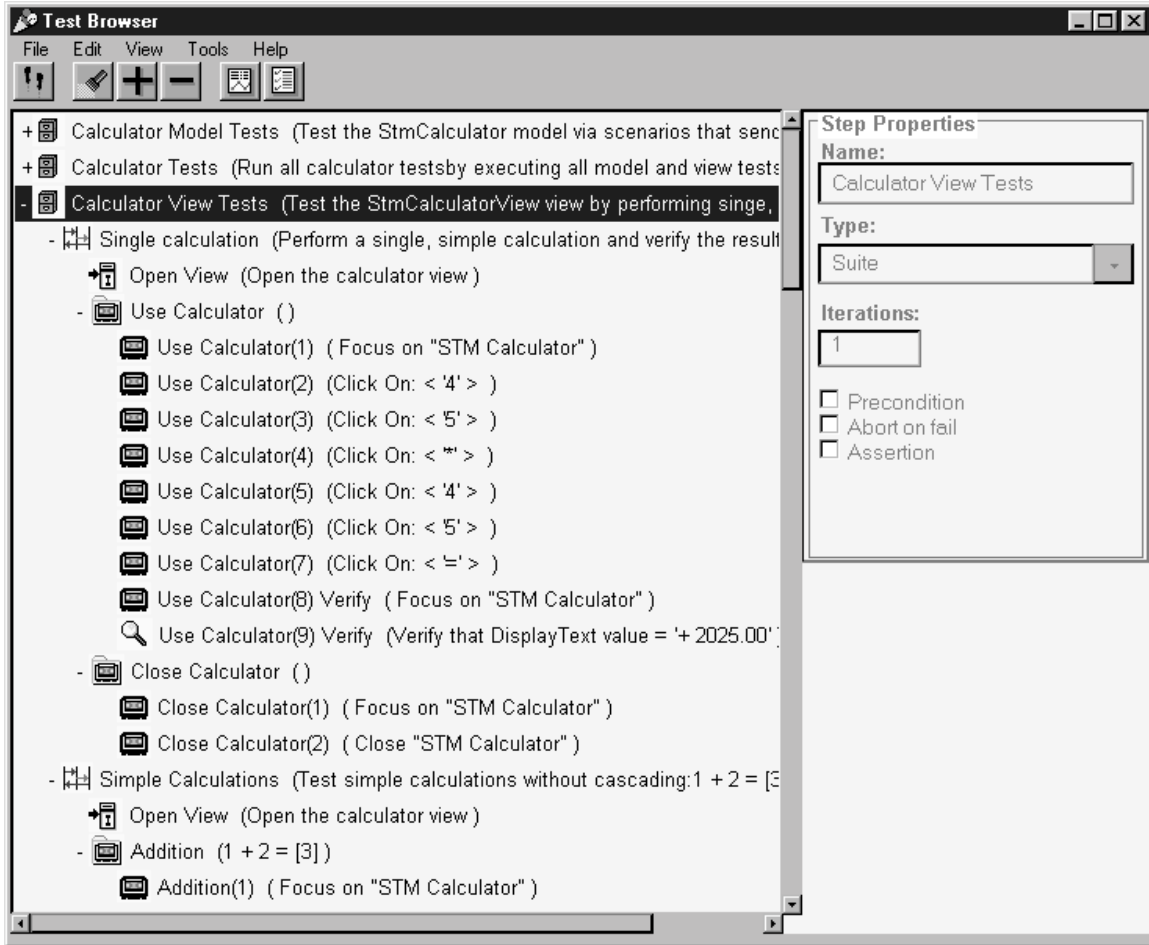
VisualAge



The screenshot shows the Test Browser application window with a menu bar (File, Edit, View, Tools, Help) and a toolbar. The main area displays a tree view of test suites and steps. The 'Calculator View Tests' suite is expanded, showing a list of steps with their names, descriptions, and types.

Step	Name	Description	Type
+	Calculator Model Tests	Test the StmCalculator model	Suite
+	Calculator Tests	Run all calculator tests	Suite
-	Calculator View Tests	Test the StmCalculatorView view	Suite
	Single calculation	Perform a single, simple calculation and v	Scenario
	Open View	Open the calculator view	Instance method
	Use Calculator		UI recording
	Use Calculator (1)	Set focus to <STM Calculator>	UI script
	Use Calculator (2)	Click on <4>	UI script
	Use Calculator (3)	Click on <5>	UI script
	Use Calculator (4)	Click on <*>	UI script
	Use Calculator (5)	Click on <4>	UI script
	Use Calculator (6)	Click on <5>	UI script
	Use Calculator (7)	Click on <=>	UI script
	Use Calculator (1)	Verify that DisplayText value = ['+ 2025.	Verification scrip
	Close Calculator		UI recording
	Close Calculator (1)	Set focus to <STM Calculator>	UI script

VisualWorks



Test Editor

VisualAge

The screenshot shows the Test Editor application window. The main area is a table listing test steps. The 'Calculator View Tests' suite is expanded, showing several steps including 'Single calculation', 'Open View', and three 'Use Calculator' steps. Below the table, there are two panels: 'Description of Calculator View Tests' and 'Calculator View Tests's implementation details:'. The description panel contains a text area with the text: 'Test the StmCalculatorView view by performing single, simple and complex calculation sequence scenarios.' The implementation details panel has sections for 'Invariant Selectors' (with a dropdown menu and 'New Method' and 'Browse...' buttons) and 'Applications Covered' (with a text area containing 'SmxCalculatorApp' and a 'Change...' button).

Step	Name	Description	Type
+	Calculator Model Tests	Test the StmCalculator model	Suite
+	Calculator Tests	Run all calculator tests	Suite
-	Calculator View Tests	Test the StmCalculatorView view	Suite
-	Single calculation	Perform a single, simple calculation and v	Scenario
-	Open View	Open the calculator view	Instance method
-	Use Calculator		UI recording
-	Use Calculator (1)	Set focus to <STM Calculator>	UI script
-	Use Calculator (2)	Click on <4>	UI script
-	Use Calculator (3)	Click on <5>	UI script

Description of Calculator View Tests

Test the StmCalculatorView view by performing single, simple and complex calculation sequence scenarios.

Calculator View Tests classification:

SMI - Calculator Example

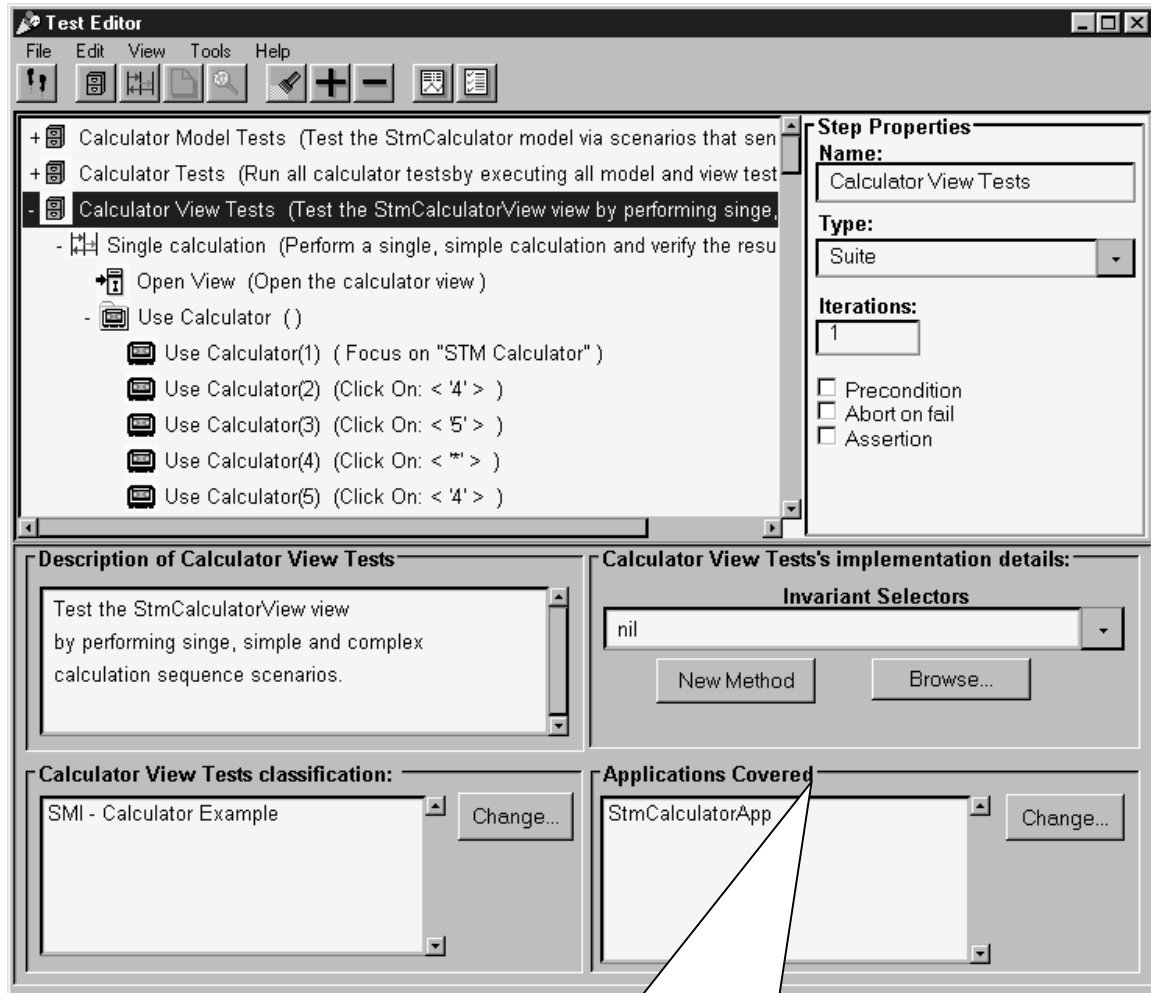
Calculator View Tests's implementation details:

Invariant Selectors

Applications Covered

SmxCalculatorApp

VisualWorks



Note: on platforms that use ENVY/developer, you will see **Applications Covered**. Otherwise you will see **NameSpaces**

Test Results Browser

VisualAge

Step	Pass	Name	Description	Execution time	Type
+	✓	Calculator View Tests	Test the StmCalculatorView view	2,653	Suite
-	✓	Calculator View Tests	Test the StmCalculatorView view	2,900	Suite
+	✓	Single calculation	Perform a single, simple calculation and	1,162	Scenario
+	✓	Simple Calculations	Test simple calculations without cascading	1,110	Scenario
+	✓	Complex Calculations	Test calculations with cascading:	628	Scenario
+	✓	Open View	Open the StmCalculator view	243	Script
+	✓	Single Equals	100 / 5 - 5 * 3 + 5 = [50]	145	UI recording

Name	Run	Pass	Fail	Time	Covered (%)	Testable (%)	Methods (#)	Testable (#)
Calculator View Test	81	81	0	2,653	58.97	82.14	39	28
Calculator View Test	81	81	0	2,900	53.85	75.00	39	28

Metric	Max	Min	Range	Mean	Median	Std Dev
Steps run	81	81	0	81.00	81	0.00
Steps passed	81	81	0	81.00	81	0.00
Steps failed	0	0	0	0.00	0	0.00
Execution time	2,900	2,653	247	2,776.50	2,653	174.66
% Testable covered	82	75	7	78.57	82	5.05
% covered	59	54	5	56.41	59	3.63

VisualWorks

The screenshot shows the 'Test Results' window in VisualWorks. The window title is 'Test Results' and it has a menu bar with 'File', 'Edit', 'View', 'Tools', 'Archive', and 'Help'. Below the menu bar are several icons. The main area displays a tree view of test results:

- [Pass] Calculator View Tests (Test the StmCalculatorView view by performing single, simple and complex calculations)
- [Pass] Single calculation (Perform a single, simple calculation and verify the results: $45 * 45 = [2025]$)
- [Pass] Open View (Open the calculator view)
- [Pass] Use Calculator ()
 - [Pass] Use Calculator(1) (Focus on "STM Calculator")
 - [Pass] Use Calculator(2) (Click On: < '4' >)
 - [Pass] Use Calculator(3) (Click On: < '5' >)
 - [Pass] Use Calculator(4) (Click On: < '*' >)
 - [Pass] Use Calculator(5) (Click On: < '4' >)
 - [Pass] Use Calculator(6) (Click On: < '5' >)

Below the tree view is a table with the following columns: Name, Run, Pass, Fail, Time, Covered(%), Testable(%), Methods(#), Testable(#). The table contains one row:

Name	Run	Pass	Fail	Time	Covered(%)	Testable(%)	Methods(#)	Testable(#)
Calculator View Tests	90	90	0	3519	84.4444	84.4444	45	45

At the bottom of the window, there is a text area containing the string '\true'.

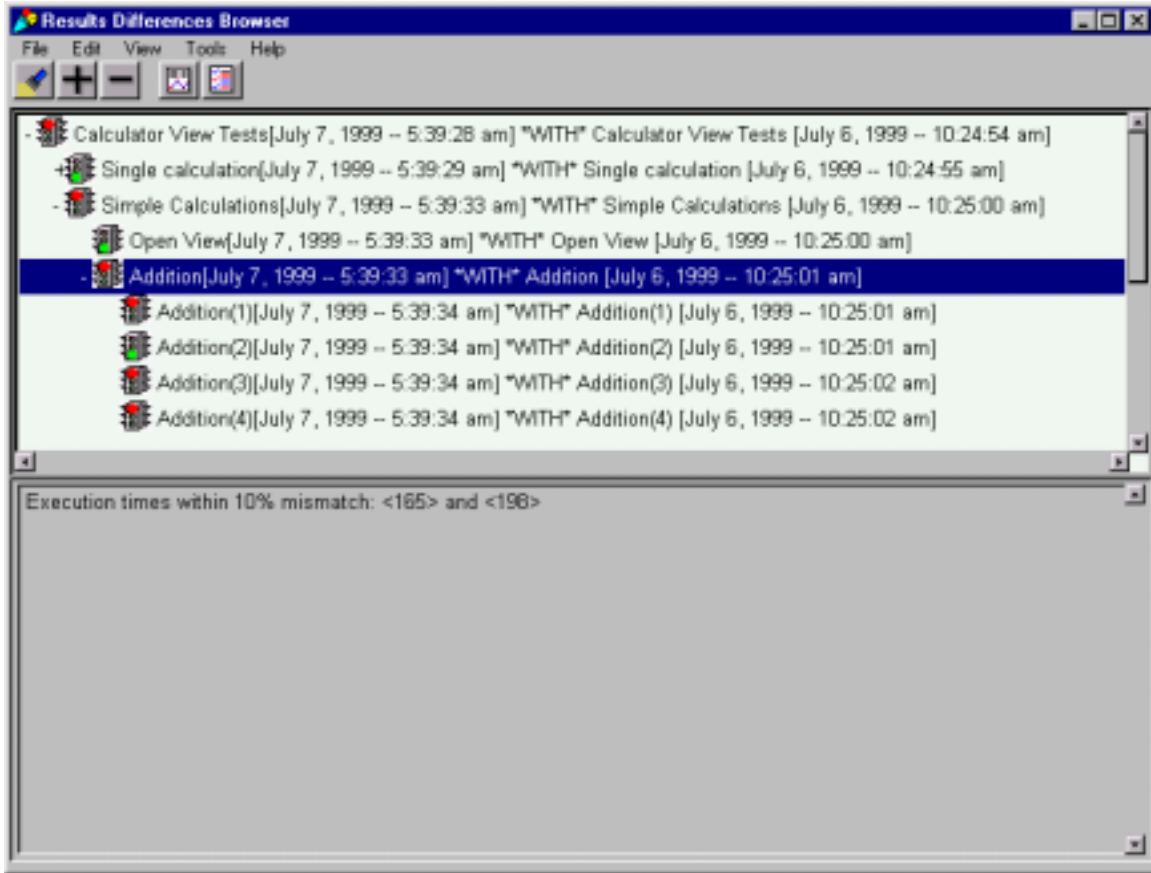
Test Differences browser

VisualAge

Match	Name	Time Stamp	Name	Time Stamp
	Calculator View Tests	7/5/99 - 11:32:28 P	Calculator View Tests	7/7/99 - 8:27:51 AM
	Single calculation	7/5/99 - 11:32:28 P	Single calculation	7/7/99 - 8:27:52 AM
	Simple Calculations	7/5/99 - 11:32:33 P	Simple Calculations	7/7/99 - 8:27:56 AM
	Open View	7/5/99 - 11:32:33 P	Open View	7/7/99 - 8:27:56 AM
	Addition	7/5/99 - 11:32:34 P	Addition	7/7/99 - 8:27:57 AM
	Subtraction	7/5/99 - 11:32:36 P	Subtraction	7/7/99 - 8:27:59 AM
	Multiplication	7/5/99 - 11:32:38 P	Multiplication	7/7/99 - 8:28:01 AM

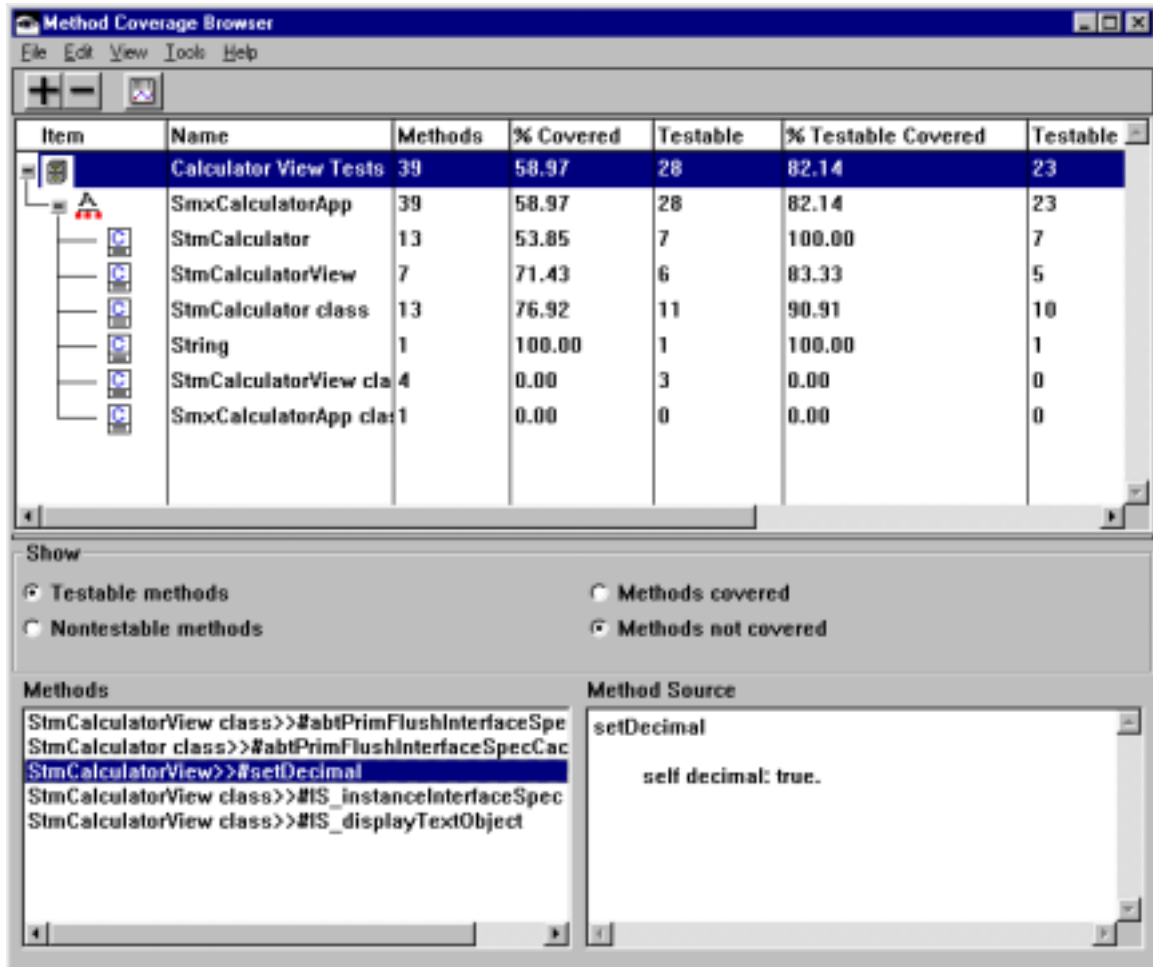
Execution times within 10% mismatch: <236> and <275>

VisualWorks



Method Coverage Analysis view

VisualAge



Item	Name	Methods	% Covered	Testable	% Testable Covered	Testable
	Calculator View Tests	39	58.97	28	82.14	23
	SmxCalculatorApp	39	58.97	28	82.14	23
	StmCalculator	13	53.85	7	100.00	7
	StmCalculatorView	7	71.43	6	83.33	5
	StmCalculator class	13	76.92	11	90.91	10
	String	1	100.00	1	100.00	1
	StmCalculatorView cla	4	0.00	3	0.00	0
	SmxCalculatorApp cla	1	0.00	0	0.00	0

Show

Testable methods Methods covered
 Nontestable methods Methods not covered

Methods

```
StmCalculatorView class>>#abtPrimFlushInterfaceSpe  
StmCalculator class>>#abtPrimFlushInterfaceSpecCac  
StmCalculatorView>>#setDecimal  
StmCalculatorView class>>#IS_instanceInterfaceSpec  
StmCalculatorView class>>#IS_displayTextObject
```

Method Source

```
setDecimal  
  
self decimal: true.
```

VisualWorks

Method Coverage Browser

File Edit View Tools Help

Calculator View Tests methods=45, tested=38 (84.4444%), not tested=7 (15.5556%)

- StmVisualWorksCalculatorViewsSubApp methods=30, tested=24 (80.0%), not tested=6 (20.0%)
 - TypeConverter class methods=1, tested=1 (100.0%), not tested=0 (0.0%)
 - StmVWCalculatorView methods=27, tested=21 (77.7778%), not tested=6 (22.2222%)
 - TypeConverter methods=1, tested=1 (100.0%), not tested=0 (0.0%)
 - StmVWCalculatorView class methods=1, tested=1 (100.0%), not tested=0 (0.0%)
- StmCalculatorApp methods=15, tested=14 (93.3333%), not tested=1 (6.66666%)
 - StmCalculator methods=13, tested=13 (100.0%), not tested=0 (0.0%)
 - StmCalculator class methods=1, tested=1 (100.0%), not tested=0 (0.0%)
 - StmCalculatorView class methods=1, tested=0 (0%), not tested=1 (100%)

Methods covered Methods not covered

StmVWCalculatorView>>#value
StmVWCalculatorView>>#value:
StmVWCalculatorView>>#zeroDisplay
StmVWCalculatorView>>#mathClearErrorAction
StmVWCalculatorView>>#setDecimal
StmVWCalculatorView>>#mathSetDecimal
StmCalculatorView class>>#open

setDecimal

self decimal: true.

Creating and Running Your First Test Case

In this section, you will create an automated test case for a simple system - a calculator. By the end of this exercise, you should have a rough idea of the steps that are involved with automating tests, but you probably won't understand everything that you did. That's all right - everything will be explained in the following chapters.

Note: You need to have loaded the calculator example that is provided to try these exercises yourself. If you are using *ENVY/developer*, load the **Smalltalk Test Mentor - Examples** configuration map. Otherwise, load the `$(VisualWorks)\STM-Calculator.pcl` parcel.

Using the Test Editor

To start the Test Editor, navigate to the Smalltalk System Transcript.

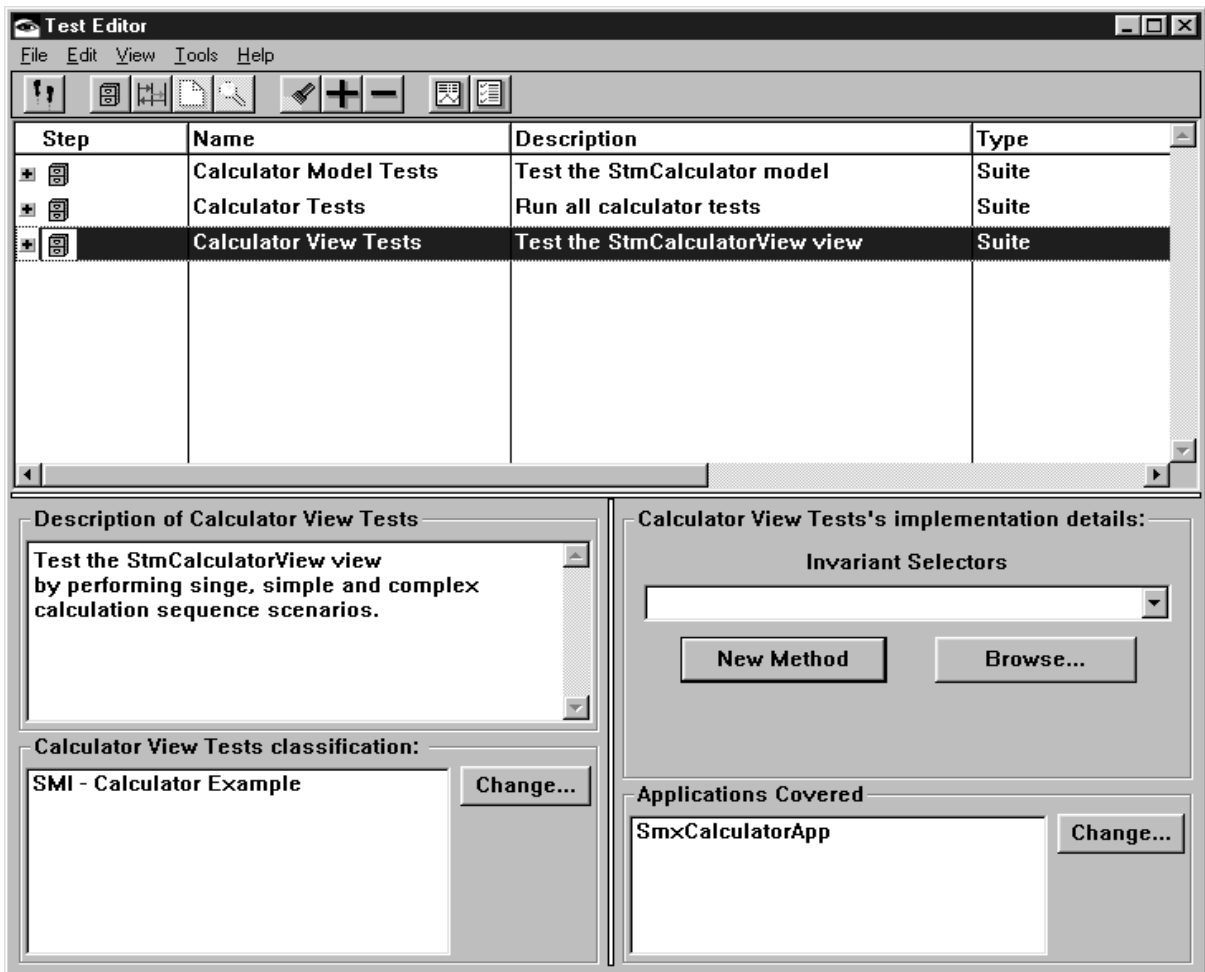
If you are using either VisualAge Smalltalk, or are using VisualWorks without *ENVY/developer*, select the **Tools** menu, and then **SilverMark's Test Mentor Editor...**

If you are using VisualWorks with *ENVY/developer*, select the **ENVY** menu and then **SilverMark's Test Mentor Editor...**

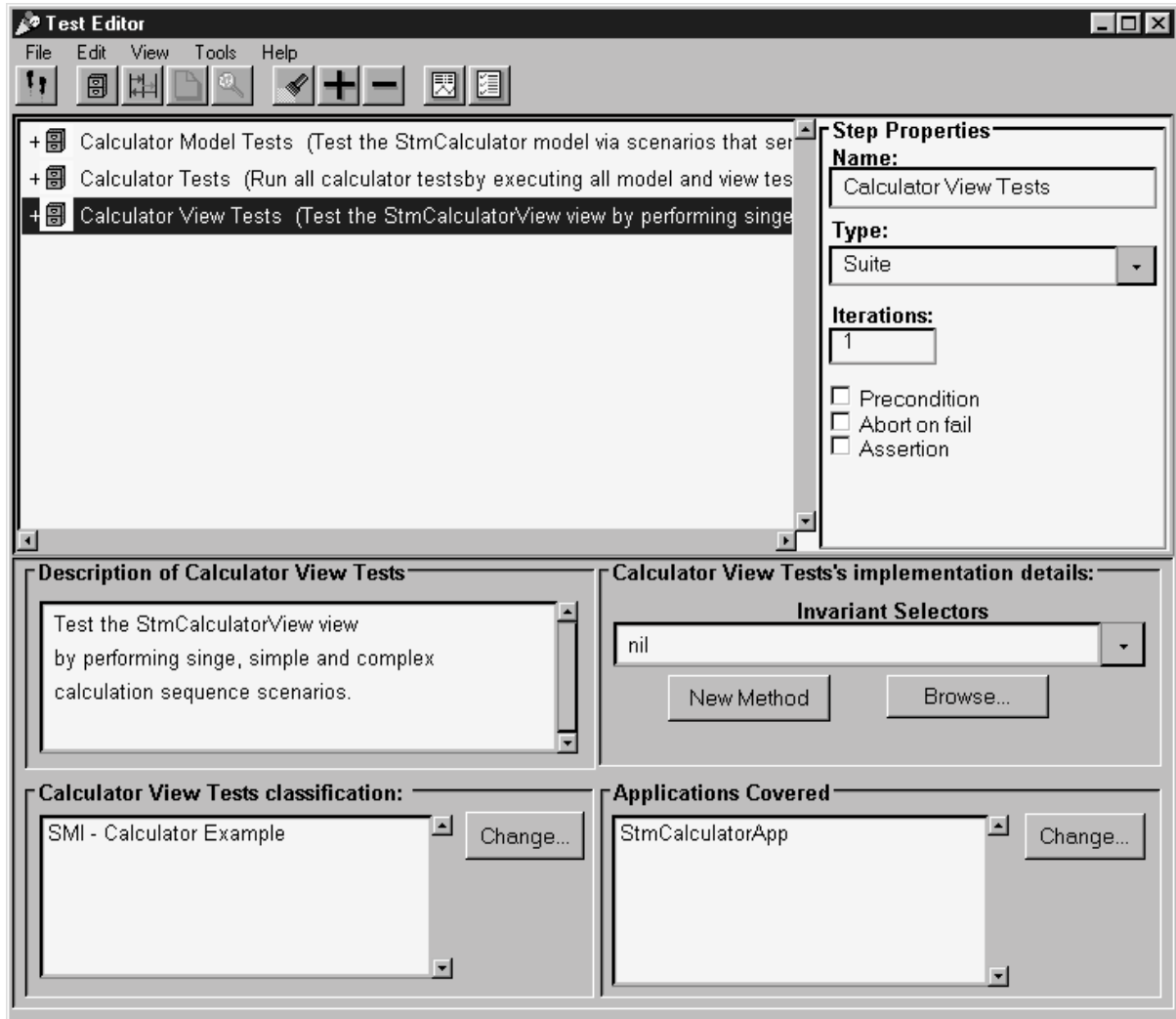
A window displays the set of tests that already exist in your Smalltalk image.


Note: Depending on which tests you already have in your image, the tests displayed by the editor may be different from those that appear in these examples.

In VisualAge, you would see this:



In VisualWorks, you would see this:



Click on the New Suite  icon in the tool bar. You will see a window titled New Suite appear:

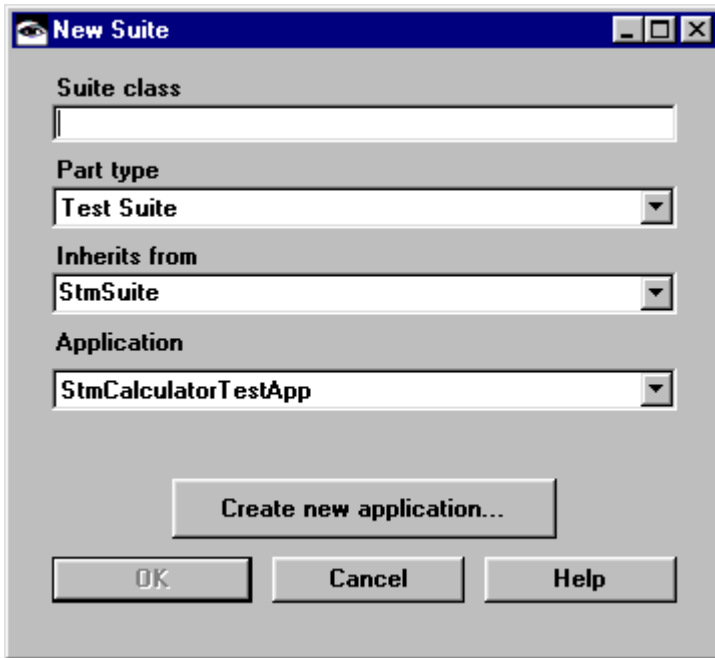


Figure 1 - New suite view with ENVY/developer

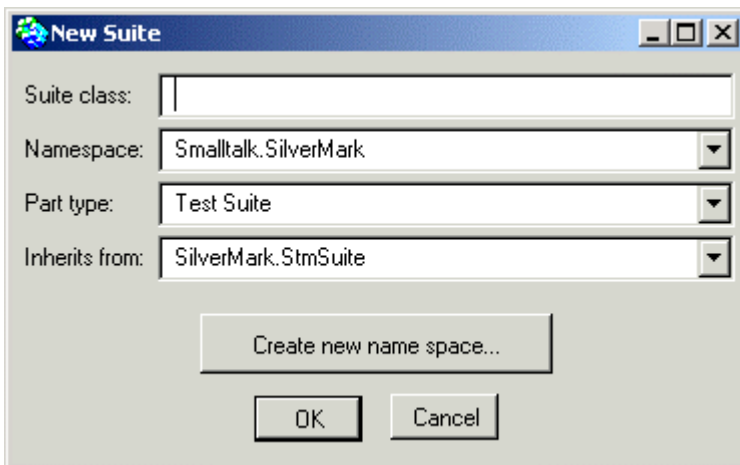


Figure 2 - New suite view in VisualWorks without ENVY/developer

Enter **MySuite1** for the suite class.

If you are using VisualAge or VisualWorks with ENVY/developer, Press Create New Application... You will be prompted for a new application name. Enter **MyTestApplication1** and press OK.

If you are using VisualWorks without ENVY/developer, press Create new name space... You will be presented with a name space selection dialog. Use this to select the name space to place the new name space:

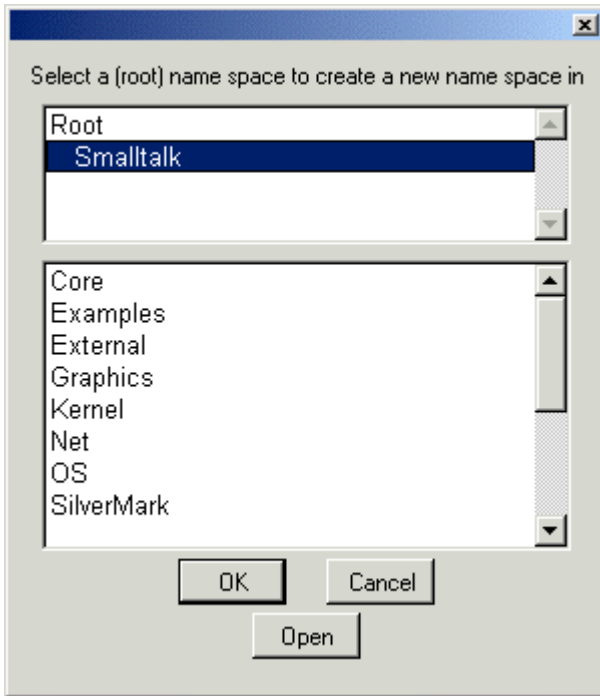


Figure 3 - Root name space selection dialog

Select **Smalltalk** and press OK. Once you select a root name space, you are prompted for the new name space name:

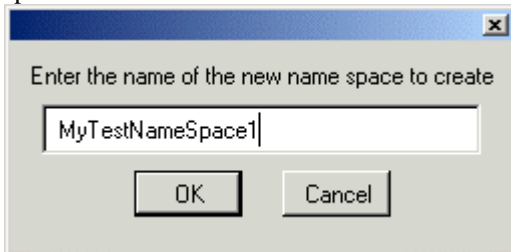
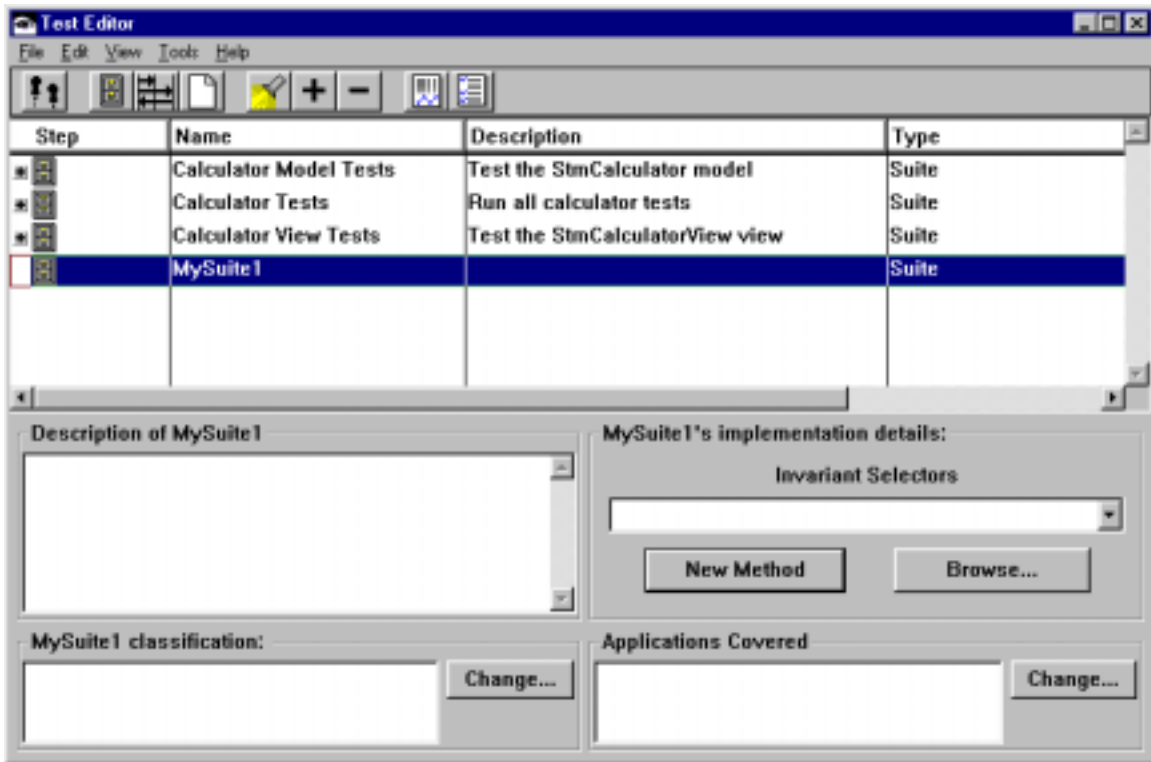



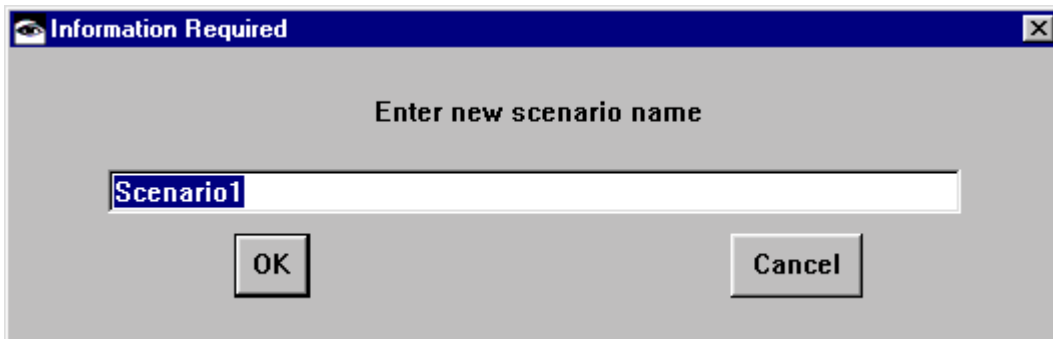
Figure 4 - New name space dialog

Enter **MyTestNameSpace1** and press OK.

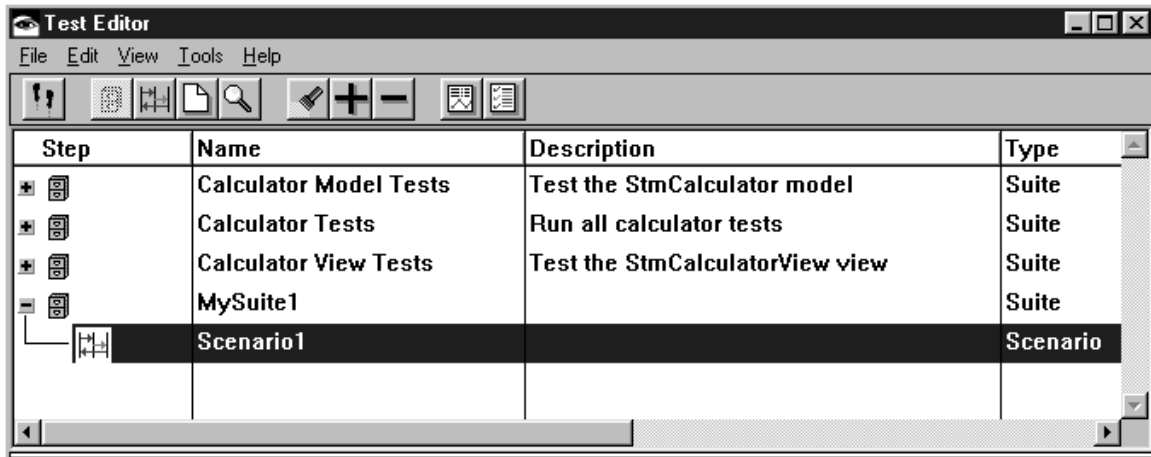
Press OK in the New suite window. You will see your new suite added like this:



You now have a class in which to place your tests. Make sure that your new suite is selected, and then click on the New Scenario  icon in the tool bar. You are prompted for a name for your scenario:

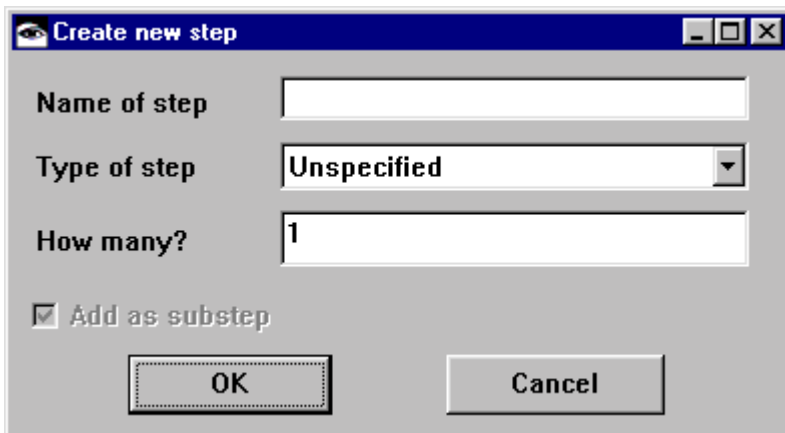


For this example, you can use the default name of **Scenario1**, although you are free to change it. When you press OK, A new scenario is added to your suite. The top of your editor should look like this:

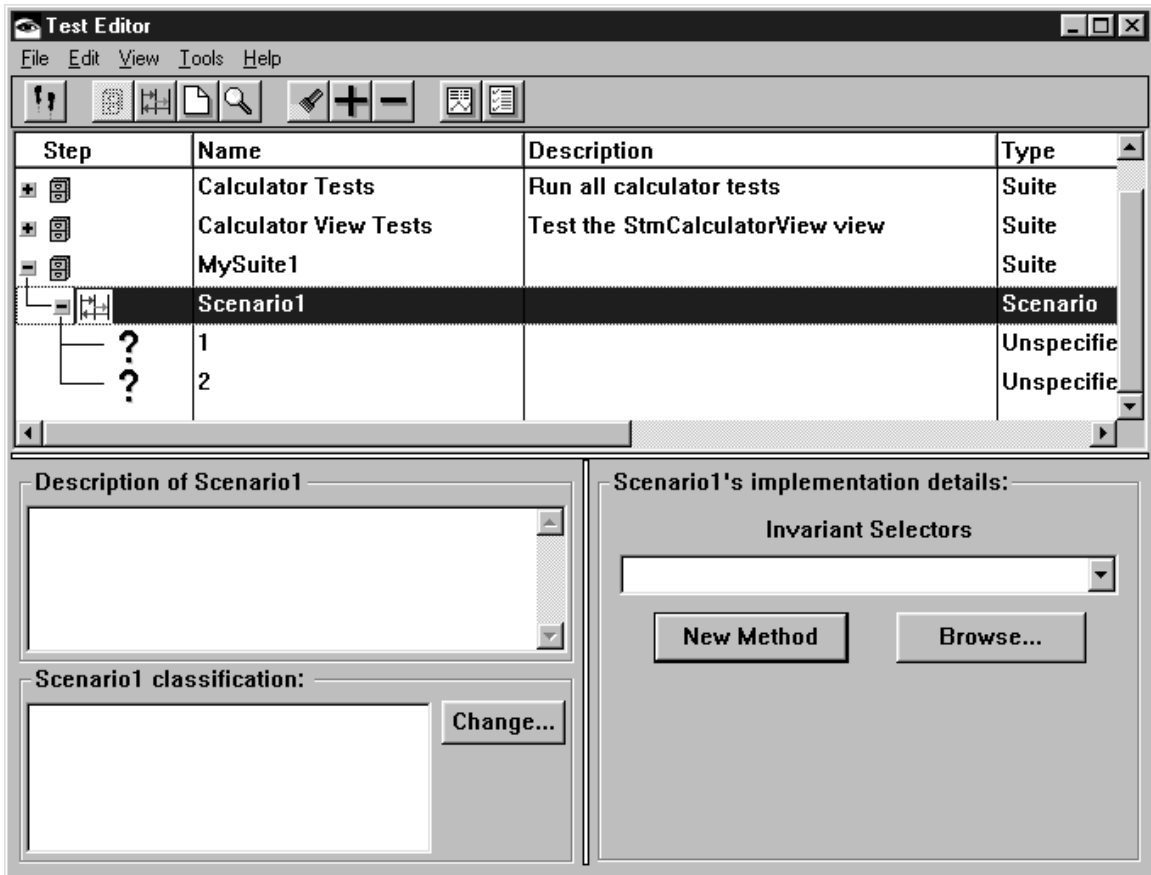


You can change the names of your suite and scenario by directly typing over them. With this preliminary work out of the way, you can now attack the task of actually testing your calculator. You do this by adding steps to your scenario.

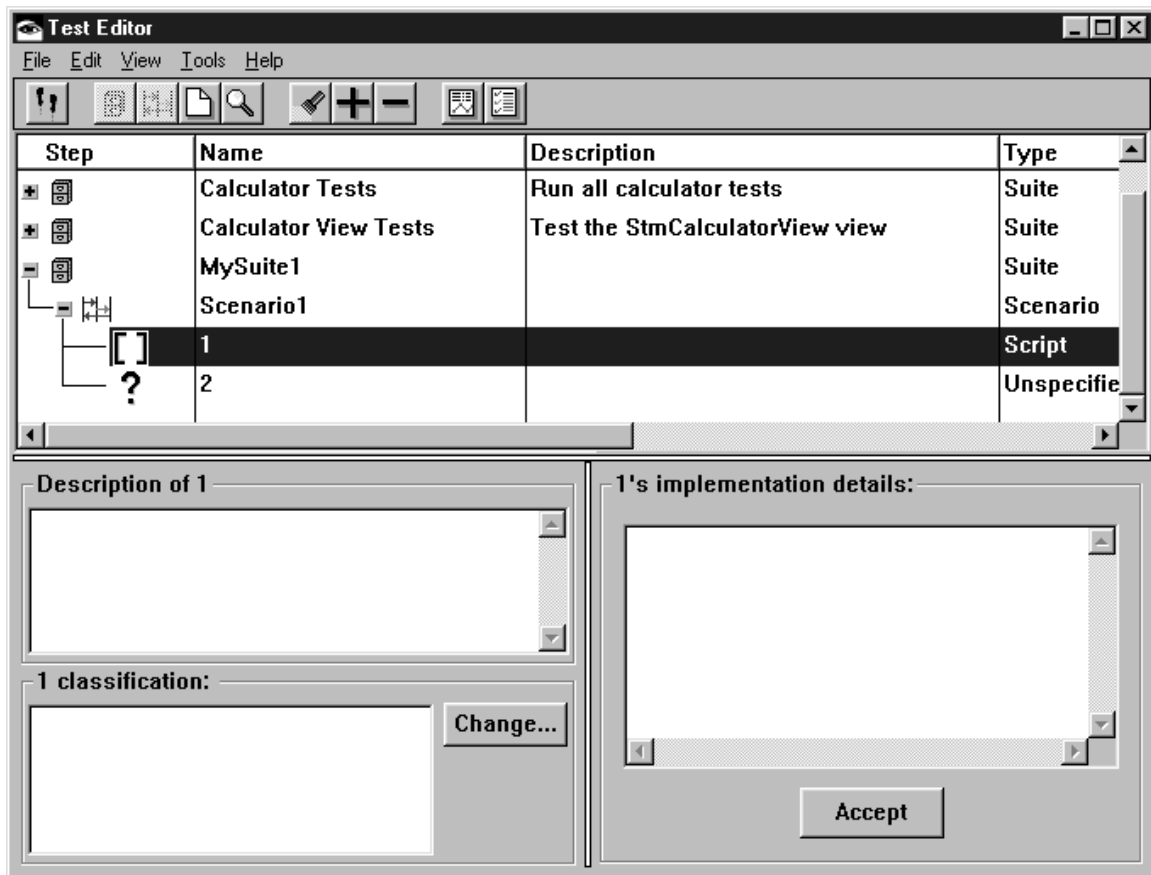
Select your new scenario and then click on the New Step  icon in the tool bar. You are presented with the following dialog:



For How Many?, enter 2. This is for the number of steps you are about to add. Press OK. The Test Editor should look like this:



The steps were automatically named '1' and '2'. You can change those names if you like by directly typing over them in the Name column. You are going to specify that the first step opens the calculator view and the second step actually tests it. To open the view you are going to need to execute a line of Smalltalk code. To do this, select the first step, click on the word **Unspecified** in the Type column and use the drop down list to select **Script**. Click on the icon for the step to have the change take effect. The Test Editor now looks like this:



In the text area under the label, **1's Implementation details**, enter the following Smalltalk code:

In VisualAge:

```
StmCalculatorView new openWidget.
```

In VisualWorks 2.5 or 3.0:

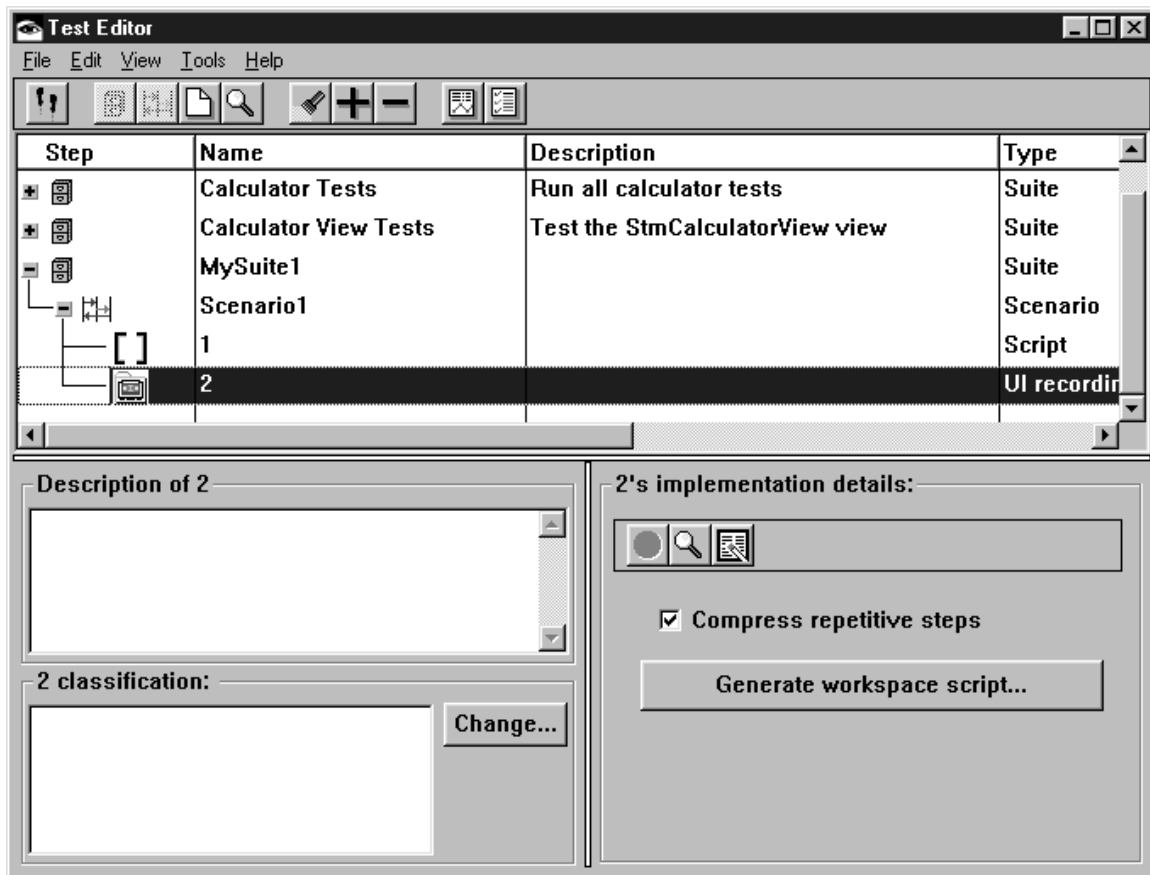
```
StmCalculatorView open
```


In VisualWorks 5i.4 or greater:

```
SilverMarkCalculator.StmCalculatorView open
```

After entering the value, press Accept. You have just entered a Smalltalk script to be executed for this step. This is the only Smalltalk code you will have to write for this test.

Now you can move on to the fun part - recording your user interface interactions. First you need to specify that the second step represents recorded user interface interactions. To do this, select the second step, click on the word **Unspecified** in the Type column and use the drop down list to select **UI recording**. Click on the icon for the step to have the change take effect. The editor should now look like this:



You can record user interface interactions by pressing the **Record**  button but there's an easier way. To show you this, we'll have to jump ahead to the next section on *Using the Quick Runner*, but before you do that, there's one more thing you might want to do.

If you are interested in method coverage metrics (that is, what percent of the calculator's methods are covered by the test you are creating) you should specify which classes the test is actually testing.

In VisualAge or VisualWorks with ENVY/developer:


Select your suite and press the **Change...** button in the **Applications Covered** group toward the lower right. Select **StmCalculatorApp** or **SmxCalculatorApp**, depending on the Test Mentor version you have installed, and press **OK**. Now the Smalltalk Test Mentor will know that when you are running your suite, you are testing methods in your calculator's application, so it can now measure method coverage.

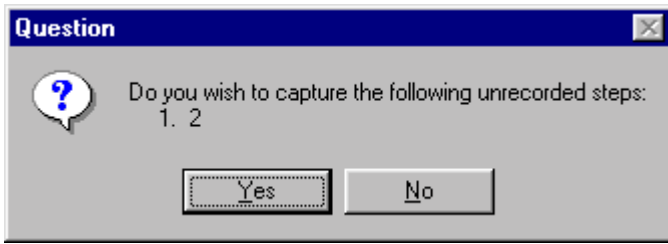
In VisualWorks without ENVY/developer:

Select your suite and press the **Change...** button in the **NameSpaces Covered** group toward the lower right. Select **SilverMarkCalculator**, in the name space selection dialog and press **OK**. Now the Smalltalk Test Mentor will know that when you are running your suite, you are testing methods in your calculator's name space, so it can now measure method coverage.

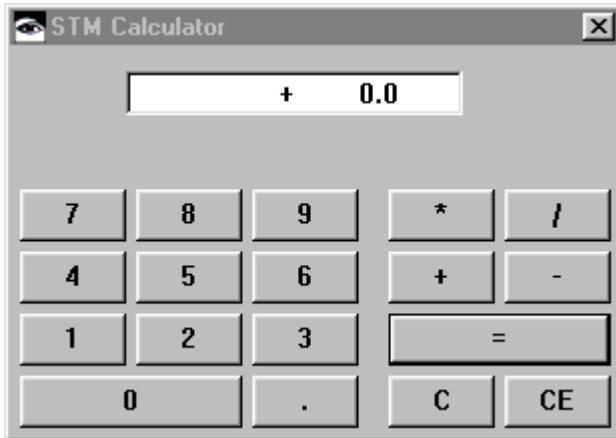
Using the Quick Runner

You can use the Quick Runner to run tests once you've defined them. You can also use the Quick Runner to capture user interface interactions for unrecorded user interface recording steps as they are executed. You will use this feature now to record your calculator view test. To do this, select your suite in the Test

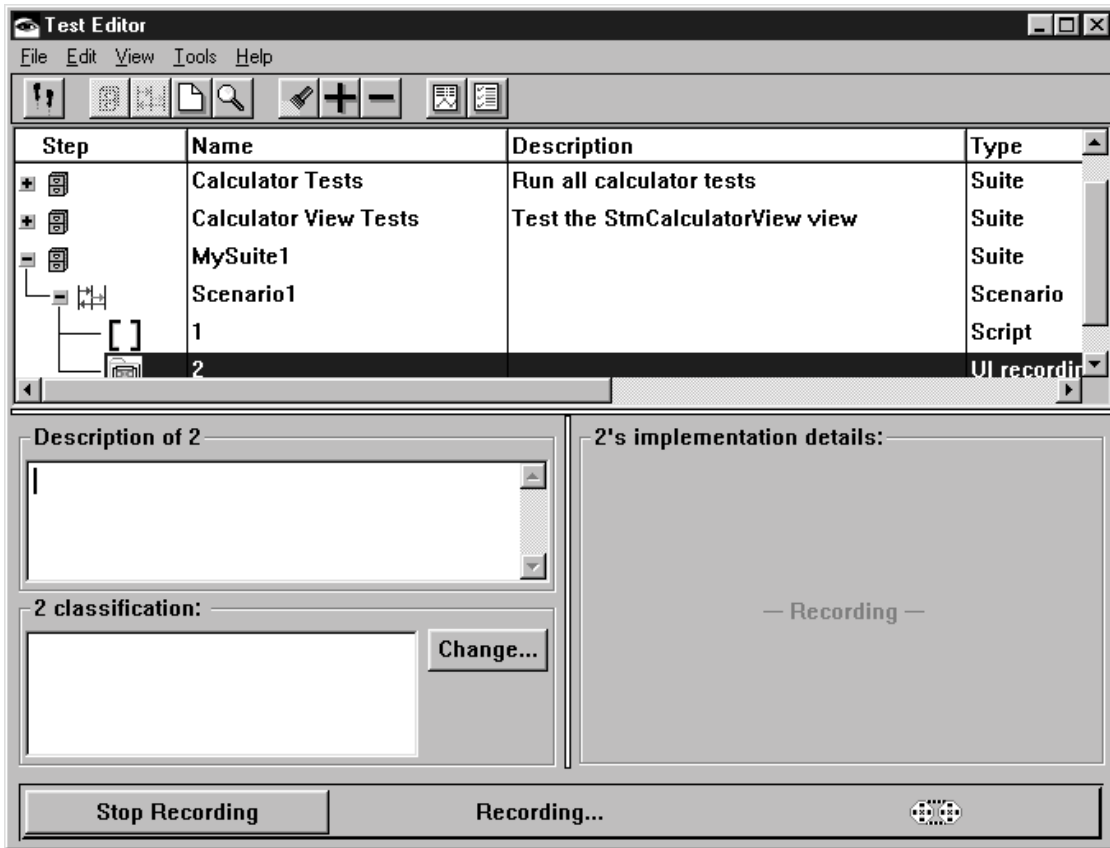
Editor and press the **Quick Run**  icon in the tool bar. You are presented with a prompt to capture user interface interactions for the empty user interface recording step:




Answer, Yes. When you do this, the Quick Runner executes the first step, which should open the calculator:



When the Quick Runner reaches the empty user interface recording step, the Test Editor switches into user interface recording mode. You can tell that it is in this mode by the user interface recording status area added to the bottom of the Test Editor:



Using your mouse, perform a simple calculation with the calculator, then close the calculator. When you are done, click on the Stop Recording button. The Stop Recording button changes to Resume Running. Press the Resume Running button to continue to the next step. Because the step you recorded was the last step, the Quick Runner completes running the steps and opens the Test Results Browser. You can close the Test Results Browser because we're not interested in that yet.

You have now finished creating your test. Go back to the Test Editor, make sure your suite is selected and press the Quick Run  button. You will see the calculator open, perform the operations you recorded and then close. At completion of the test, a window titled Test Results reappears.

Viewing the Results of Running Your Test

When the execution of a test completes, the results are shown in a Test Results Browser. If you've been following the above steps until now, you should see a Test Results Browser with your suite in a list on top.

Select your suite and press the Expand  button on the tool bar. You should see the following:

The screenshot shows the 'Test Results' window with a tree view on the left and a summary table at the bottom. The tree view shows a hierarchy: MySuite1 (Pass) -> Scenario1 (Pass) -> 1 (Pass) -> 2 (Pass) -> 2 (1) (Pass) -> 2 (2) (Pass) -> 2 (3) (Pass). The summary table below the tree view is as follows:

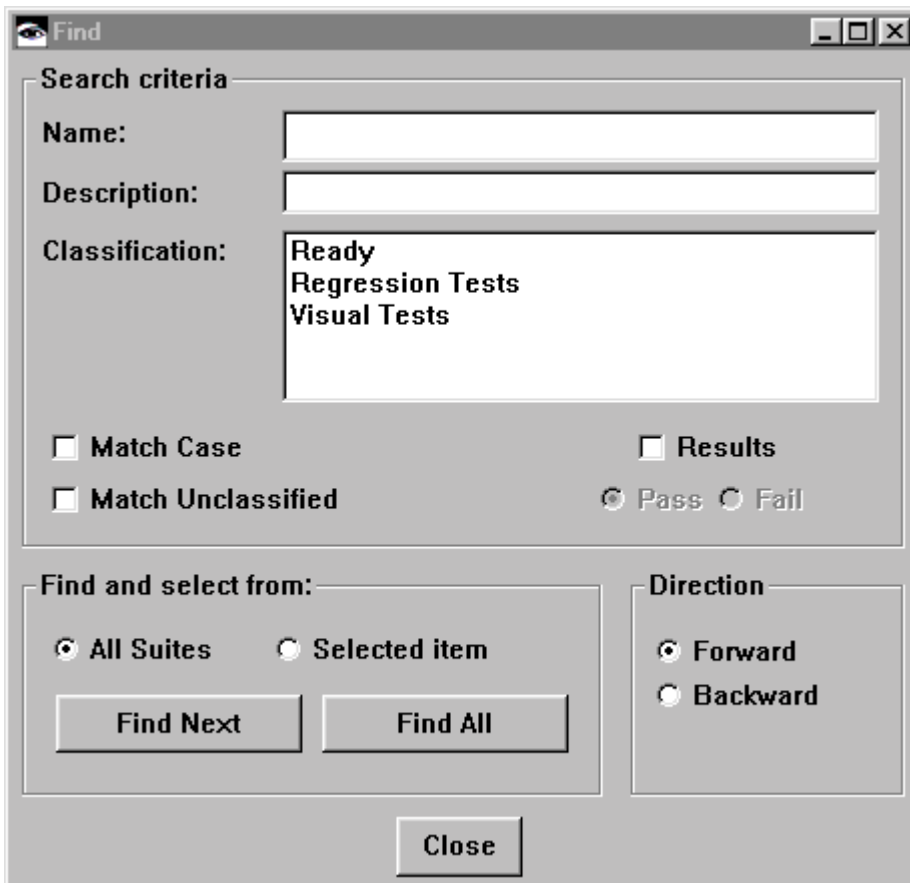
Name	Run	Pass	Fail	Time	Covered (%)	Testable (%)	Methods (#)	Testable (#)
MySuite1	9	9	0	963	43.59	60.71	39	28

Below the summary table, there is a text area containing the string: `'true|'`

The Test Results Browser shows the results of having run the suite, scenario and each step. You can see whether the step passed or failed by the traffic light icons. In this simple test, all the steps should have passed. The browser also shows execution time for each step, as well as metrics for code coverage. For a fuller explanation, you should read the appropriate sections in the *Smalltalk Test Mentor User's Reference*, and *Smalltalk Test Mentor User's Guide*.

Here's one last interesting thing you can try. If you would like to see statistics on the execution time of all the button clicks on the calculator view, try this:

In the Test Results Browser, press the right mouse button over the list of steps and select Find.... You should see the following dialog:



Enter **Click** next to Description and press the Find All button. All the test steps that have the **Click** text within their description will be selected. You should see the following:

Step	Pass	Name	Description	Execution time	Ty
		2 (2)	Click on <4>	69	UI
		2 (3)	Click on <5>	14	UI
		2 (4)	Click on <^>	71	UI
		2 (5)	Click on <4>	42	UI
		2 (6)	Click on <5>	14	UI
		2 (7)	Click on <=>	54	UI
		2 (8)	Close <STM Calculator>	164	UI

Name	Run	Pass	Fail	Time	Covered (%)	Testable (%)	Methods (#)	Testable (#)
2 (2)	1	1	0	69				
2 (3)	1	1	0	14				

Metric	Max	Min	Range	Mean	Median	Std Dev
Steps run	1	1	0	1.00	1	0.00
Steps passed	1	1	0	1.00	1	0.00
Steps failed	0	0	0	0.00	0	0.00
Execution time	71	14	57	44.00	71	25.53

Notice that when you select multiple items you see statistics at the bottom for those items. At the bottom are those for execution time. Because this is a trivial example, the numbers are not very interesting, except for the fact that each button click interaction with the calculator took the same fourteen milliseconds to execute.

Conclusion

You have now created a test that automatically exercises the user interface of the simple example view, run the test, and viewed the results. You can go back and create different scenarios for different uses of the view if you like or move on the other sections of this manual.

SilverMark's Test Mentor User's Reference

Test Case Structure

Overview

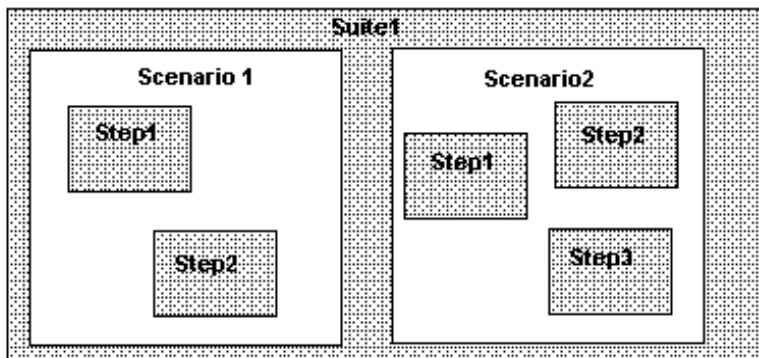
The Smalltalk Test Mentor test case structure is based on the notion that every test case is a step, and that a step can be composed of more steps. When you create a test case, you are assembling different types of steps - the *Suite* and the *Scenario*. A suite is the coarsest level of step granularity. It is used to collect related scenarios. A scenario corresponds to a particular usage of the system or part of the system under test. A particular usage of a system is often called a *use case*. Scenarios are composed of steps and, as you will see later, some steps can be composed of more steps.

For example, if you wanted to write a complete set of tests for a loan application software system, you might create a suite for each window in the system. For each window's test suite you would create a separate scenario for each use of that window. You would then fill each scenario with the steps required to execute the scenario's use case.

With the Smalltalk Test Mentor, you can reuse suites and scenarios within other suites and scenarios. This enables you to build up tests that call other tests for broader levels of testing. In the above example, you might create a test scenario that traverses multiple windows by reusing some of those windows' specific scenarios.

Test representation

One important thing to remember about test suites is that *each suite is represented by its own class*. When this book mentions a test class, it is really talking about a particular suite's class. All other steps are represented within methods within their suite's class.



In the above diagram, *Suite1* represents a suite. *Suite1* is defined by a Smalltalk class. The two scenarios defined within *Suite1*, *Scenario1* and *Scenario2*, are defined by methods within *Suite1*'s class. The various steps contained by *Scenario1* and *Scenario2* are also represented by methods within *Suite1*. An executing scenario is actually an instance of its suite's class populated with its particular steps. For more details see the section titled *Smalltalk Test Mentor Programmer's Guide*.

The Test Step

A test step embodies a single, atomic test on a system. A test step is the smallest unit of granularity for a scenario. You can make your scenarios as coarse or fine-grained as you like by giving steps more or less responsibility for implementing testing behavior. A scenario can be composed of one step or over one

hundred¹. Steps may be implemented in various ways, depending on the requirements of the test scenario and the characteristics of the system under test.

Test Steps differ in two ways - by their *implementation* and their *properties*.

Test Step Implementation

When you specify the type of a test step, you are specifying how you wish it to be implemented. Each different implementation of a test step has its own testing behavior.

Test Step Properties

All steps share a common specification of properties that define them, in addition to properties specific to their implementation. Because test suites and scenarios are each a special kind of step, many properties that are used to specify steps also apply to suites and scenarios. Common properties are shown in the following table:

Property Name	Property Description
Name	The name given to a suite, scenario or step. It can be any string of text. When code for a scenario is generated, the initial name for the scenario is used to generate a selector for the scenario. For more details see the section titled <i>Smalltalk Test Mentor Programmer's Guide</i> . When you create a new step, a default value for name is generated for you but it can be overridden.
Description	A text description of a suite, scenario or step. A typical usage of this is to have domain experts or test designers lay out suites, scenarios and steps prior to having a test developer actually define the specifics of a test. The default value is empty .
Classification	A list of names under which the step is classified. Once a step is classified, you can use search facilities to find steps classified in a certain manner. You can use any values you like as classifications. The default value is empty .
Precondition	Specifies whether passing the given suite, scenario or step is a precondition to executing any subsequent peer steps. The default value is false .
Abort on Fail	If a step fails and the parent of the step is set to abort on fail, none of the remaining steps for the parent will be executed. A typical usage for this is to avoid executing steps in a sequence once one of them fails. The default value is false .
Assertion	Specifies that the step returns a Boolean value and that the Boolean value determines whether the step passed or failed. This is most useful

¹ The steps for a scenario are code generated into a method. IBM Smalltalk V3 has a limitation on method size of approximately 64k bytes (later versions do not have this limit), thus limiting the number of steps that can be generated for a scenario. The actual number will vary depending on the size of the code generated for each step. VisualWorks has a limit of 256 method literals. To circumvent this limitation, long tests in VisualWorks are stored in methods as binary String representations of their respective compiled methods.

	in script, instance method, and class method steps.
Iterations	Specifies the number of times to execute the step when the step is run. You can use this to easily create stress tests by specifying a high number of iterations. You can also disable a step by setting its iterations to zero.

Class Method Step

A *class method step* is a step that performs a message send to a class. Often, class test methods are used to add unit-level or diagnostic testing behavior to model classes. The class method step enables you to execute those tests as steps. A class method step is specified by a class and one of its class methods' selectors.

Instance Method Step

An *instance method step* is a step that performs a message send to a method selector on the instance side of a test class (suite's class). Remember that test suites reside in a class that contains scenarios and steps. These test classes contain both the code for the specification of test cases and the implementation of test steps. This means that test case developers are able to implement instance methods that perform testing behavior within the same class that defines the Suites, Scenarios, and Steps.

Suite Step

This step is specified by the name of a test suite. This type of step is used to encourage the reuse of test suites within other test scenarios. Once you have created a test suite, it is available for reuse within other scenarios as a suite step. A suite step is not the same as a suite. A suite step is a step that creates a new instance of a particular suite and executes it. A suite embodies the specification of itself, as well. When you edit a suite, you can add scenarios to it. A suite step is a reference to a suite, not the suite itself - which means that when you edit this type of step, you can only specify which suite to use and not the implementation details of the suite step itself.

Note 1: When a suite step is executed, a new instance of it is created and executed. If a suite step references a suite with sub-steps and any of those sub-steps are instance method steps, the receiver for those instance method steps will be the instance of the referenced suite - not the instance of the suite under which the suite step is running.

Note 2: The execution time for a suite step is the sum of the execution times for the scenarios that it contains.

Scenario Step

This step is specified by the name of a test scenario within a specified test suite. This type of step is used to encourage the reuse of test scenarios within other scenarios. Once you have created a test scenario, it is available for reuse within other scenarios as a scenario step. A scenario step is not the same as a scenario. A scenario step is simply a step that creates a new instance of a particular scenario and executes it. A scenario embodies the specification of itself, as well. When you edit a scenario, you can add steps to it. When you edit a scenario step, you simply specify which scenario to use. In short, a scenario step is a reference to a scenario, not the scenario itself.

Note 1: When a scenario step is executed, a new instance of it is created and executed. If a scenario step references a scenario with sub-steps and any of those sub-steps are instance method steps, the receiver for those instance method steps will be the instance of the referenced scenario - not the instance of the suite under which the scenario step is running.

Note 2: The execution time for a scenario step is the sum of the execution times for the steps that it contains.

Workspace Step

A *workspace step* is a step that is specified by the path and file name of a Smalltalk workspace. When this type of step is executed, the file is read, compiled and executed. The intent of providing this type of step is to enable users that already have a base of test code in workspaces to become immediately productive with the tool. We recommend that ultimately, the code in the workspaces be migrated to methods within test classes so they can be under Envy source control. This type of step cannot be executed in a packaged runtime image.

Manual Intervention Step

A *manual intervention* step is specified as text that is displayed when the step is executed. This type of step is typically used when, during the course of a test, the person executing the test is required to perform some physical action that cannot be performed programmatically. For example, a test of the robustness of a communications driver that physically loses a physical connection might precipitate the use of a manual intervention step that instructs the person executing the test to physically disconnect a cable. The instruction to perform a manual intervention step is displayed in a window. Once the person running a test has performed the task as described in this window, they may press a pass or fail button, and enter some notes to be stored with the results of executing the step.

Note: The execution time for a manual intervention step is always zero.

Collection Step

A *collection step* is a step that holds other steps. Its should be used for adding structure to a test case. For example, you may want to create a test scenario that has setup, body and cleanup collection steps. Each collection step contains the steps required to perform its respective task.

Note: The execution time for a collection step is the sum of the execution times for the steps that it contains.

Conditional Collection Step

The conditional *collection step* is derived from the collection step with the additional behavior of only executing its child steps if its condition is met. The condition is defined by a Smalltalk expression that is assumed to return a Boolean.

File Iteration Step

The *file iteration step* is derived from the collection step and operates similarly except that the step is executed once for each row in a specified ASCII test file.

A file iteration step works as follows when executed:

1. While there are unread records in the test data file, do:
2. Read the next record from the test data file and parse the values into Test Mentor variables.
3. Execute the steps contained by the file iteration step. It is assumed that these steps will use the variables that have been set above in some meaningful way.

The format of the test data file should follow the following syntax as specified in Backus-Naur Form (BNF):

row ::= comment | items

comment ::= commentCharacter commentText

items ::= item | items separator item

item ::= nameOfVariable assignmentCharacter valueOfVariable

nameOfVariable ::= name | name conversionIndicator

conversionIndicator ::= < conversionCharacter >

conversionCharacter ::= B | C | F | N | S

separator ::= any character not expected to be a member of the characters in the above

assignmentCharacter ::= any character not expected to be a member of the characters in the above

commentCharacter ::= the character, '/', which may be changed via StmStates>>#commentCharacter:

commentText ::= any text

Note: the conversion characters stand for Boolean, Character, Float, Number, String. The default is String. A conversion specifier of Number results in asNumber being sent to the value.

When values are extracted from a row, they are stored as variables accessible through the #varNamed: message.

Consider the following example file:

```
Name=Munster, H.@Address=1313 Mockingbird lane@Weight<N>=324@Height<N>=120@Gender<C>=M@Smoker<B>=true
Name=Munster, L.@Address=1313 Mockingbird lane@Weight=130@Height<N>=60@Gender<C>=F@Smoker<B>=false
Name=Gilligan, W.@Address=Uncharted Desert Island@Weight<N>=110@Height<N>=67@Gender<C>=M@Smoker<B>=false
Name=Skipper@Address=Uncharted Desert Island@Weight<N>=230@Height<N>=69@Gender<C>=M@Smoker<B>=false
```

The separator character used is @ and the assignment character is =. These are the default values, but can be overridden in the Test Editor for the step.

In the above example, each row defines values for the following variables, converted to objects of the following classes and accessible by the following code:

Variable name	Conversion	Code to access
Name	String	self varNamed: 'Name'
Address	String	self varNamed: 'Address'
Weight	Number	self varNamed: 'Weight'
Height	Number	self varNamed: 'Height'
Gender	Character	self varNamed: 'Gender'
Smoker	Boolean	self varNamed: 'Smoker'

Using #varNamed: will be discussed in the section entitled *Smalltalk Test Mentor Programmer's Guide*.

Short cut: In the above example, it was only necessary to specify repeated items once. For example, the above example file could be shortened to the following with the same effect:

```
Name=Munster, H.@Address=1313 Mockingbird lane@Weight<N>=324@Height<N>=120@Gender<C>=M@Smoker<B>=true
Name=Munster, L. @Weight=130@Height<N>=60@Gender<C>=F@Smoker<B>=false
Name=Gilligan, W.@Address=Uncharted Desert Island@Weight<N>=110@Height<N>=67@Gender<C>=M
Name=Skipper @Weight<N>=230@Height<N>=69
```

Script Step

A *script step* specifies a block of Smalltalk code to execute as a step. The Test Editor compiles the block within the context of its scenario's instance.

User Interface Step

A user interface step is a special form of script step that delays for a set amount of time after executing the script, to give the user interface a chance to update the display. This time is specified in the Preferences settings or by executing:

```
StmStates settlingTime: <anInteger>
```

The user interface recording facilities of the Test Editor create user interface steps for you. You can also create them yourself. See the section titled *User Interface Playback APIs* for specific information on scripting user interface steps.

User Interface Verification Step

A *user interface verification step* is a special kind of user interface step that passes or fails based on the Boolean value returned by the script. Use this step to verify whether a widget is in an expected state. The Smalltalk code for this step can be automatically generated for you via the Visual Verification Wizard, or you can write the script yourself for especially complex verifications. See the section titled, *User Interface Playback APIs* for specific information on scripting user interface verification steps.

User Interface Recording Step

A User Interface Recording step is effectively a collection step with additional edit-time behavior for recording steps. You can use the Test Editor to create user interface steps within a user interface recording step. If you are not interested in using the user interface recording facility of the Test Editor, you can safely replace a user interface recording step with a collection step.

Unspecified Step

An unspecified step is a place holder for later specification of a step's implementation. The typical usage for an unspecified step is for a domain expert to lay out a sequence of unspecified steps to describe the flow of a particular usage scenario for a system under test. The domain expert does not need to understand Smalltalk to do this. All they need to do is specify each step's name, description and operational properties like *precondition* and *abort on fail*. At a later time, test case developers would specify the implementation particulars for the unspecified steps. To make this simpler, there are facilities to morph steps from one implementation to the other within the Test Editor.

The Test Scenario

A test scenario, like the suite, is a special type of test step. A test scenario is mostly a structural unit. Its main purpose is to represent a particular usage of the system under test as a collection of test steps. In addition to the common properties, test suites are specified by the following additional properties:

Property Name	Property Description
Invariant Selector	<p>An invariant is an assertion about the state of the system under test that is assumed to always be true regardless of the actions upon that system.</p> <p>An invariant selector is a selector for a method within the scenario's test class (the class associated with the scenario's owning suite). This message implements behavior that verifies the presumably unchanging state of the system under test. When an invariant selector is specified, an instance method step, with the invariant selector specified as its selector, will execute after each of the scenario's steps. This is a powerful feature that enables you to automatically test for invariant conditions after each step without having to create many extra test steps.</p> <p>Note: The <u>precondition</u>, <u>assertion</u> and <u>abort on fail</u> properties for</p>

	the automatically created instance method step are both set to true .
--	--

The Test Suite

A test suite is a special type of test step. When you create a new test suite, you also create a new Smalltalk class that corresponds to the suite. When you use the Test Editor and save your changes, all code for that suite, including its scenarios and the scenarios' steps, are generated into the suite's class. In most cases, when you create test methods for a suite, you will create them within the suite's class. You may think of a suite as a test class. In addition to the common **properties**, test suites are specified by the following **additional properties**:

Property Name	Property Description
Invariant Selector	This is the same properties as the one defined for the test scenario, with the exception that when an invariant selector is specified, the message indicated by the selector will be sent after each of the suite's scenarios is executed.
Applications covered (with ENVY/developer) or Namespaces covered (VisualWorks without ENVY/developer)	<p>In order for the Smalltalk Test Mentor to calculate test case method coverage, it needs to know which methods are being tested.</p> <p>In versions of Smalltalk that use ENVY/developer, Test Mentor uses ENVY/developer applications to determine which classes and methods to observe.</p> <p>In versions of VisualWorks Smalltalk that do not use ENVY/developer, Test Mentor uses name spaces to determine which classes and methods to observe.</p> <p>You use the applications or name spaces covered property of the test suite to specify which methods are under test. When you execute a test, the Smalltalk Test Mentor monitors these methods to see which ones are executed as a result of executing the test. This metric provides a measurement of how effective the test is.</p>

Test Mentor Views

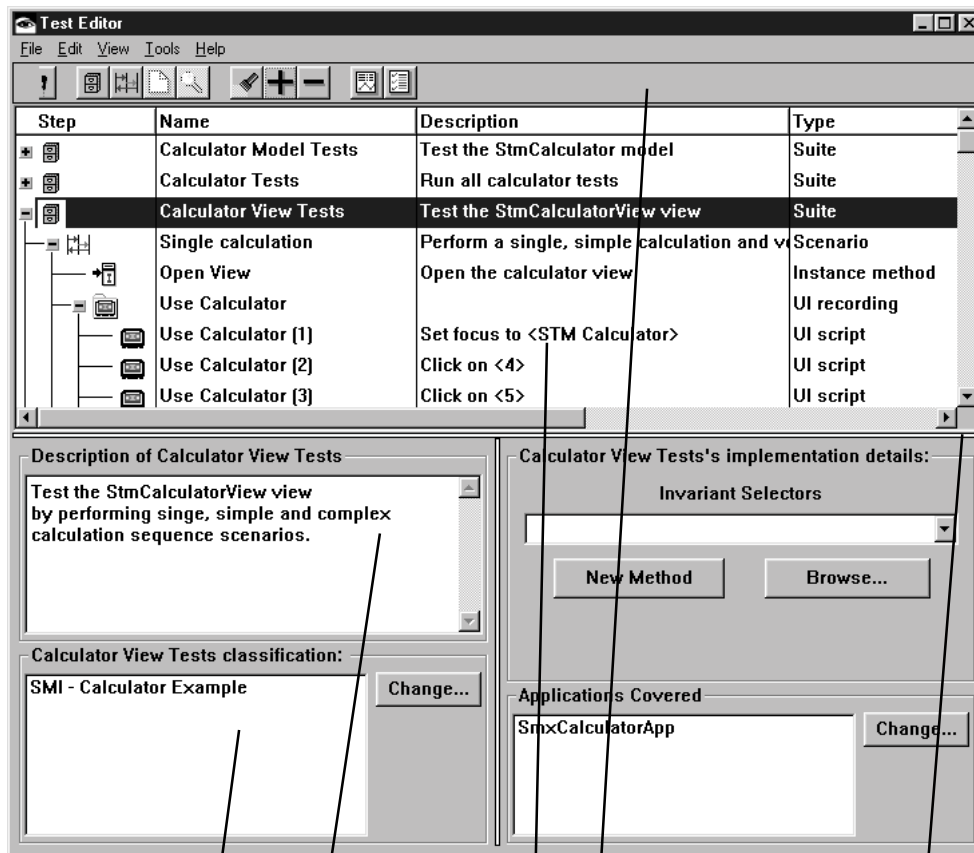
Although the Smalltalk Test Mentor was designed to be used with or without a user interface, use of the views provided by the product can make testing tasks much simpler and the learning curve much less steep.

If you look at the screen captures of the Smalltalk Test Mentor views that appear on the next few pages, you will notice a consistent theme that runs from view to view. Most views are structured into an *upper* and a *lower* section. The upper section contains a hierarchical list of items of one sort or another. In this manual, the upper section will sometimes be referred to as the *list view*. The lower section is used to present details about the item or items selected in the upper section. In this manual, the lower section is sometimes called the *details view*. The two sections are separated by a movable sash that enables you to change the proportional size of the sections. This structure is the same whether you are editing a test case or viewing the results of the execution of it - only the information presented and behavior of the views differ.

The Test Editor

The Test Editor is one of the two Smalltalk Test Mentor views you can open from the System Transcript and VisualAge Organizer (VisualAge only). You use this to view the implementation of existing tests and to create new ones.

VisualAge:



Classificati
on for step

Description
of step

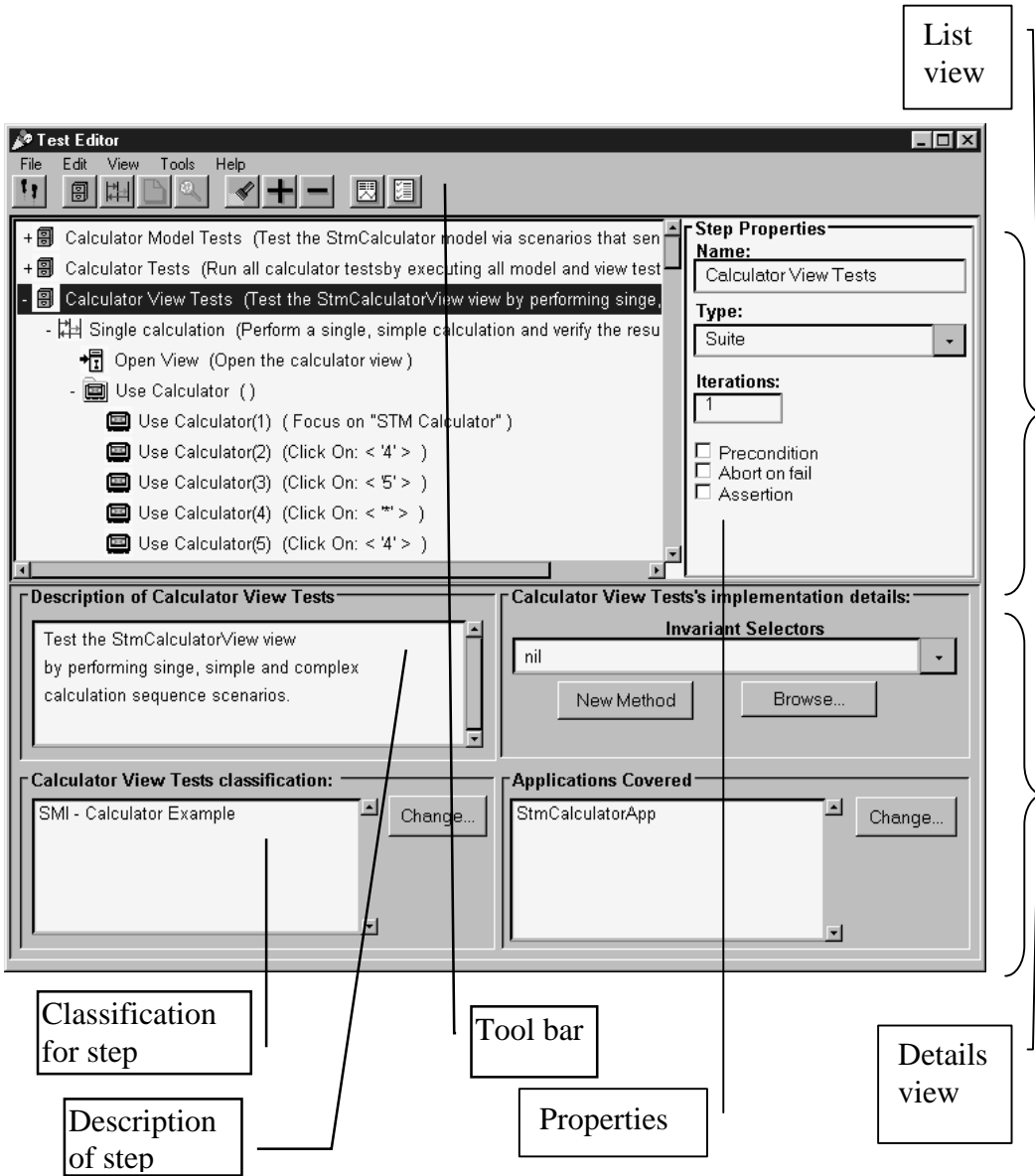
List view

Tool bar

Movable sash















Step details
view

VisualWorks:



The List View

The upper section is a hierarchical list of all the test steps in the image. Each row corresponds to a step. When you first open the view, only the suites in the image are displayed. You can expand a suite to show its scenarios by double clicking on the suite's icon. You can then expand a scenario to show its steps by double clicking the scenario's icon². You can use the same technique to expand any step that contains sub-steps. The icons used for each type of step are shown in the following table:

Step type	Icon
Class method	
Collection	
Conditional Collection	
File Iteration	
Instance method	
Manual intervention	
Suite	
Scenario	
Script	
User interface	
User interface verification	
User interface recording	
Unspecified	
Workspace	

The list view shows each step's icon and how the step fits into the hierarchy of steps, as well as its name, description, precondition and abort-on-fail properties. As discussed in *the Differences between VisualAge and VisualWorks* section of *Getting Started*, in VisualAge the list view shows properties in columns. And in VisualWorks, there is a separate area of the list view that shows the properties for a selected item.

Column or property name	Description
Step	The step column shows the icon and containment relationship between a step and its parent steps.

² In VisualAge Smalltalk you can expand rows by clicking on the + next to the icon. Doing this in VisualWorks has no effect.

Name	The name column contains the step names. This is editable via a text control when a cell within the column has focus.
Description	The description column contains the step descriptions. This is editable via a text control when a cell within the column has focus. You can also edit the description in the lower, details view. The latter mechanism is usually more convenient because the Multi-Line Entry control is better suited for long descriptions.
Type	<p>The type column contains a description of the step implementation types. Values are:</p> <ul style="list-style-type: none"> Class Method Collection of Steps Conditional collection of steps File iteration Instance method Manual intervention Scenario Suite Script UI Script UI Verification Script Unspecified Workspace <p>Cells in this column are editable via a Drop-Down List control when they have focus.</p>
Iterations	The number of times to execute the step
Precondition	The precondition column indicates the precondition property of its respective steps.
Abort on Fail	The abort on fail column indicates the abort-on-fail property of its respective steps.
Assertion	Whether to expect a Boolean value to be returned by the step and to use that value to determine whether the step has passed.

The Details View

When you select a single row (step) in the upper section (list view), the lower section (details view) configures itself to best show the details of the selected step. If you select multiple steps, the details view becomes disabled. When the details view is enabled, you can set values for the selected step. The following section shows details views for each type of step. Each screen capture shows the configuration of the details view when a step of a particular type is selected. The following screens are taken from various configurations that have been used in testing the product. You can safely ignore the particulars of the steps' descriptions and implementation details.

Class Method Step Details View

The screenshot shows a dialog box titled "Class Method Step Details View" with two main sections. The left section, "Description of Model Tests", contains a text area with "Perform calculator model self test". Below it, "Model Tests classification:" lists "SML - Calculator Example" and "SML - Internal Regression" with a "Change..." button. The right section, "Model Tests 's implementation details:", has a "Class:" dropdown set to "StmCalculator" with a "Change class..." button, a "Class method:" dropdown set to "selfTest", and a "Show superclass methods" checkbox which is unchecked.

The class method step is configured by specifying a class and a class method selector for the class. Two drop-down lists are used for these items.

Note: If you are using ENVI/developer, the choice of possible classes is determined by the applications you specified for the *applications covered* property of the step's owning suite. Only classes controlled by those applications will appear in the drop-down list of classes.

Collection of Steps Details View

The screenshot shows a dialog box titled "Collection of Steps Details View" with two main sections. The left section, "Description of Collection of Steps", contains a text area with "Clean up all open views". Below it, "Collection of Steps classification:" lists "Structural Cleanup" with a "Change..." button. The right section, "Collection of Steps's implementation details:", contains the text "No editable implementation details".

There are no properties to set in the details view for a collection step. A collection step is configured by the steps it contains.

Conditional Collection Step Details View

The screenshot shows a software interface for configuring a conditional collection step. It is divided into two main panes. The left pane, titled 'Description of Dialog interactions', contains a text area with the text 'Handle confirmation dialog if open' and a 'Dialog interactions classification:' section with a 'Change...' button. The right pane, titled 'Dialog interactions's implementation details:', contains a text area with the text 'ActiveWindow checkForOpenedWindowTitled: 'Confirm'' and an 'Accept' button.

A conditional collection step is like a collection step with the additional property of a script that when executed is expected to evaluate to a Boolean. When the step executes, it evaluates its script. If the script evaluates to True, the contained steps are executed.

File Iteration Step Details View

The screenshot shows a software interface for configuring a file iteration step. It is divided into two main panes. The left pane, titled 'Description of New Customers', contains a text area with the text 'Apply data in test data file to the 'New Customer' dialog to create a set of test customers.' and a 'New Customers classification:' section with 'Regression Tests' and 'Runnable' and a 'Change...' button. The right pane, titled 'New Customers's implementation details:', contains a 'Text data file name:' field with the value 'newCust.tbd', 'Open' and 'Browse...' buttons, a 'Separator:' dropdown menu set to '@', and an 'Assignment:' dropdown menu set to '='.

A file iteration step is specified by the file to read data from, the character to separate variable assignments by and the character to use for those assignments.

An example row might take the form of the following:

```
Name=Munster, H.@Address=1313 Mockingbird lane@Weight<N>=324@Height<N>=120@Gender<C>=M@Smoker<B>=true
```

The '@' character separates each variable assignment and the '=' character is used for assignment. See the previous section describing the file iteration step for more details.

Instance Method Step Details View

The screenshot shows a dialog box for configuring an instance method step. It is divided into two main sections. The left section, titled "Description of Instance Method Step", contains a text area with the text "Open the calculator view" and a "Change..." button. Below this is the "Instance Method Step classification:" section, which lists "Build Verification", "Regression", and "UI Automation", with a "Change..." button to the right. The right section, titled "Instance Method Step's implementation details:", contains a "Selectors:" dropdown menu currently showing "openView". Below the dropdown are two buttons: "New Method" and "Browse...".

The instance method step is configured by specifying a selector for an instance method within the step's owning suite's class, from the drop-down list of instance method selectors. You can create a new method with the New method... button, or browse the class to edit its code.

Suite Step Details View

The screenshot shows a dialog box for configuring a suite step. It is divided into two main sections. The left section, titled "Description of Calculator Model Tests", contains a text area with the text "Test the StmCalculator model" and a "Change..." button. Below this is the "Calculator Model Tests classification:" section, which lists "Ready", "Regression Tests", and "Visual Tests", with a "Change..." button to the right. The right section, titled "Calculator Model Tests's implementation details:", contains a "Suite:" dropdown menu currently showing "Calculator Model Tests[StmCalculatorModelTests]".

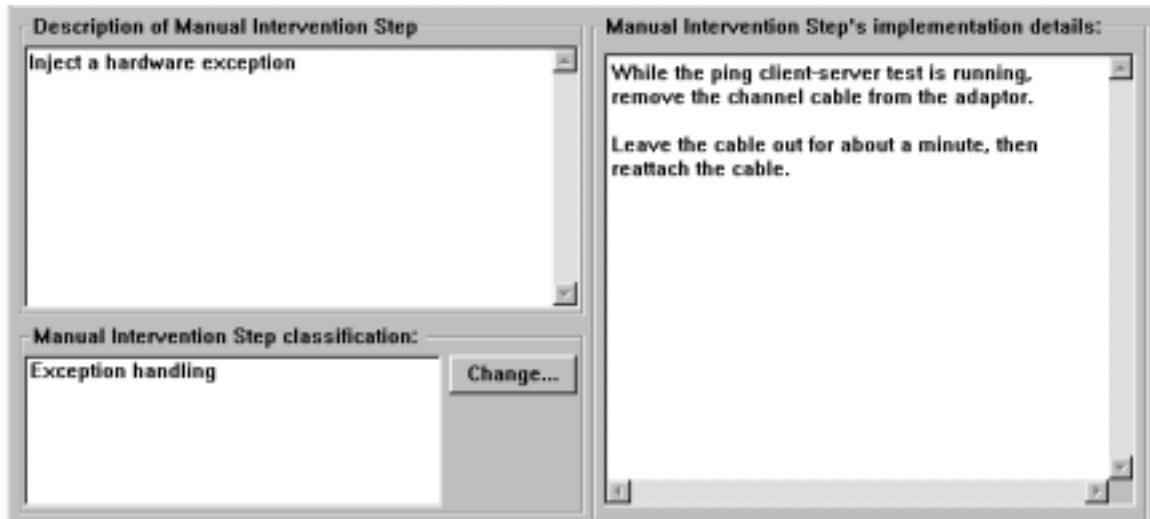
The suite step is configured by selecting a suite. You select a suite from a list of suite classes (subclasses of StmSuite) and their names in a drop-down list.

Scenario Step Details View

The screenshot shows a dialog box for configuring a scenario step. It is divided into two main sections. The left section, titled "Description of Calculator Model Tests", contains a text area with the text "Reuse complex calculations scenario." and a "Change..." button. Below this is the "Calculator Model Tests classification:" section, which lists "Ready", "Regression Tests", and "Visual Tests", with a "Change..." button to the right. The right section, titled "Calculator Model Tests's implementation details:", contains a "Suite:" dropdown menu currently showing "Calculator Model Tests[StmCalculatorModelTests]" and a "Scenario:" dropdown menu currently showing "Complex Calculations".

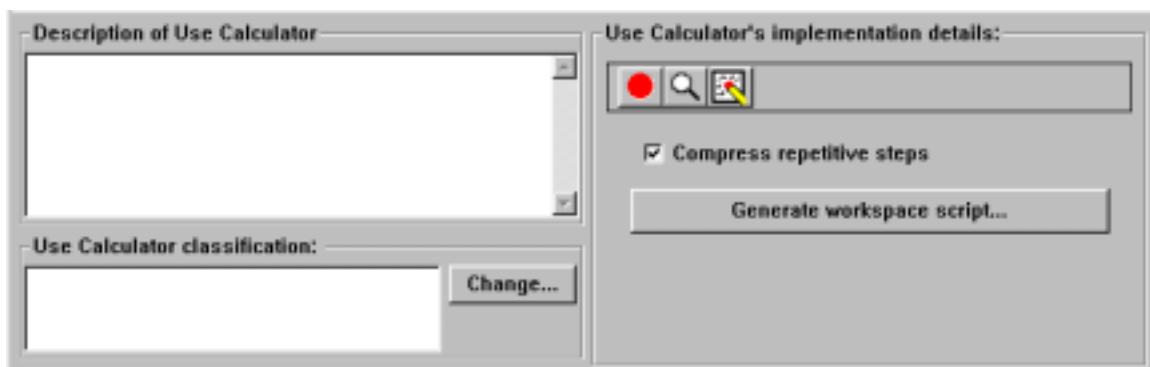
The scenario step is configured by selecting a suite, and a scenario within it.

Manual Intervention Step Details View




A manual intervention step is specified by the text to be displayed when the manual intervention step prompt is shown.

User Interface Recording Step Details View



Recording User Interface interactions

The user interface recording step is typically configured by recording steps into it. This is done by using the record  button, although you could also use the short cut that you saw in the *Getting Started* section. Recording is covered in more detail in *How to Record User Interface Interactions* in the *Smalltalk Test Mentor User's Guide*, and the *Recording User Interface Interactions*.

You can generate a workspace filled with the Smalltalk code that implements all user interface script steps or user interface verification script steps, contained within the user interface recording step, by pressing the Generate workspace script... button. You should consult the *Smalltalk Test Mentor Programmer's Guide* section for information on the Smalltalk code that is generated.


Some widgets for which values are incrementally set send a notification on each value change. All of these extra (and unnecessary) notifications are recorded. For example, typing "Smalltalk" into a text widget will cause the following values to be recorded:

"S"
"Sm"
"Sma"
"Smal"
"Small"
"Smallt"


"Smallta"
"Smalltal"
"Smalltalk"

To avoid this, check Compress repetitive steps, which suppresses multiple intermediate steps, leaving only a step that sets the final value.

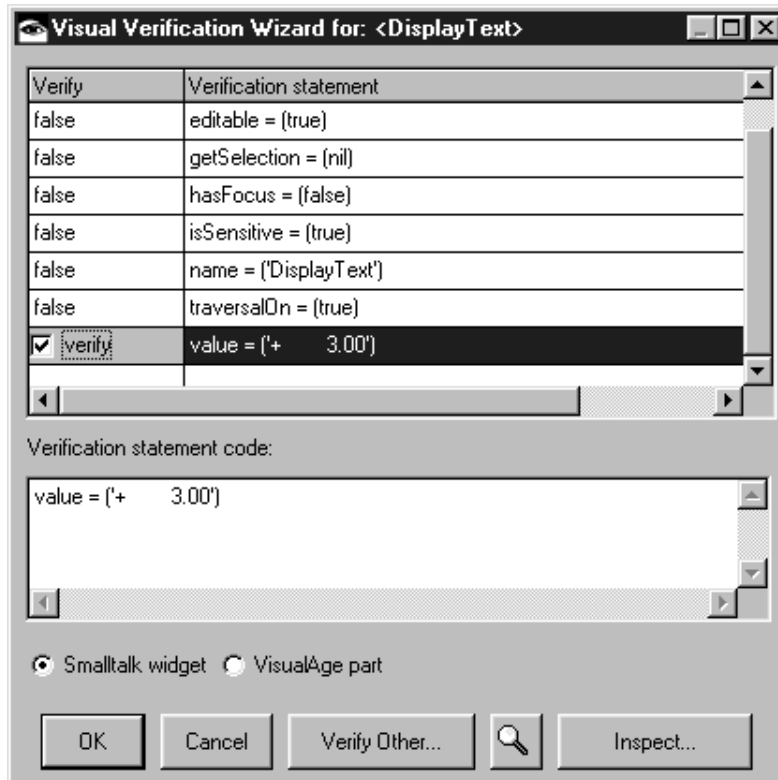
Clearing the recording

You can use the clear recording  button to delete all steps contained by the user interface recording step.

Adding Verification steps

You can use the verify  button or press F2 (**VisualAge only**) to add a verification step. When you do either of the above, the cursor is loaded with a ? graphic in VisualAge and a ⊕ graphic in VisualWorks. Simply move the cursor to a widget that you wish to verify and click on it. A verification window will open showing specific states for the widget that can be verified:

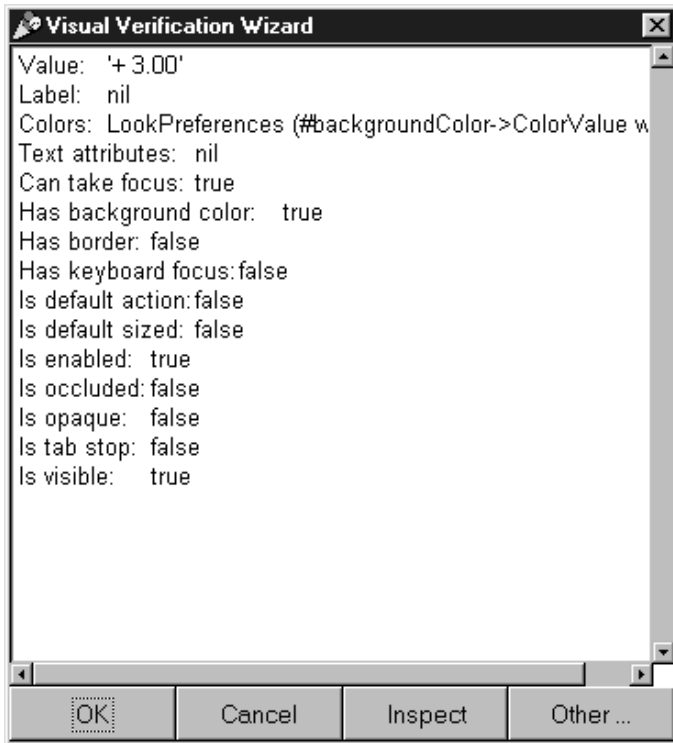
VisualAge:



Set the items in the Verify column that you want to verify to **true** and press OK.

The cursor changes to \Rightarrow when it is over the Test Editor list view. Simply select the step in the Test Editor list view to drop the new verification steps after.

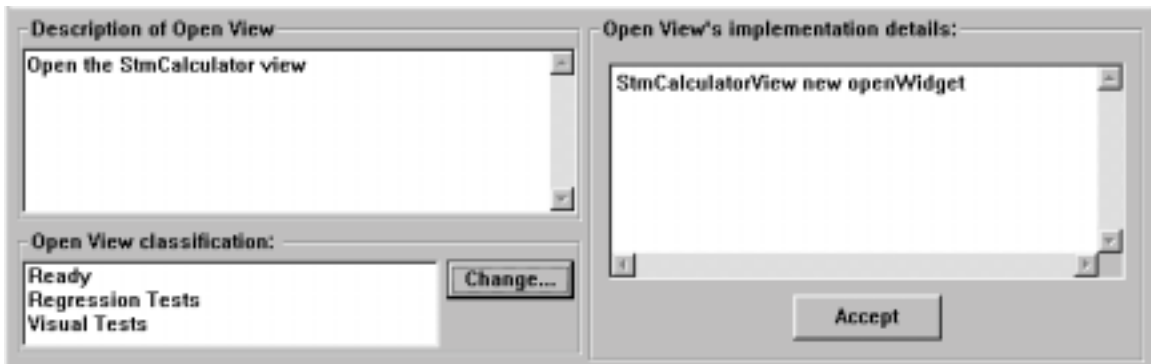
VisualWorks:



Select the items in the multiple selection list to be verified and press OK.

The new verification step is added to the end of the steps contained by the UI recording step.

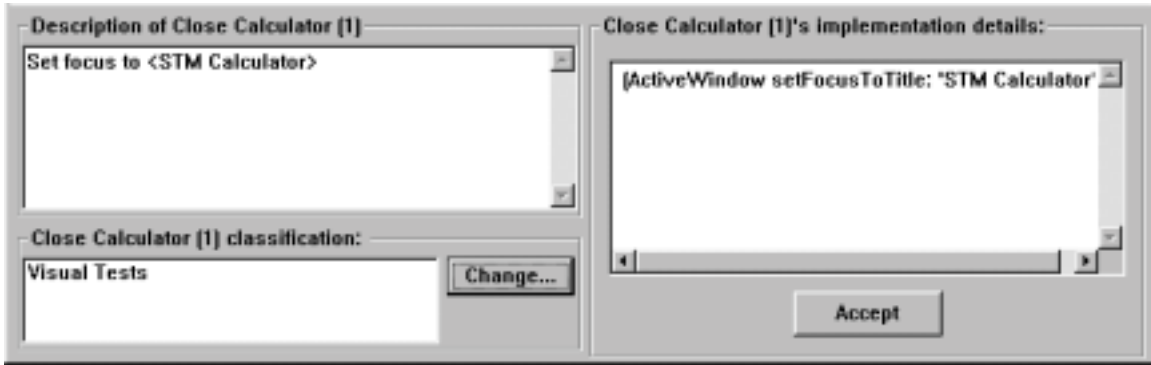
Script Step Details View



A script step is configured by specifying Smalltalk code to execute, in the multi-line entry text area. You must press Accept, or your additions or changes will not be recognized. When you press Accept, the script is compiled as a zero-argument block in the context of the scenario, which is an instance of the owning suite's class. Any references to **self** are references to an instance of the class in which the step is defined.

You can also select text and display, execute or inspect the results of evaluating it, from the text area's popup menu.

User Interface Step Details View



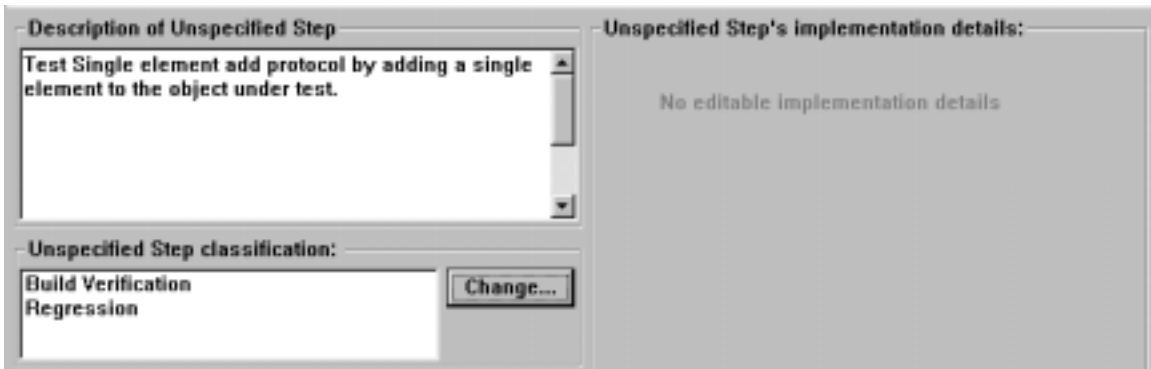
The user interface step is configured by a script that automates some aspect of the user interface of the system under test. Once you have entered the script in the multi-line entry text area, you must press Accept, or your additions or changes will not be recognized. When you press Accept, the script is compiled as a zero-argument block in the context of the scenario, which is an instance of the class in which the step is defined.

You can also select text and display, execute or inspect the results of evaluating it, from the text area's popup menu.

User Interface Verification Step Details View

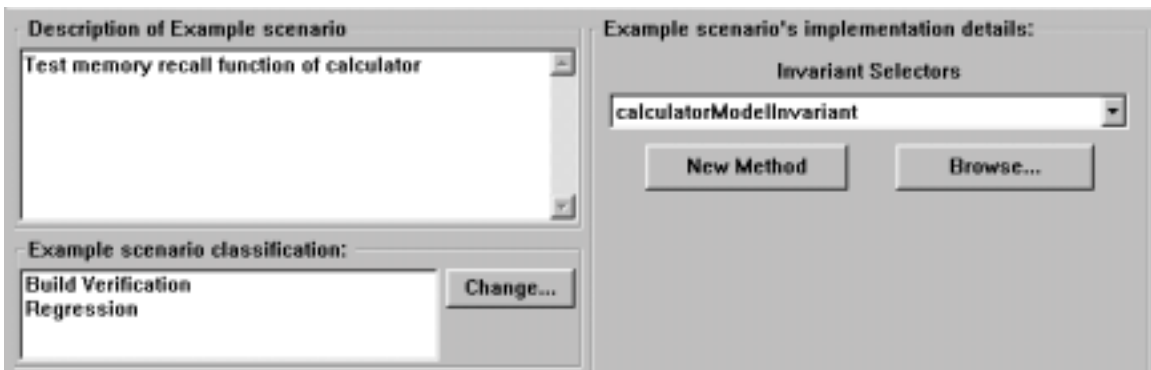
A user interface verification script step is specified in exactly the same way as a user interface script step except that the script must evaluate to a Boolean value.

Unspecified Step Details View



There are no properties to set in the details view for an unspecified step.

Scenario Details View



A scenario is configured in terms of the steps that it contains and its invariant selector. An invariant selector is selected in the same way that an instance method selector is chosen - by selecting an instance method selector from those defined by the class in which the scenario is defined. You can also create a new method or browse the class.

Suite Details View

The screenshot shows a dialog box titled "Suite Details View" for "MySuite1". It is divided into four quadrants:

- Top-Left:** "Description of MySuite1" with a text area containing "Suite of tests to exercise all aspects of the calculator model class."
- Top-Right:** "MySuite1's implementation details:" containing a dropdown menu for "Invariant Selectors" with "calculatorModelInvariant" selected, and two buttons: "New Method" and "Browse...".
- Bottom-Left:** "MySuite1 classification:" with a text area containing "Build Verification" and "Regression", and a "Change..." button.
- Bottom-Right:** "Applications Covered" with a text area containing "SimCalculatorApp" and a "Change..." button.

A suite is configured in terms of the scenarios that it contains, an invariant selector, and a list of methods for the system under test that the suite covers by executing.

The methods under test are specified indirectly through a list of ENVY/developer applications, in systems that use ENVY/developer, or as VisualWorks namespaces.











An invariant selector is selected in the same way that an invariant selector for a scenario is chosen - by selecting an instance method selector from those defined by the class in which the suite is defined. You can also create a new method or browse the class.

You specify the applications or namespaces covered by pressing the Change... button and selecting the application or namespace to include in the list.

Test Editor Tool Bar



The Tool Bar presents shortcuts to commonly used operations. From left to right they are:

Button Name	Button Function
 Run	Run the selected items, using the Quick Runner.
 New Suite	Open the New Suite dialog.
 New Scenario	Create a new, default named, scenario. This will add the scenario after the currently selected scenario or as the last scenario of the currently selected suite. This button is disabled if a step is selected.
 New Step	Open the New Step dialog.
 Find	Open the Find dialog.
 Expand	Expand all the sub-steps of the selected step or steps, if any.
 Contract	Contract all the sub-steps of the selected step or steps, if any.
 Verify	Enter widget verification mode
 Report	Create a report of the selected step or steps.
 Results	Open the Results Browser view.

Test Editor Menus

File

Item Name	Item Function
New	Open a new Smalltalk Workspace
Open...	Open a Smalltalk Workspace
Save Test Cases	Save any suites, scenarios or steps that have been changed since the last save. Changes are saved by generating code into the class in which each step that has been changed is defined. This is know as the <i>suite class</i> .
Save Image...	Save the Smalltalk Image
Save Image As...	Save the Smalltalk Image with a new name
Exit Smalltalk Test Mentor	Exit the Test Editor

Edit

Item Name	Item Function
Find...	Open the Find dialog.
New Suite...	Open the New Suite dialog.
New Scenario	Create a new scenario. You are prompted for a scenario name. The name you enter is also used generate a selector for the scenario when it is saved as code. This will add the scenario after the currently selected scenario or as the last scenario of the currently selected suite. This button is disabled if a step is selected.
New Step...	Open the New Step dialog.
Copy	Create an exact duplicate of the selected item, and place the copy immediately after it. Note: You can not copy a suite.
Delete	Delete all selected suites, scenarios or steps. If you select an item and the item that contains it, the highest level selected item will be deleted. For example, if you select a scenario and one of its steps, the scenario will be deleted, removing all the steps it contains, regardless of the step selected.
Renumber	Locates any steps contained by the selected step whose names contain parenthesized integers, and regenerates new names with the integers based on their respective steps' indices..

View

Item Name	Item Function
Expand	Expand all items beneath the selected item or items, if any.
Contract	Contract all the items of the selected item or items, if any.
Refresh	Refresh the entire list of suites. This is performed by reinstantiating all suites from their repository definitions. If you have unsaved changes, you will be prompted to save any changes.
Filter Suites...	Presents a list of suite classifications to be selected from. If you press OK , only those suites that are selected and suites that do not have any classification set are shown.
Show All Suites...	Shows all suites regardless of classification.

Tools

Item Name	Item Function
Quick Run	Execute the selected items using the Quick Runner
Open Runner...	Open the test runner on the selected steps
Simple Report...	Create a simple text report of the selected step or steps in a window.
RTF Report...	Generates a formatted Rich Text Format (RTF) report, prompting for a file name to store the report into. It also opens a viewer if available on the particular operating system.
Results...	Open the Results Browser view.
Preferences	Open the Preferences Settings view
Open Preferences Workspace...	Opens a workspace containing various settable preferences.
Register Product...	Open the product registration dialog

Help

Item Name	Item Function
About Smalltalk Test Mentor...	Opens a window that describes the Smalltalk Test Mentor Product and how to contact SilverMark, Inc.

Test Editor Operations

Opening the Test Editor

You open the Test Editor by selecting the Smalltalk Test Mentor Editor... item of the Smalltalk Tools menu in the System Transcript, or the Options menu in the VisualAge Organizer.

Creating Steps

Whether you are able to add a new step depends on what kind of step you have selected in the list view, as the following table shows:

Selection	Can Create Suite	Can Create Scenario	Can Create Step
A Suite	Yes	Yes	No
A Scenario	No	Yes	Yes
A Step	No	No	Yes
Multiple Items	No	No	No
No Items	Yes	No	No

Creating a New Suite

Select either the New Suite... menu item or  button on the tool bar. The dialog titled New Suite will open. The New Suite dialog looks like this:

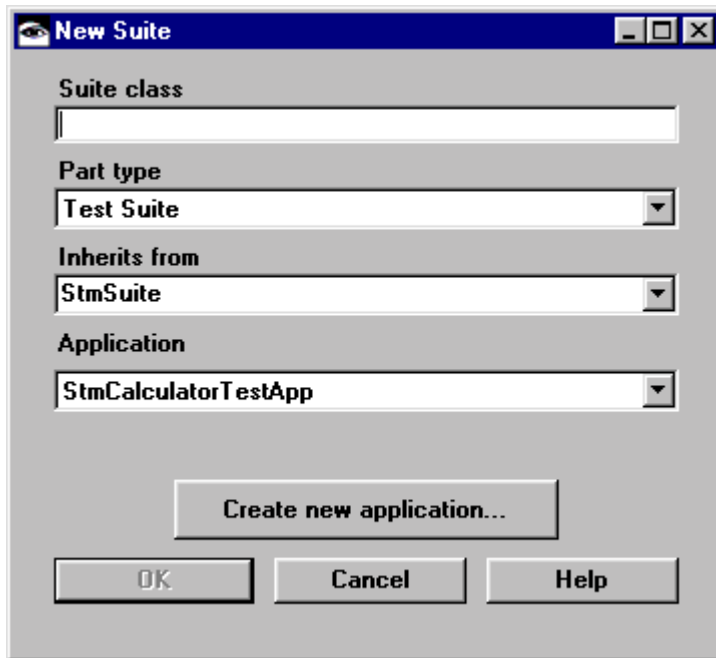


Figure 5 - New suite view with ENVY/developer

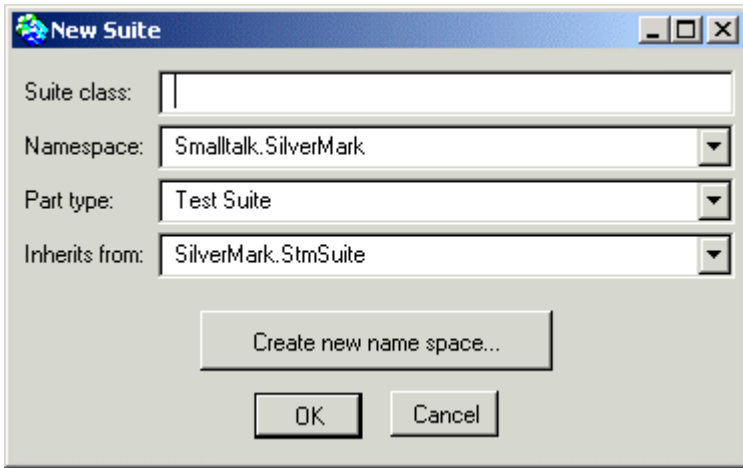



Figure 6 - New suite view in VisualWorks without ENVY/developer


Fill in the Suite Class with the class name of the suite, and which class it inherits from. All suites should have **StmTestSuite** in their inheritance chain. As you create tests you will probably end up creating a test suite hierarchy rooted at **StmTestSuite**. This will be discussed in more detail in *How to Design Tests*. You can select an application or name space, or you can use the [Create New Application...](#) or [Create new name space...](#) button to create a new application or name space for the suite. When you press **OK**, a new class will be created and a new suite with the same name as its class will appear in the Test Editor. You can change the name, description and any other properties of the suite.

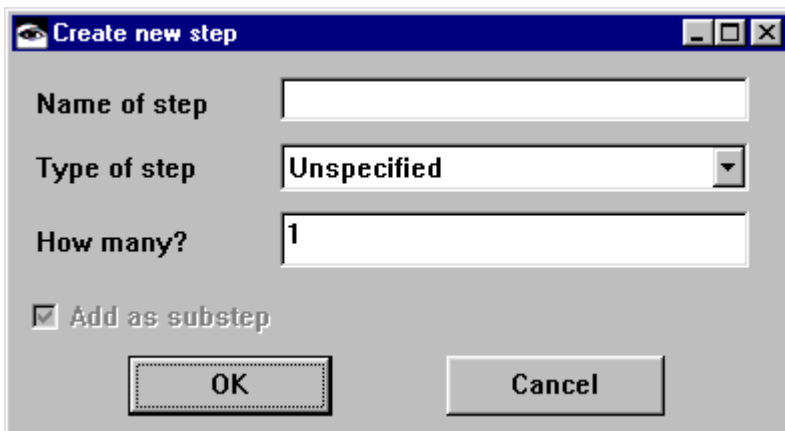
Note: If you exit the editor without saving the suite, the suite's class will still exist in the image and will appear in the editor as a suite the next time you open it.

Creating a New Scenario

Select either the [New Scenario](#) menu item or  button in the tool bar. You are prompted for a name for the scenario. The name is also used as a basis for deriving a selector for the scenario when code for it is generated. See *Specifying a Scenario* in the *Smalltalk Test Mentor Programmer's Guide* section for details about this. If a suite was selected, a new scenario appears after the last scenario of the selected suite. If a scenario was selected, a new scenario appears after the selected scenario. You can change the name, description and any other properties of the scenario.

Creating a New Step

You can add a new step to either a scenario or a collection step. You add a new step by selecting either the [New Step...](#) menu item or  button in the tool bar. When you do this the [Create New Step dialog](#) appears:



This requires the following parameters to be filled in:

Field	Value
Name of Step	The name of the step. It can be any string, although it is best to make it descriptive but brief .
Type of step	You can select any of the step types described in Test Step Implementation, in this drop-down list. You can always leave Unspecified selected and change it later from within the editor.
How Many?	You can specify the number of new steps of the selected type to add. Each step is created with the above name and the step's index appended to it.
Add as sub-step	If the selected step is a collection-type step, this field will be enabled. If you check the checkbox, the new steps will be added as a child of the step, rather than as peers immediately following.

Modifying a Step's Properties

You modify a step's properties by selecting the step and editing the values as they are presented in the user interface.

VisualAge:

Some step properties are presented as editable cells in the step list columns. Click on a cell to begin editing its value. Click elsewhere to accept the value.

VisualWorks:

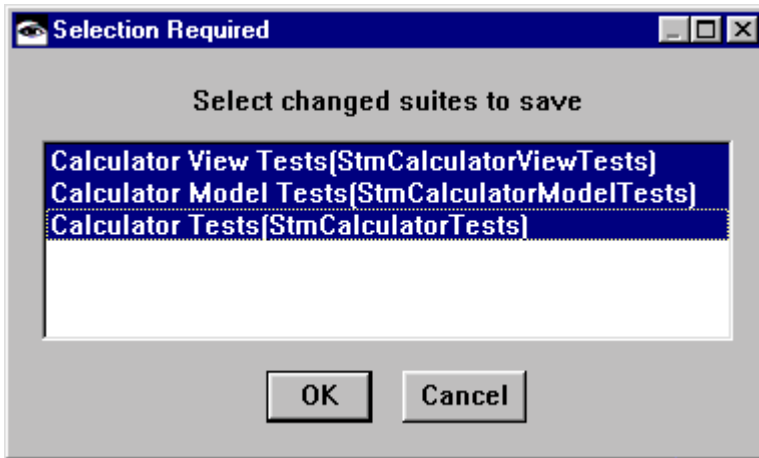
In VisualWorks some step properties are presented to the right of the list view for the selected step. You must tab out of a property's control to accept a new value.

In the case of some values, you may need to change focus to a different field for the value to be accepted. The name and description fields in the details view are one example where you would need to do this.

Saving Changes

When you add, delete or modify a step, the Test Editor marks it or its parent step as having been changed. If you wish to keep the changes, you need to use the **Save Test Cases** menu item to generate code, otherwise you will lose the changes when you leave the editor. The only exception, as mentioned earlier, is that once you create a test suite, its class remains in the image until you explicitly delete it.

When you request to save changes, you are presented with a list of suites that contain changes to save:



In the above example, changes were detected in the three suites listed. Deselect the suites to **not** save and press **OK** to generate code for them. When the Smalltalk Test Mentor saves tests, it makes a best effort to generate a minimal amount of code for them. The code generated is discussed in detail in the section titled *Smalltalk Test Mentor Programmer's Guide*.

Recording User Interface Interactions


You can record user interface interactions into a user interface recording step in one of two ways:

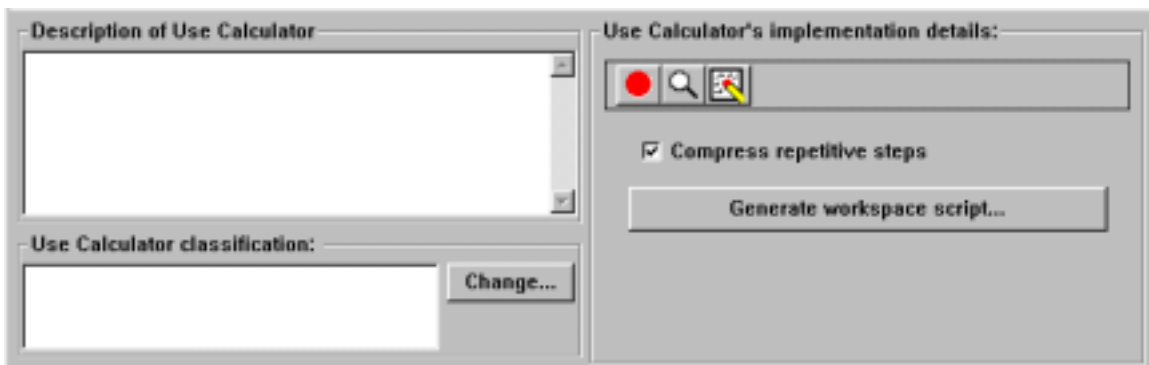
- By entering user interface *recording* mode when a user interface recording step is selected from the Test Editor.

- By entering user interface *capture* mode as a result of launching the Quick Runner for one or more empty user interface recording steps.

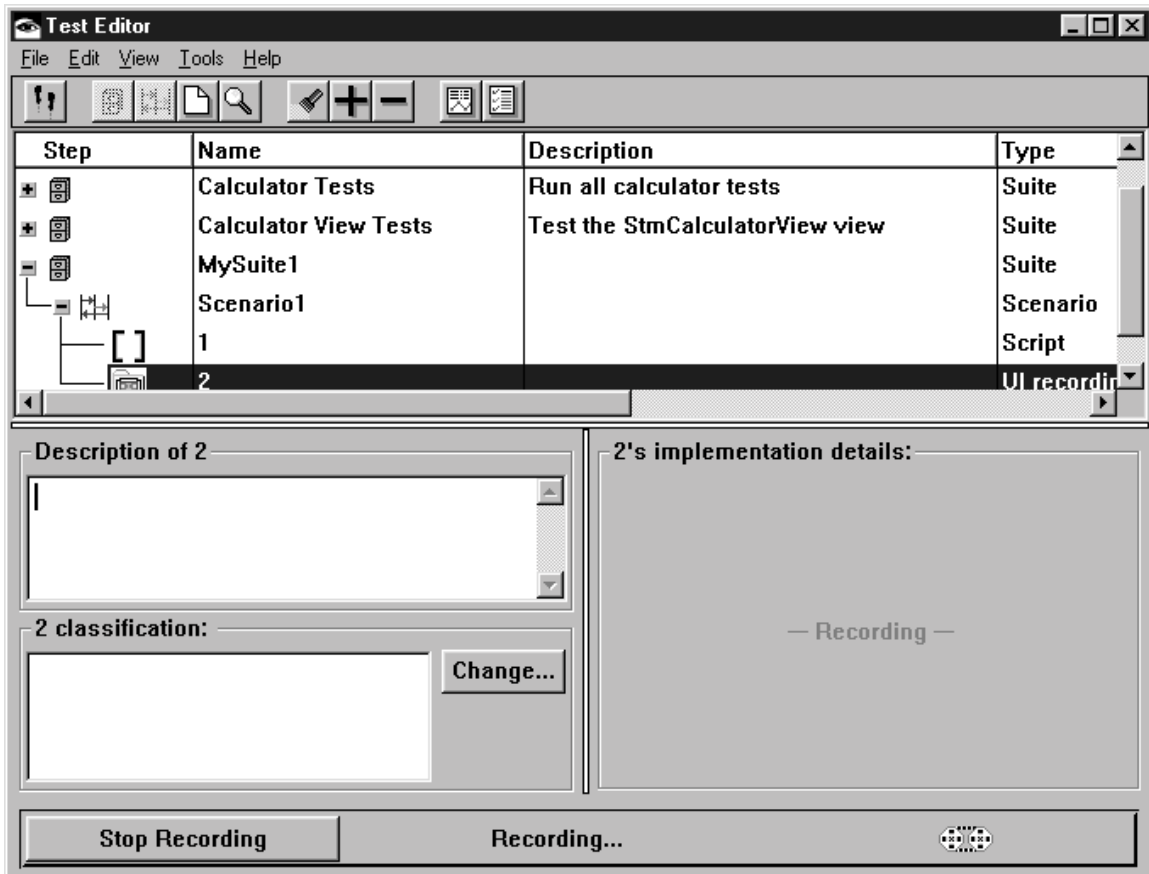
User interface recording mode

User interface recording mode is simply the mode of recording user interface interactions for a particular user interface recording step. You enter user interface recording mode by selecting a user interface

recording step and pressing the Record  button in the user interface recording step details view:



This starts user interface recording mode. When in user interface recording mode, the Test Editor adds a recording status area to the bottom of the details view:



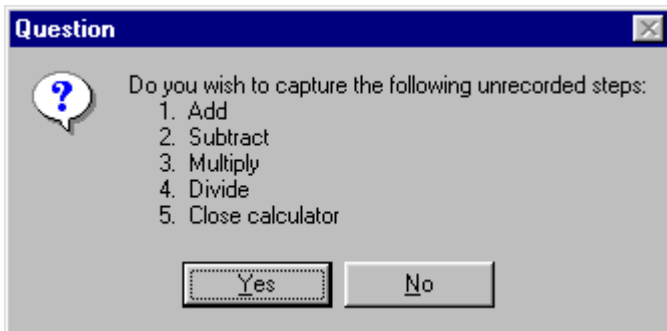
While the Test Editor is in this mode, user interface interactions for any view opened after entering user interface recording mode, are recorded and added as steps to the user interface recording step.

Note: Make sure that you open the view or views to be recorded **after** you enter user interface recording mode. This is because some widgets are only watched once they are created, while the Test Editor is in user interface recording mode.

When you are done recording, press the Stop Recording button. The recorded user interface interactions are translated into user interface steps and added after the last step contained by the user interface recording step, if any. The user interface recording status area is also removed.

User interface capture mode

User interface capture mode is entered whenever the Quick Runner detects one or more empty user interface recording steps. When you run the Quick Runner on an item that contains one or more empty user interface recording steps, you are prompted to enter user interface capture mode:



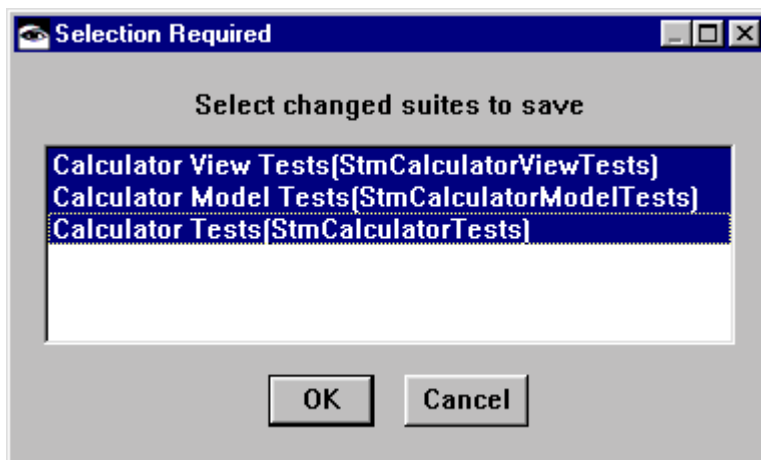
If you answer No, all the steps, including the empty user interface steps, are executed as they normally would. If you answer Yes, the Quick Runner executes each step as it normally would until it reaches an empty user interface recording step to capture, at which time the Quick Runner enters user interface recording mode for the empty user interface recording step, as described above. When you press the Stop Recording button, the Stop Recording button changes to Resume Running instead of the recording status area going away entirely. At this point, the Quick Runner is in a paused state. You can edit the recorded steps or add user interface recording steps.

When you press Resume Running, the Quick Runner continues executing steps until the next empty user interface recording step is encountered, at which point, it enters user interface recording mode for that step, and so on.

You can terminate capture mode when the Quick Runner is paused by pressing the Terminate button in the user interface recording status area.

Closing the Test Editor

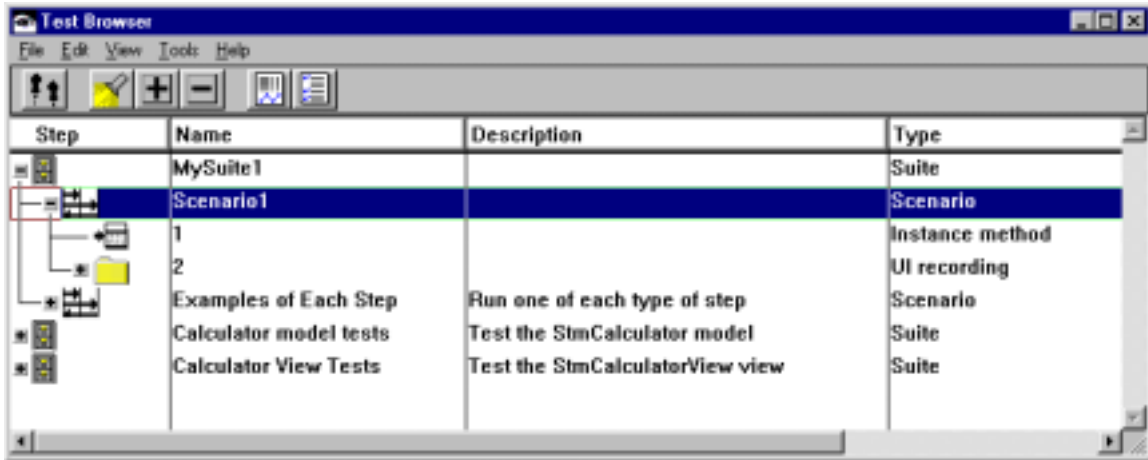
You close the Test Editor by using the standard window closing control appropriate to your operating system, or by selecting the Exit Smalltalk Test Mentor item in the File menu.. If you have made changes to any test steps you will be prompted to save them. If you answer Yes to the prompt, you will be prompted again to select which changed suites to save:



When you press OK, Smalltalk code is generated into the classes for the suites selected in the list or none if you press Cancel. If you answer No to the prompt to save, no changes are saved.

The Test Browser

The Test Browser is the best view to use for managing test cases where you do not wish to create or modify tests. It is also the only view that you can use to manage tests in the Smalltalk Test Mentor - *runner* product. The Test Browser operates in much the same way as the Test Editor. You can also open it by selecting Smalltalk Test Mentor Browser... from the System Transcript Smalltalk Tools menu. This view is composed of a single section equivalent to the details view in the Test Editor. This view is basically the Test Editor view with the exception that the columns are read-only, there is no details view, and all edit related menu items and Tool Bar items are absent.



See *The Test Editor* for a description of the columns in the Test Browser's list view.

Opening the Test Browser

You open the Test Browser by selecting the SilverMark's Test Mentor Browser... item of the Smalltalk Tools menu in the System Transcript.

Closing the Test Browser

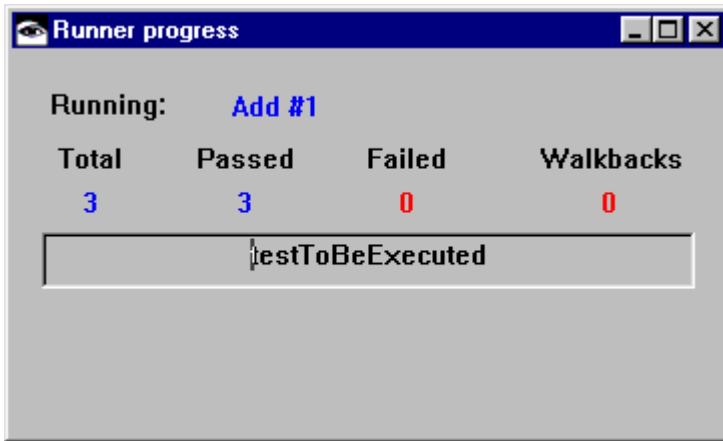
You close the Test Browser by using the standard window closing control appropriate to your operating system, or by selecting the Exit Smalltalk Test Mentor item in the File menu.

The Quick Runner

The Quick Runner provides a fast way to run one or more selected steps. You launch it by selecting one or more steps from either the Test Editor, Test Browser, or Test Runner, and then pressing the Quick Run



button, or the Quick Run item in the Tools menu, or the Quick Run list view popup menu item. The Quick Runner displays a progress window while the selected steps are executing, opens a Results Browser to display the results of execution, and then closes. The Quick Runner progress is displayed in a window like this:



While the Quick Runner is executing, you can not control execution, except to terminate it by closing the above window.

The progress items shown are:

Count Item	Description
Total	The total number of steps run so far
Passed	The number of steps run so far that have passed.
Failed	The number of steps run so far that have failed for any reason.
Walkbacks	The number of exceptions trapped by the test runner. When a step raises an exception that is trapped by the test runner, the following step may or may not be executed depending on the execution controls (precondition, abort-on-fail) of the offending step or its parent.
Running	The name of the suite, scenario or step that is executing.
State	The current state of execution. On a reasonably fast processor, this will usually change too fast to clearly see.

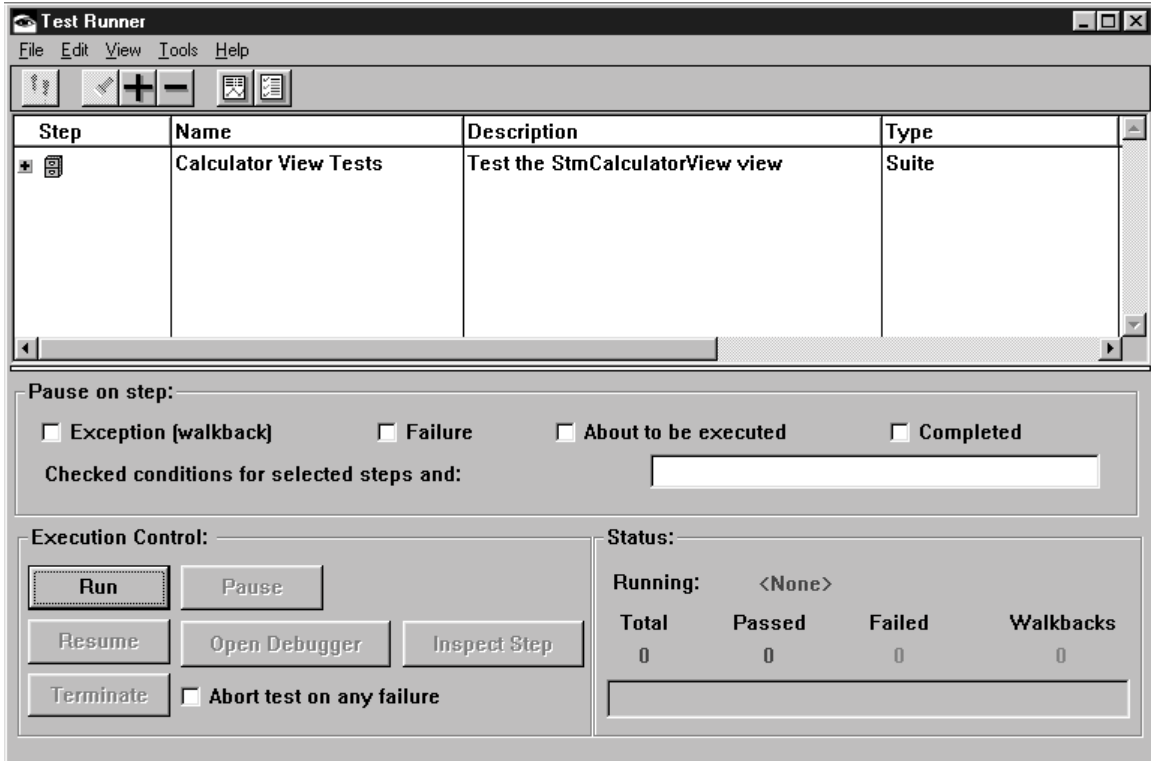
Note 1: If you select a step and a step that contains it, the selection of the contained step is ignored and the entire containing step is executed. For example, if you select a collection step and one of the steps that it contains, the collection step and all the steps that it contains are executed, ignoring the selection of the single step within it.

Note 2: The order in which you select items determines the order in which they are executed.

Note 3: If you launch the Quick Runner from a Test Editor, and any of the selected steps are empty user interface recording steps, you will be prompted to capture user interface interactions for those steps. See *Creating View Tests* and *How to record user interface interactions* for more details about this.

The Test Runner

The Test Runner allows you to execute test steps. You can open a Test Runner by selecting items in the list view to be executed, and selecting the **Run** button or the **Open Runner** item in the **Tools** menu, or the **Open Runner** list view popup menu item. The list view shows which steps are to be run. The details view contains controls that affect the execution of the test. The details view for the Test Runner is called the *Execution View*.



Test Runner List View

The Test Runner runs all items that appear in the list view, as well as any items that they contain, in the order in which they appear. The order of items is determined by the order in which the items were selected in the view from which the Test Runner was launched.

Like the list view columns in the Test Browser, the columns in the Test Runner list view are read-only. See *The Test Editor* for a description of the columns in the list view.

Test Runner Execution View

You use the Test Runner execution view to control the execution of the items that appear in the Test Runner list view. You can start, pause or terminate a test by using the controls in the execution view.

The Execution view is divided into two main areas delineated by group boxes. The upper area, labeled Pause on step, presents controls to allow you to pause on various conditions:

Condition label	Description
Exception	Checking this causes the Test Runner to pause when the execution of a step has caused an exception to be raised.

Failure	Checking this causes the Test Runner to pause when a test fails.
About to be executed	Checking this causes the Test Runner to pause when a step is about to be executed.
Completed	Checking this causes the Test Runner to pause when a step has completed.

The conditions checked apply to either all executed steps or specific steps. When no items in the list view are selected, the checked conditions are applied to each step that is executed. When one or more items are selected, the checked conditions apply to only those items. You can also directly enter the name of a step to apply the checked conditions to in addition to those selected. You do this by entering an item name in the text field. Next to the Check conditions for selected steps and: label.

The lower area, labeled Execution Control, presents the controls for starting a test, resuming a paused test or terminating a test. The controls are:

Option	Function
Run	Execute the test cases with all of the selected options.
Resume	Continue executing a suspended test case.
Terminate	Stop the execution of the test cases. You will not be able to resume the test if you terminate it.
Open Debugger	When a test execution is suspended because of an exception, you can use this option to open the Smalltalk debugger on the process on which the steps are executing. Note: Once you open the debugger, the <u>Execution Control</u> buttons become disabled. You can only resume or terminate the test by pressing the <u>Resume</u> button in the debugger or by closing the debugger, respectively.
Inspect Step	Open a Smalltalk Inspector on the currently paused step. See the <i>Smalltalk Test Mentor Programmer's Guide</i> for discussion on what aspects of an instance of a step are interesting
Abort on any test failure	Check this option to force the test to prematurely terminate on the failure of any step.

Note: Terminate is not preemptive. The test is terminated after the currently executing step completes.







On the right side of the lower area is a display of execution progress. The values displayed are:

Count Item	Description
Total	The total number of steps run so far.
Passed	The number of steps run so far that have passed.
Failed	The number of steps run so far that have failed for any reason.
Walkbacks	The number of exceptions trapped by the test runner. When a step raises an exception that is trapped by the test runner, the following step may or may not be executed depending on the execution controls (precondition, abort-on-fail) of the offending step or its parent.
Running	The name of the suite, scenario or step that is executing.
State	The current state of execution. On a reasonably fast processor, this will usually change too fast to clearly see.

Test Runner Tool Bar



The Tool Bar presents shortcuts to commonly used operations. From left to right they are:

Button Name	Button Function
 Run	Run the selected items, using the Quick Runner.
 Find	Open the Find dialog.
 Expand	Expand all the sub-steps of the selected step or steps, if any.
 Contract	Contract all the sub-steps of the selected step or steps, if any.
 Report	Create a report of the selected step or steps.
 Results	Open the Results Browser view.

Test Runner Menus

File

Item Name	Item Function
New	Open a new Smalltalk Workspace
Open...	Open a Smalltalk Workspace
Save Image...	Save the Smalltalk Image
Save Image As...	Save the Smalltalk Image with a new name
Exit Smalltalk Test Mentor	Exit the Test Editor

Edit

Item Name	Item Function
Find...	Open the Find dialog.

View

Item Name	Item Function
Expand	Expand all the sub-steps of the selected step or steps, if any.
Contract	Contract all the sub-steps of the selected step or steps, if any.

Tools

Item Name	Item Function
Quick Run	Execute the selected items using the Quick Runner
Open Runner....	Open the test runner on the selected steps
Report	Create a report of the selected step or steps.
Results...	Open the Results Browser view.
Preferences	Open the Preferences Settings view

Help

Item Name	Item Function
About Smalltalk Test Mentor...	Opens a window that describes the Smalltalk Test Mentor Product and how to contact SilverMark, Inc.

Test Runner Operations

Running a Simple Test

If you have opened the Test Runner view from another view with steps selected, you should see those steps in the Test Runner list view.

Note: If in the view from which you launched the Test Runner, you had selected both a step and any of its sub-steps, the sub-step selection is ignored.

To run the selected steps, simply press the Run button in the execution view. As each step is executed, you should see its name displayed next to the Running: label. Also, you should see the count of step results incremented after each step is executed. Once the selected steps and their sub-steps have completed execution, The Test Results Browser is automatically opened. The Test Results Browser shows the results for each of the steps that were executed. The Results Browser will be covered in the section entitled

The Test Results Browser.

Pausing the Execution of a Test

If you wish to debug a failing test, or just want to execute a test in a stepwise manner, you need to use the controls in the execution view. The main concept to understand is that the Test Runner pauses on any condition you select for *any step* unless you specify a particular step or steps to pause on. There are three ways to make the Test Runner pause on a step:

By selecting items in the test runner list view.

By Entering a step name in the Checked conditions for selected steps and: field

By performing both of the above

Once you have performed either of the above actions, the Test Runner will pause on conditions only when they are signaled, by executing the steps indicated by the above mechanism. Otherwise, meeting the checked conditions for any step will cause the Test Runner to pause.

Consider the following scenarios and their outcomes:

Scenario	Outcome
No pause conditions are checked No steps are selected and no step name is entered in the <u>Checked conditions for selected steps and:</u> field	All the steps are executed without pause.
One or more pause conditions are selected No steps are selected	Each step is paused on each of the selected pause conditions.
No pause conditions are selected One or more steps are selected and a step name is entered in the <u>Checked conditions for selected steps and:</u> field.	All the steps are executed without pause
One or more pause conditions are selected One or more steps are selected and a step name is entered in the <u>Checked conditions for selected steps and:</u> field.	Only the selected steps, as well as any steps with the same name as the string entered in the <u>Checked conditions for selected steps and:</u> field, are paused on each of the pause conditions.

What You Can do With a Paused Step

The main reason to pause a test on a particular condition is to be able to view the state of the test or system under test once that condition has occurred. To that end, the Test Runner view provides controls to allow you to open the Smalltalk debugger on a process suspended due to an exception, as well as to directly inspect the instance of the test case that you are running.

To open the Smalltalk debugger when a step is paused, simply click on the Open Debugger push button.

Note: Once you open the debugger, the Execution Control buttons become disabled. You can only resume or terminate the test by pressing the Resume button in the debugger or by closing the debugger, respectively.

When you do this, the Smalltalk debugger is opened on the suspended process. Usually this is only useful if you want to trace the execution of your step, or are trying to debug an exception condition. When you pause on a step that is about to be executed, you will probably have to step over one or more message sends in order to reach the point at which the heart of the step is executed. This is always in a subclass implementation of `StmContext>>#basicPerformTest`.

To inspect the instance of the test step, click on the Inspect Test Step push button. Different test steps will have different instance variables of interest to you. See the *Smalltalk Test Mentor Programmer's Guide* for discussion on what aspects of an instance of a step are useful to view from an inspector.

Finding the Cause of an Exception

When the Test Runner has paused on an exception condition, click on the Open Debugger push button. A Smalltalk Debugger is opened with the first stack entry for the executed step selected. You can move up the stack to locate the source of the exception. The actual location will vary depending on the exception, and type of step. To resume operation, click on the Resume push button. When you resume, the exception is logged for the offending step and the next step or steps are executed.

Note 1: *It is often simpler to check Open debugger on exceptions in the Misc. tab of the Test Mentor Preferences notebook, rather than setting the test runner view to pause on exceptions. In general you will perform less stack navigation in the debugger.*

Note 2: *Once you have opened the Smalltalk debugger, make sure that you do not close it or the suspended process will be terminated.*

Note 3: Any time the debugger is opened during a test, playback of prompters implemented by the operating system is disabled for the rest of the test. See the section on recording and playing back prompters and suspended views for more information on this.

Aborting a Test on Any Failed Step

In some cases you may want to only execute a test's steps up to the first failure. For example, if you wanted to run a lengthy test with expectations of no failures, you might want the test to abort on the first failure. You can do this two ways. In cases where you *always* want your test to behave this way, you would set every suite, scenario or collection step (that is, any step that could be a parent of other steps) to abort on failure (of its sub-steps). In cases where you want to occasionally abort a test on a step's failure, you can use the Abort test on any failure check box in the Test Runner. If any steps fail during the execution of your test, the test is terminated and the results are displayed by the Test Results Browser, as they would have been had the test continued to normal completion, but without results for the steps following the failed step.

Caution: *If your test has steps to clean up after itself, those steps may not be run.*

The Test Results Browser

The Test Results Browser allows you to view and analyze the results of test execution. It, like most other Smalltalk Test Mentor views, is divided into an upper, list view, and a lower, details view. The list view shows which steps have been executed and the results of having done so. The details view shows details of the results from executing individual steps, or a statistical comparison if multiple steps are selected.

The screenshot shows the 'Test Results' window with a menu bar (File, Edit, View, Tools, Archive, Help) and a toolbar. The main area is a table with columns: Step, Pass, Name, Description, Execution time, and Type. The 'Pass' column contains traffic light icons. The 'Step' column shows a tree structure. Below the main table is a summary table with columns: Name, Run, Pass, Fail, Time, Covered (%), Testable (%), Methods (#), and Testable (#). At the bottom is a 'Metric' table with columns: Metric, Max, Min, Range, Mean, Median, and Std Dev.

Step	Pass	Name	Description	Execution time	Type
Calculator View Tests		Calculator View Tests	Test the StmCalculatorView view	2,653	Summary
Single calculation		Single calculation	Perform a single, simple calculation and verify the result	1,133	Scenario
Open View		Open View	Open the calculator view	604	Initial
Use Calculator		Use Calculator		355	UI
Use Calculator (1)		Use Calculator (1)	Set focus to <STM Calculator>	37	UI
Use Calculator (2)		Use Calculator (2)	Click on <4>	66	UI
Use Calculator (3)		Use Calculator (3)	Click on <5>	16	UI

Name	Run	Pass	Fail	Time	Covered (%)	Testable (%)	Methods (#)	Testable (#)
Calculator View Test	81	81	0	2,653	58.97	82.14	39	28
Calculator View Test	81	81	0	2,900	53.85	75.00	39	28

Metric	Max	Min	Range	Mean	Median	Std Dev
Steps run	81	81	0	81.00	81	0.00
Steps passed	81	81	0	81.00	81	0.00
Steps failed	0	0	0	0.00	0	0.00
Execution time	2,900	2,653	247	2,776.50	2,653	174.66
% Testable covered	82	75	7	78.57	82	5.05
% covered	59	54	5	56.41	59	1.63

The List View

The list view, like that of the previously described views, is a hierarchical collection of items. Each item in the Results Browser's list view corresponds to the results of executing a test step.

In VisualWorks, items are presented as Strings in the following format:

[Pass or Fail] Name (Description)

In VisualAge, item properties are presented as cells in a their respective columns. The columns shown are:

Column Name	Description
Step	The step column shows the icon and containment relationship between the steps and their parents.
Pass	The pass column contains icons to show the success or failure of the column's respective steps. There are three possible icons that may be shown, all of which are of traffic lights: Green light - Step execution success Yellow light - Step execution failure. Red light - Step execution failure from an exception.
Name	The name column contains the names of the steps whose respective results are displayed

	results are displayed.
Description	The description column contains the descriptions of the steps whose respective results are displayed.
Execution time	The execution time column contains the times to execute the steps whose respective results are displayed. Execution time is measured in milliseconds. An execution time of 0 indicates that the time was too small for the Smalltalk timer to measure. Execution times for steps that truly represent collections of other steps are shown as the sum of the steps they contain. For example, a suite will always have as an execution time the total time to execute all its scenarios.
Type	The type column contains a description of the step implementation types. Values are the same as those described under The Test Editor.
Time Stamp	The time stamp column contains the times at which the steps whose respective results are displayed were executed.
Iteration	The execution iteration of the steps that created the results.
Iterations	The number of expected iterations for the each step.
Abort on Fail	The abort on fail column indicates precondition property of its respective steps.
Precondition	The precondition column indicates precondition property of its respective steps.

Note: When looking at results, you might want to think of the iteration and iterations values in terms of *iteration x out of y iterations for the results' step.*

Note: In VisualWorks, the name, step and pass icons, as well as execution time are shown in the list.

The Details View

The details view has three selection modes:

Selection	Details view configuration
No items selected	Nothing is shown
One item selected	The details view shows a summary of various metrics of the selected step, as well as the value returned by the selected step when it was executed. For example, if the instance method executed by an instance method step returned nil, you would see 'nil' in the details view.
Multiple items selected	The details view shows a statistical comparison of the above metrics for each of the selected steps.






The metrics for single selection are shown in a table with the following columns:

Column name	Description
Name	The name of the step whose results are shown
Run	The number of steps run as a result of executing the selected step. This typically has a value of 1 for steps that do not contain other steps. The value for a step that contains other steps (like suites, scenarios and collection steps) is the sum of all the values in the Run column for all sub-steps and their sub-steps, if any.
Pass	The number of steps that did not fail, not counting steps that contain other steps.
Fail	The number of steps that failed, not counting steps that contain other steps.
Time	The execution times of the selected steps.
Testable %	The percent of testable methods in the application(s) under test that were covered by the test. There are certain conditions under which a method's coverage can not be tested. Primitives, for example, can not have their execution measured.
Covered %	The percent of all methods in the application(s) under test that were covered by the test, whether they are testable for coverage or not. This number will typically be lower than the above number.
Testable #	The number of methods in the application under test that are testable.
Methods #	The number of methods in the application under test.

Test Results Tool Bar



The Tool Bar presents shortcuts to commonly used operations. From left to right they are:

Button Name	Button Function
 Find	Open the Find dialog.
 Expand	Expand all the results contained by the selected result or results, if any.
 Contract	Contract all the results contained by the selected result or results.
 Report	Create a report of the selected results.
 Results	Open another Results Browser view.

Test Results Menus

File

Item Name	Item Function
New	Open a new Smalltalk Workspace
Open...	Open a Smalltalk Workspace
Export Results	Export the selected results as <i>comma separated variables</i> . This feature allows you to write a file containing the selected items in a format that can be read by most spreadsheet programs. When you use this, you will be prompted for a target file.
Save Image...	Save the Smalltalk Image
Save Image As...	Save the Smalltalk Image with a new name
Exit Smalltalk Test Mentor	Exit the Test Editor

Exporting Results

You can export results as comma separated values readable by popular programs like Excel™. To do this, simply select any step or steps and then select the **Export results...** menu item in the **File** drop-down menu. You will be prompted for a file to create. You can then use the appropriate import facilities for comma separated values in your application of choice.

Edit

Item Name	Item Function
Find...	Open the Find dialog.

View

Item Name	Item Function
Expand	Expand all the results contained by the selected result or results, if any.
Contract	Contract all the results contained by the selected result or results.

Tools

Item Name	Item Function
Run	Open a <u>new</u> test runner on the selected steps.
Report	Create a report of the selected step or steps.
Differences...	Open a Differences Browser on the selected steps. This is only enabled when <u>two</u> items are selected.
Coverage Analysis...	Open a Method Coverage Browser on the selected step. This is only enabled when <u>one</u> item is selected.
Results	Open a Results Browser

Archive

Item Name	Item Function
Store results into archive (ENVY/developer only)	Store the currently selected results into the results archive. You can only store results for a suite and the suite's class must be a version. This is used to guarantee that test results are associated with a stable, unchanging edition of the test class that created them.
Load results from archive (ENVY/developer only)	This loads stored results from the archive. When you select this item you are presented with a list of stored test results. The results are displayed in terms of the time they were stored and class and version of the test class they are stored under. You can only load results created by the same edition of a suite's class that is currently loaded in the image.
Store results into file	Store the currently selected results into a file. You can only store results for a suite and the suite's class must be a version. This is used to guarantee that test results are associated with a stable, unchanging edition of the test class that created them.

Load results from file	This loads stored results from a file as chosen from a file dialog. You usually use this option to load results generated by a packaged image.
------------------------	--

Note 1: If you load results from a file and the results were created by an edition of a test suite's class that is substantially different from the edition present in the image, you will get unpredictable results

Note 2: You can override both the check for storing a versioned suite class and the check for loading the results for a suite generated by the same version of the suite class that is currently loaded in the image, through preferences settings. The reason for overriding these checks is if you wish to compare the results of a suite with slight (non-structural) variations. If you do this, be very careful. Structural variations (different numbers of steps, or different shaped hierarchies) may cause inconsistent or incorrect results to be shown.

Archiving Results

You can store test results in an archive for later analysis. When you want to view them again, or wish to compare them to other results, you can retrieve the stored results. You store and retrieve results from the Test Results Browser.

When you store results, they are associated in the archive with the version of the suite class whose execution produced them. Because of this, results archival imposes the following two restrictions:

You can only store the results of having executed a suite.

The suite whose results are being stored must be a **versioned class**.

By imposing restriction #2, you can always be certain which edition of your test was responsible for creating a particular set of test results.

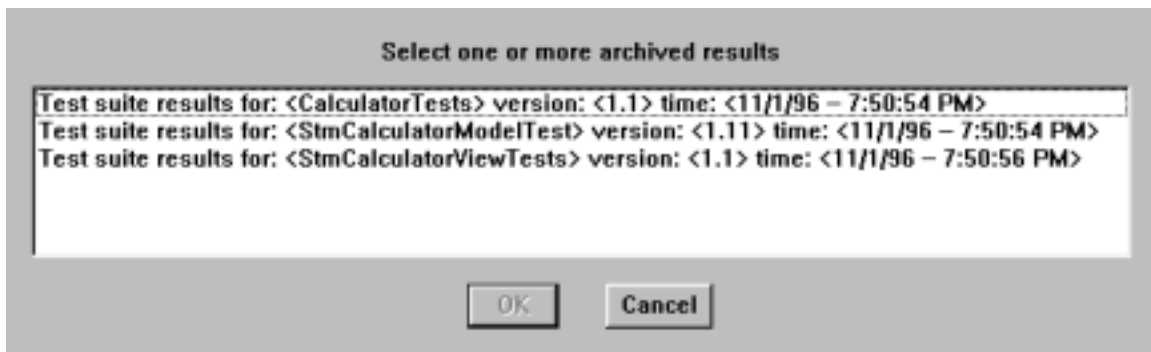
Note: Although we do not recommend doing so, you can use the preferences to disable this restriction.

Storing Results in the Archive

From the Test Results Browser, select the suite or suites to be stored, then select the Store results into archive menu item under the Archive drop-down menu. The results will be associated with the correct version of your suite's class automatically.

Retrieving Results from the Archive

From the Test Results Browser, select the Load results from archive menu item under the Archive drop-down menu. When you do this, you will be presented with a list of previously stored results:



Select the suites you are interested in loading and press OK. The results you select are loaded and added to those displayed in the Test Results Browser at the end of the list.

Help

Item Name	Item Function
About Smalltalk Test Mentor...	Opens a window that describes the Smalltalk Test Mentor Product and how to contact SilverMark, Inc.

The Test Results Differences Browser

The Test Results Differences Browser allows you to compare results to find the differences between them. This is especially useful for finding the differences between two executions of a suite for rapid pinpointing of new or fixed problems.

You launch the Test Results Differences Browser by selecting two steps to compare in the Test Results Browser and then selecting the Differences... item in the Tools menu or the Differences... item in the list view popup. When you do this, you are prompted to enter a maximum allowable percent difference between execution times. This is used to determine if execution times are within an allowable percent difference from one another.

The Test Results Differences Browser calculates differences by starting with the two selected results items and recursively comparing those items and their nested results items down the hierarchy (tree) of results, producing a hierarchy of comparison results of the same general structure.

In the case where the hierarchies of the selected results items do not match, the comparison is performed on as many items as possible.

Like many of the other Smalltalk Test Mentor browsers, the view is divided into an upper, list view and a lower, details view. The items in the upper view represent individual comparisons in a hierarchy that reflects that of the items being compared. The details view shows the reasons, if any, that the compared items did not match.

Match	Name	Time Stamp	Name	Time Stamp
+	Calculator View Tests	7/5/99 - 11:32:28 P	Calculator View Tests	7/7/99 - 8:27:51 AM
+	Single calculation	7/5/99 - 11:32:28 P	Single calculation	7/7/99 - 8:27:52 AM
+	Simple Calculations	7/5/99 - 11:32:33 P	Simple Calculations	7/7/99 - 8:27:56 AM
+	Open View	7/5/99 - 11:32:33 P	Open View	7/7/99 - 8:27:56 AM
+	Addition	7/5/99 - 11:32:34 P	Addition	7/7/99 - 8:27:57 AM
+	Subtraction	7/5/99 - 11:32:36 P	Subtraction	7/7/99 - 8:27:59 AM
+	Multiplication	7/5/99 - 11:32:38 P	Multiplication	7/7/99 - 8:28:01 AM

Execution times within 10% mismatch: <236> and <275>

Test Results Differences Browser List View

The list view, like that of the previously described views, is a hierarchical collection of items. Each item in the Results Differences Browser's list view corresponds to the results of comparing two results.

In VisualWorks, each item in the list view indicates the items compared as a String in the form:

```
Name1 [ date1 - time1 ] *WITH* Name2 [ date2 - time2 ]
```

In VisualAge the results are arranged in columns. The columns shown are:

Column Name	Description
Match	The match column contains icons to show the success or failure of the respective comparison. There are two possible icons that may be shown, both of which are of traffic lights: Green light - Step execution success Red light - Step execution failure from an exception.
Name	The name column contains the names of the first selected comparison items.
Time Stamp	The time stamp column contains the times at which the steps for the first comparison items were executed.
Name	The name column contains the names of the second selected comparison items.
Time Stamp	The time stamp column contains the times at which the steps for the second comparison items were executed.

Test Results Differences Browser Details View






The details view for the Test Results lists the reason or reasons, if any, for the selected comparison's mismatch. Test result items are compared by the following criteria:

Comparison	Explanation
Name	A comparison of the names of the two steps. This is useful for ensuring that you are, in fact, comparing similar hierarchies of results.
Step passed	A comparison of the <i>passed</i> property of the two results.
Number of passed steps	A comparison of the number of steps that have passed. Each step's results contains a value for the number of steps that have passed. Results for most types of steps have a value of one, if the step passed. For collection steps, this value is based on the number of steps that the collection step contains that have passed.
Number of steps	A comparison of the number of steps. Each step's results contains a value for the number of steps executed as a result of executing the step. For most types of steps, this value is one. For collection steps, this value is based on the number of steps that the collection step contains.
Execution time	A comparison of the execution times results for the two steps. If the execution times differ by the percent specified at the prompt, the comparison is marked as having a mismatch.

Test Results Differences Browser Tool Bar



The Tool Bar presents shortcuts to commonly used operations. From left to right they are:

Button Name	Button Function
 Find	Open the Find dialog.
 Expand	Expand all the comparison items contained by the selected item or items.
 Contract	Contract all the comparison items contained by the selected item or items.
 Report	Create a report of the selected comparison items.
 Results	Open the Results Browser view.

Test Results Differences Browser Menus

File

Item Name	Item Function
New	Open a new Smalltalk Workspace
Open...	Open a Smalltalk Workspace
Save Image...	Save the Smalltalk Image
Save Image As...	Save the Smalltalk Image with a new name
Exit Smalltalk Test Mentor	Exit the Test Editor

Edit

Item Name	Item Function
Find...	Open the Find dialog.

View

Item Name	Item Function
Expand	Expand all the comparison items contained by the selected item or items.
Contract	Contract all the comparison items contained by the selected item or items.

	selected item or items.
--	-------------------------

Tools

Item Name	Item Function
Report	Create a report of the selected comparison items.
Results	Open the Results Browser view.

Help

Item Name	Item Function
About Smalltalk Test Mentor...	Opens a window that describes the Smalltalk Test Mentor Product and how to contact SilverMark, Inc.

Method Coverage Browser

The Method Coverage Browser is used for determining which methods in your system under test are being executed as a result of executing test suites.

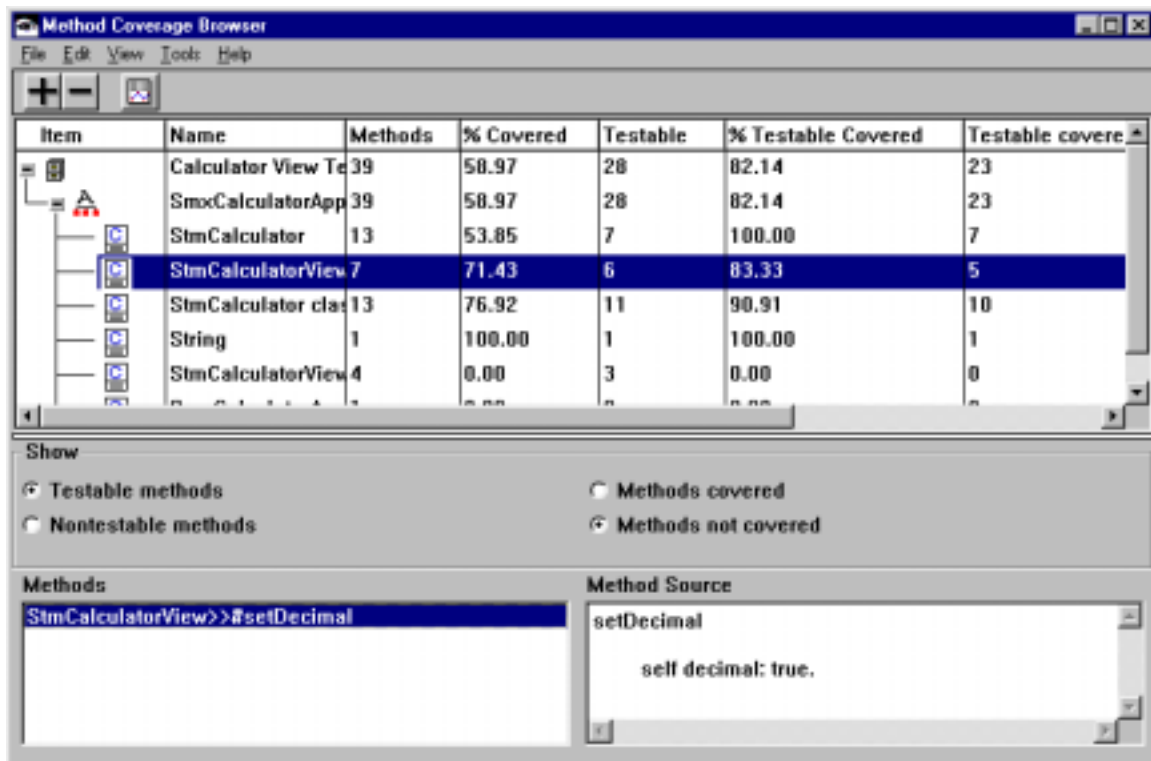
You launch the Method Coverage Browser by selecting the results of executing a suite in the Test Results Browser and then selecting the Coverage analysis... item in the Tools menu or the Coverage analysis... item in the list view popup.

The Test Results Differences Browser calculates method coverage metrics by suite, application or name space, and class. You can view methods that are covered by testing or not covered. The method coverage browser for VisualAge Smalltalk shows which methods are testable and which are not.

Note: In VisualAge Smalltalk, methods which are primitive or special cannot be tested. Methods whose bytecodes are in ROM cannot be tested.

Like many of the other Smalltalk Test Mentor browsers, the view is divided into an upper, list view and a lower, details view. The items in the upper view represent method coverage metrics for suite, applications or name spaces, and classes. The details view shows methods and their source strings that may or may not have been tested, depending on the desired view.

VisualAge:



The screenshot shows the Method Coverage Browser window with a table of coverage metrics and a details view for the setDecimal method.

Item	Name	Methods	% Covered	Testable	% Testable Covered	Testable covered
	Calculator View Te	39	58.97	28	82.14	23
	SmxCalculatorApp	39	58.97	28	82.14	23
	StmCalculator	13	53.85	7	100.00	7
	StmCalculatorView 7		71.43	6	83.33	5
	StmCalculator clas	13	76.92	11	90.91	10
	String	1	100.00	1	100.00	1
	StmCalculatorView 4		0.00	3	0.00	0

Below the table, there are two sections: "Show" and "Methods".

Show

- Testable methods
- Nontestable methods
- Methods covered
- Methods not covered

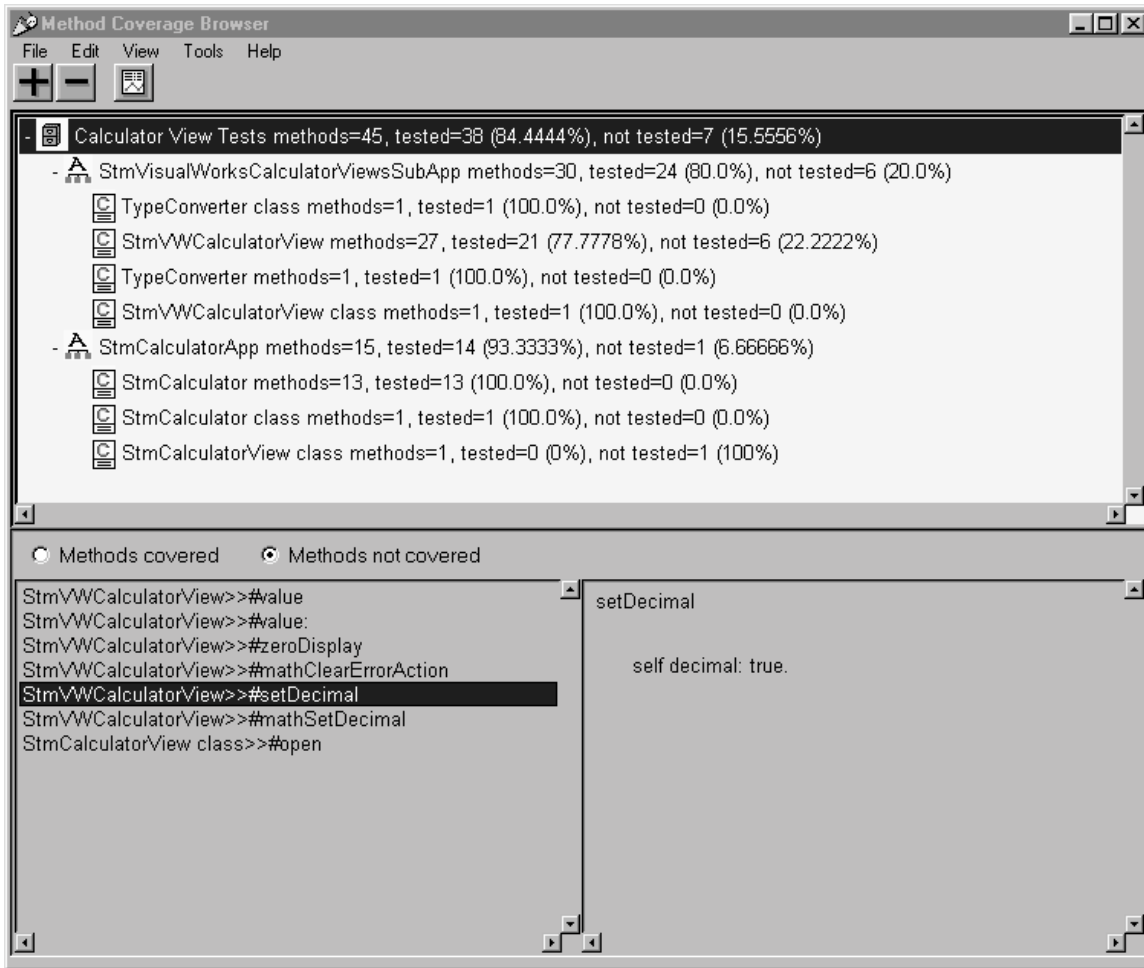
Methods

StmCalculatorView>>#setDecimal

Method Source

```
setDecimal
    self decimal: true.
```

VisualWorks:



Method Coverage Browser List View

The list view, like that of the previously described views, is a hierarchical collection of items. Each item in the Method Coverage Browser's list view corresponds to a suite, application or name space, or class. The following icons are used to denote the items:

Icon	Item
	Metrics for a suite
	Metrics for an application or name space
	Metrics for a class

In VisualWorks, the items in the list view are described as text in the form:

Name methods = mm, tested = tt (tt%), not tested = nt (nt%)

The following table describes the format of the text:

Text part	Description
-----------	-------------

Name	The name column contains the either the suite name, application or name space, or class, as appropriate.
mm	The number of methods contained by the item. For a suite, it is the total number of methods in all the applications specified as covered by the suite. For an application, it is the total number of methods controlled by the application. For a class, it is the total number of methods defined by the class.
tt	The total number of methods contained by this item that were tested
tt%	The percent of those methods contained by the item that were tested
nt	The total number of methods contained by this item that were not tested
nt%	The percent of those methods contained by the item that were not tested

In VisualAge, the columns in the list view show coverage metrics for the items. The following table describes the columns in terms of their values for each item:

Column Name	Description
Item	The item column shows the icon and containment relationship between items.
Name	The name column contains the either the suite name, application or name space, or class, as appropriate.
Methods	The number of methods contained by the item. For a suite, it is the total number of methods in all the applications specified as covered by the suite. For an application, it is the total number of methods controlled by the application. For a class, it is the total number of methods defined by the class.
% Covered	The percent of all methods contained by the item that were covered by execution of the suite.
Testable	The total number of methods contained by the item that are testable.
% Testable covered	The percent of those methods contained by the item that are testable.
Testable covered	The total number of testable methods contained by the item that were covered by executing the suite.
% Testable not covered	The percent of all testable methods contained by the item that were not covered by executing the suite.
Untestable	The total number of methods contained by the item that are not testable.

Method Coverage Browser Details View




The details view for the Method Coverage Browser shows several views of the methods contained by the selected item. You can view testable methods or nontestable methods. If you elect to view testable methods, you can either view methods covered or methods not covered. If you elect to view nontestable methods, that choice is not presented because, by being nontestable, their coverage status can not be determined.

When you select a method, the source string for it is displayed to the right of it.

Method Coverage Browser Tool Bar



The Tool Bar presents shortcuts to commonly used operations. From left to right they are:

Button Name	Button Function
 Expand	Expand all the items contained by the selected items, if any.
 Contract	Contract all the items contained by the selected item or items.
 Report	Create a report of the selected item or items.

Test Results Differences Browser Menus

File

Item Name	Item Function
New	Open a new Smalltalk Workspace
Open...	Open a Smalltalk Workspace
Save Image...	Save the Smalltalk Image
Save Image As...	Save the Smalltalk Image with a new name
Exit Smalltalk Test Mentor	Exit the Test Editor

Edit

Item Name	Item Function
There are no items in the Find menu	N/A

View

Item Name	Item Function
Expand	Expand all the items contained by the selected items, if any.

Contract	Contract all the items contained by the selected item or items.
----------	---

Tools

Item Name	Item Function
Report	Create a report of the selected item or items.

Help

Item Name	Item Function
About Smalltalk Test Mentor...	Opens a window that describes the Smalltalk Test Mentor Product and how to contact SilverMark, Inc.

The Preferences View

The *preferences view* is used to set global values for editing, recording and execution. Preferences are shown in a notebook with four pages:

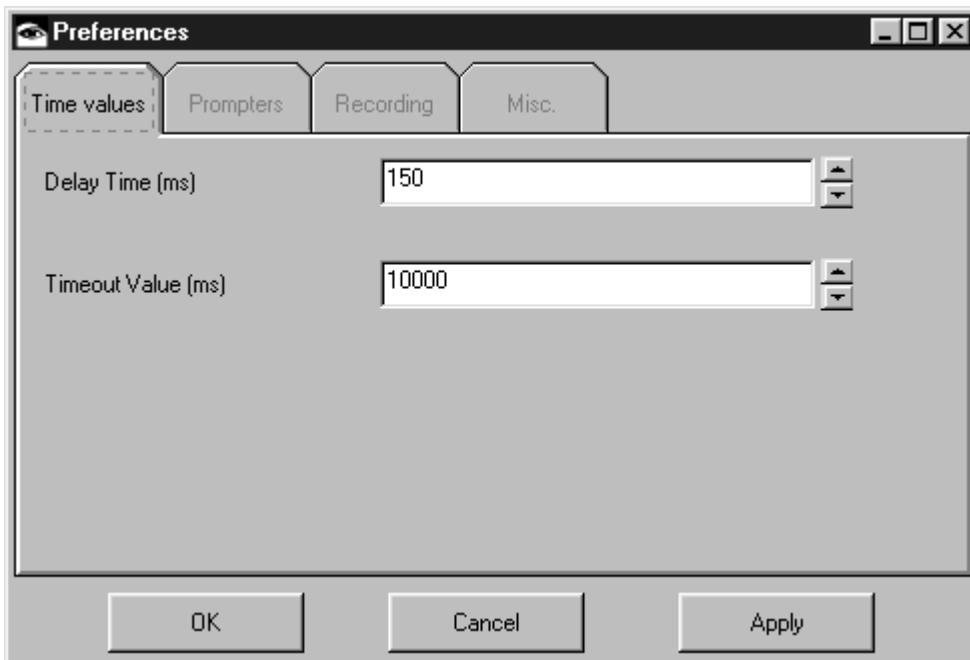
Time values - Values related to delays and settling times.

Prompters - Settings that effect recording and playback redirection of prompters.

Recording - Settings that effect the type of UI playback code that is generated

Miscellaneous - Settings that effect how exceptions are handled, restrictions on archived results and other unrelated items.

Time values



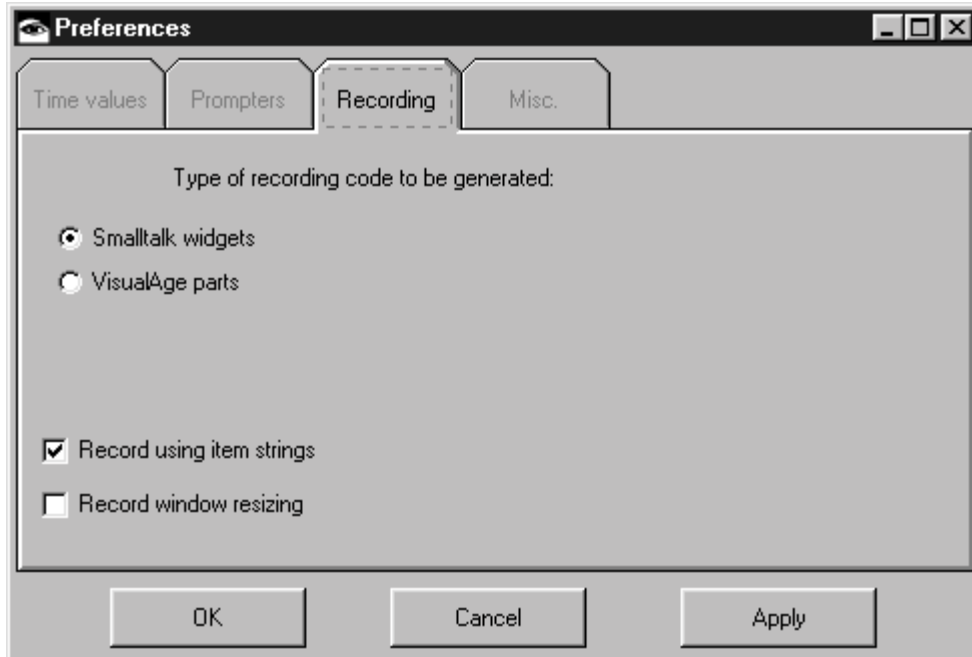
Name	Default value	Description
Delay time	150 (milliseconds)	The amount of time to delay after executing either a user interface step or a user interface verification step. This allows for settling after a UI operation.
Timeout value	10000 (milliseconds)	The amount of time to wait for a particular condition to be met. This is used when waiting for widgets to appear in order to play back user interface interactions with them. For example, if a click of a push button causes a window to be opened and the next operation is to set focus to that window, the timeout value is used to determine how long to keep trying to find that window because it may take a while to actually be created and opened. Presumably the timeout value allows enough time under normal circumstances for widgets to be addressable.

Prompters



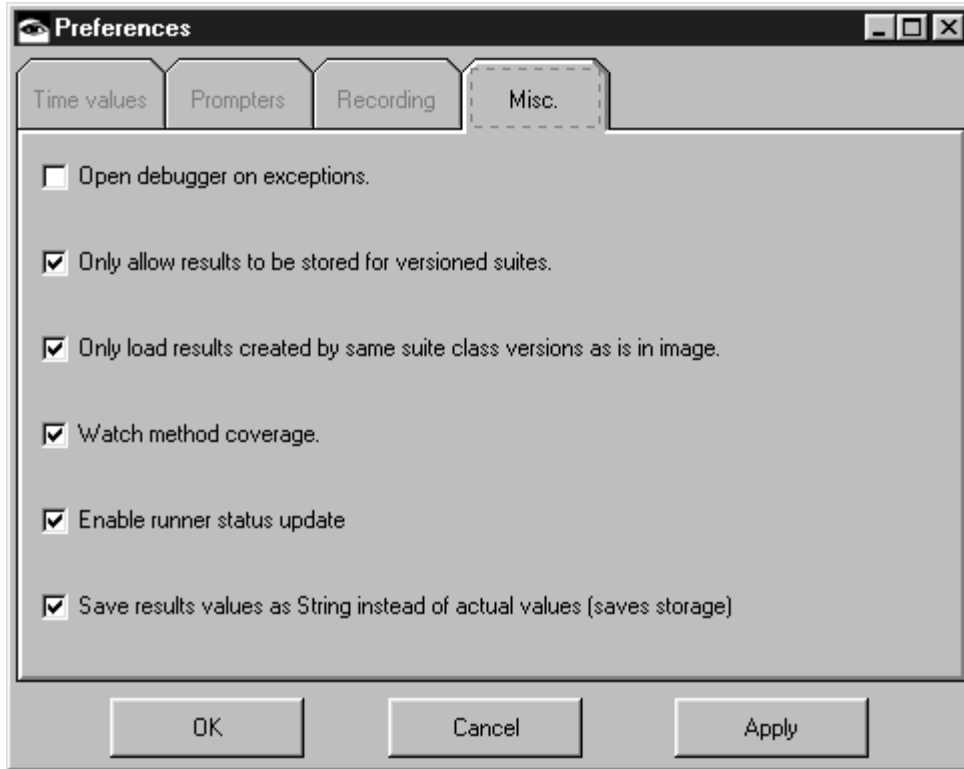
Name	Default value	Description
Enable prompter recording	true	Enable recording of prompter interactions.
Enable prompter runtime redirection	true	Enable redirection of system prompter values to that of their recorded values. If you expect that there will not be a one-to-one correspondence of recorded prompter values to those that appear, set this value to false .

Recording



Name	Default value	Description
Smalltalk widgets	true	Generate user interface playback code that addresses widgets directly. In VisualWorks, this is the only choice.
VisualAge parts <i>(VisualAge only)</i>	false	Generate user interface playback code that addresses VisualAge visual parts directly (only available for VisualAge Smalltalk).
Record using item strings	True	Generate list selection playback steps that reference the list item by display string instead of index.
Record window resizing <i>(VisualAge only)</i>	False	Generate user interface playback steps for window resize events (VisualAge only)

Miscellaneous



Name	Default value	Description
Open debugger on exceptions	false	An exception during the execution of a step is trapped and logged in the results for that step. The Test Runner provides a setting to pause on the detection of an exception. Under some circumstances, it is desirable to circumvent the exception handling facilities that the Smalltalk Test Mentor provides and open a debugger directly on the process that originated the exception. By setting this to true , you can do this.
Only allow results to be stored for versioned suites (ENVY/developer only)	true	Test results should almost always be tied directly to a versioned test suite class. Some times, you may want to vary tests slightly to vary the stimulus to the application under test and log its reaction. You can use this setting to disable the checking that prevents you from storing results from a suite class that is not versioned.
Only load results created by same suite class versions as in image (ENVY/developer only).	true	When you load archived test results, you should almost always load them into an image that contains the same suite class that created those results. Under some circumstances, (see the above item), you may wish to load results from a different version of the suite class or an edition that is not versioned. This setting allows you to

		disable the checking that prevents exact suite class version verification.
Watch method coverage	true	In order to more realistically measure execution times, you may wish to disable method coverage watching during test execution.
Enable runner status update	true	Enable display of test execution progress status. It is some times necessary to turn this off in order to minimize extraneous UI activity while tests are executing.
Save results values as String instead of actual values (saves storage)	true	When a step executes, the result of it having executed is stored with the test results and made available from the Test Results browser for display and inspection. For some steps, this results in extension of some objects' tenure that would otherwise be garbage collected. For example, the results of executing a UI playback step generally holds on to the widget that was the target of the interaction. Use this option to automatically store the #printString representation of the object instead of the object itself.

Warning 1: You should be careful when you elect to have the debugger open on exceptions on a test that has steps that are specified to abort on fail. When a failure is detected, an exception is signaled that is normally trapped internally. When you disable internal exception handling, those failures manifest themselves as exceptions.

Warning 2: You should be **EXTREMELY CAREFUL** when disabling checking on suite class versions in the above settings. Structural changes in tests, like different numbers of steps or step hierarchy differences, can cause results to be inconsistent or incorrect.

Recording and playing back prompters and suspended views ***(VisualAge Smalltalk Only)***

There are two special cases to recording and playing back user interface interactions: system prompters and views that force their calling processes to suspend until those views are closed. The Smalltalk Test Mentor handles these special situations nearly transparently. This section explains the special considerations that need to be made for steps generated as a result of recording these situations.

System prompters

System prompters provide a unique challenge to recording and playing back user interface interactions because they are composite views implemented by the Operating System instead of in Smalltalk. The solution to this challenge is to record and play back the values *returned* by the prompters, rather than the interactions with the prompters' widgets themselves. During user interface recording, when a system prompter returns a value, a step is generated that contains the prompter's returned value. During user interface interaction playback, when that step is run, it queues that value in a list of values to return for encountered prompters. When a system prompter is encountered, instead of opening the prompter and transferring control to the operating system, the next queued prompter value is returned. Because of this, you will never actually see a system prompter open during playback.

The main item to note is that the returned value for the prompter must be set *before* the action that causes the prompter to be opened. Steps that set these values can appear any place within a test prior to the action that initiates the prompter. They are usually generated with a description that starts with **Return prompter value**:

Because of the nature of system prompter playback, it is possible to cause your image to enter a state where system prompters are not opened, if a prompter is requested, outside of the main stream execution of a test. For instance opening a debugger from a paused step could be problematic if using it caused a prompter to be opened. Because of this, *prompter playback is automatically disabled whenever you open a debugger during execution of a test, for the rest of the duration of the test.*

Suspended views

Many views use `CwAppContext>>#readAndDispatchWhile:` or `AbtShellView>>#suspendExecutionUntilRemoved` to block the process that opened the view until the view is closed. Because user interface interactions are run on a separate thread, there isn't a problem.

When the Smalltalk Test Mentor executes a test, it effectively operates on a single thread of execution. When a user interface automation message is sent³, the message is executed on the same thread that the test is executed on. If an automation message results in the test process being blocked⁴, no more automation messages can be sent, including those required to manipulate the suspended view, or any required to close the view and allow the process to be resumed.

The way the Smalltalk Test Mentor circumvents this problem is by implementing special code that executes the steps required to automate the suspended view during the time it would normally be suspended, without actually allowing it to be suspended.

When the Test Editor records user interface interactions, it structures the steps in this way automatically. The description generated for the step that contains the steps that automate a suspended view is set to **<Playback for prompter / suspended window>** for identification.

For example, consider a system with a logon dialog that is launched as a result of a push button click and then suspends the thread that launched it until the logon dialog is destroyed (as a result of Ok being pressed). A recorded scenario might be to click on the

³ For example, a request to click a pushbutton, like `(ActiveWindow widgetNamed: 'Go') click`

⁴ From either `CwAppContext>>#readAndDispatchWhile:` or `AbtShellView>>#suspendExecutionUntilRemoved`

Logon push button that launches the logon dialog, type in a name and password and then click Ok:

Set focus to 'Main window'

Click on Logon

Set focus to 'Logon window'

Enter userID

Enter password

Click Ok

Set focus to 'Main window'

Verify status text = 'Logged On'

The recording for this would appear as a single step that launches the logon dialog followed by a UI recording step containing the actions to perform while the suspended dialog is, in fact, suspended, and then a verification step:

Step Name	Step Type
Set focus to 'Main window'	UI script
Click on <u>Logon</u>	UI script
<Playback for prompter / suspended window>	UI recording
Set focus to 'Logon window'	UI script
Enter password	UI script
Click <u>Ok</u>	UI script
Set focus to 'Main window'	UI script
Verify status text = 'Logged On'	UI verification

When the above steps are executed, the test runner causes the Logon button to be clicked. When the logon window suspends itself, the very next step, as well as any steps that it contains, are executed. These steps should perform the actions on the suspended view that would normally be performed while it is suspended *including any steps required to close it.*

SilverMark's Smalltalk Test Mentor User's Guide

Creating and Editing Tests

The features described in this section are only applicable to the *Smalltalk Test Mentor* product. You cannot create or edit tests from the *Smalltalk Test Mentor - Runner* product.

How to open the Test Editor

The Test Editor can be opened from either the System Transcript or, in VisualAge Smalltalk, the VisualAge Organizer.

If you are using either VisualAge Smalltalk, or are using VisualWorks without *ENVY/developer*, select the Tools menu, and then SilverMark's Test Mentor Editor....

If you are using VisualWorks with *ENVY/developer*, select the ENVY menu and then SilverMark's Test Mentor Editor....

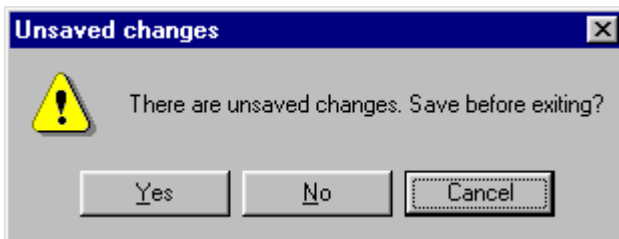
In VisualAge Smalltalk you can open the editor from the VisualAge Organizer by navigating to the Options menu and selecting SilverMark's Test Mentor Editor....

How to close the Test Editor

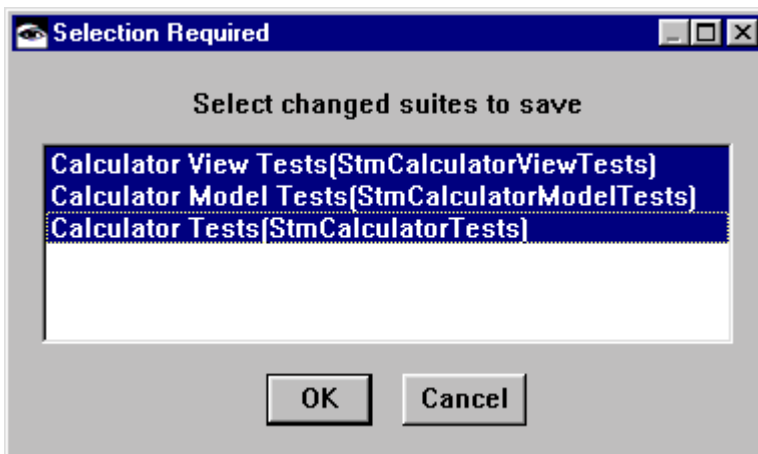
You can close an open test editor by any of the following means:

- Close the Test Editor window using the system provided window close mechanism
- Select the Exit Smalltalk Test Mentor menu item in the Test Editor's File pulldown menu.

You may be prompted to save changes if you have made any changes to suites, scenarios or steps since opening the Test Editor.




If you select Cancel, the request to close the window is cancelled. If you select No, the view is closed and any changes are discarded. If you select Yes, you are prompted to select which changes to save by selecting the classes in which the changed suites, scenarios or steps are defined.



When you press **OK**, changes are saved in the form of code generated into the selected suite classes. If you select **Cancel**, all changes are discarded.

How to create a new suite

- From an open Test Editor, select either no steps or an existing suite, then perform either of the following:
- Select the **New Suite...** menu item from the **Edit** pulldown menu.
- Select the **New Suite...** menu item from the list view popup menu
- Press the **New Suite...**  button on the tool bar.

The **Create new suite** dialog opens:

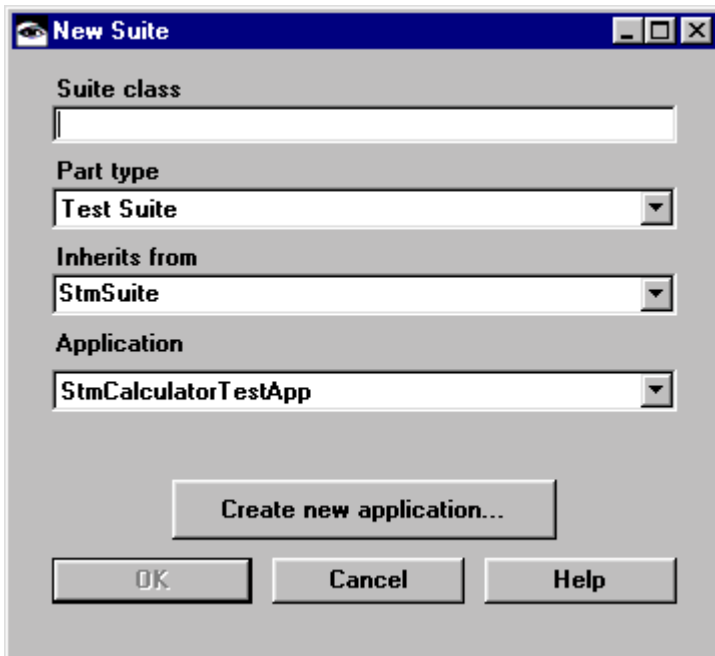


Figure 7 - New suite dialog for systems with ENVY/developer

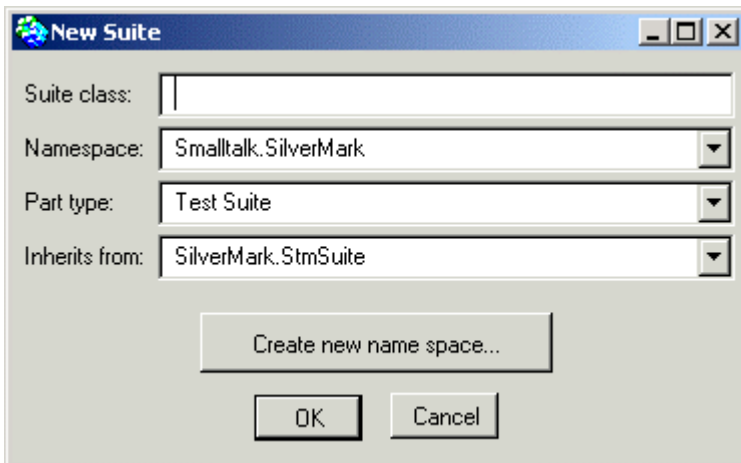



Figure 8 - New suite dialog for systems without ENVY/developer

Fill in the following entry fields in the above New Suite dialog:

Entry field name	Description
Suite class	The name of a class to create. This class will represent the test suite that you are creating. All scenarios and steps you create will be defined as methods within the class you specify here.
Part Type	This is already set to Test Suite . You can not change this.
Inherits from	Select a class that your new class should inherit from. The default is StmSuite, which is the root class that all suites must inherit from.
Application (systems with ENVY/developer)	The application in which to create the new suite class. You can create a new one by pressing the <u>Create New application...</u> button. If you do this, a prompt for the application is presented.
Namespace (systems without ENVY/developer)	The namespace in which to create the new suite class. You can create a new one by pressing the <u>Create New name space...</u> button. If you do this, a prompt for a new name space is presented.

How to create a new scenario

From an open Test Editor, select either a suite or an existing scenario, then perform either of the following:


- Select the New Scenario... menu item from the Edit pulldown menu.
- Select the New Scenario... menu item from the list view popup menu
- Press the New Scenario...  button on the tool bar.

You will be prompted for the scenario name.

Note: The name you use will also be used to generate the selector for the scenario. Once this value is set, you can not change it. The selector is derived from the scenario name whereby any characters that are not valid for a method selector are compressed out.

How to add a step or steps to a scenario

From an open Test Editor, select either a scenario or an existing step, then perform either of the following:

- Select the New Step... menu item from the Edit pulldown menu.
- Select the New Step... menu item from the list view popup menu
- Press the New Step...  button on the tool bar.

The Create new step dialog opens:



Fill in the following fields in the above dialog:


Entry field name	Description
Name of Step	The name of the step. It can be any string, although it is best to make it descriptive but brief.
Type of step	You can select any of the step types described in the <i>Smalltalk Test Mentor User's Guide</i> in this drop-down list. You can always leave Unspecified selected and change it later from within the editor.
How Many?	Specify the number of new steps of the above type to add. Each step will be created with the above name with the step's index appended to it.
Add as substep	If the selected step is a collection step, this field will be enabled. If you check the checkbox, the new steps will be added to the collection step, rather than as a peer immediately following it.

When you press OK, the step will be added either immediately following the step you selected or as the last step within the collection step you selected, if you checked Add as substep.

How to change the ordering of steps within the Test Editor

You can change the ordering of steps within the Test Editor by selecting them and dragging them to their new location. In VisualAge Smalltalk, when you move one or more steps the insertion point is indicated by a line drawn between two steps. In VisualWorks, the insertion point is always the step immediately following the step that the pointer is over or within the step that the pointer is over if it is a collection-type step.

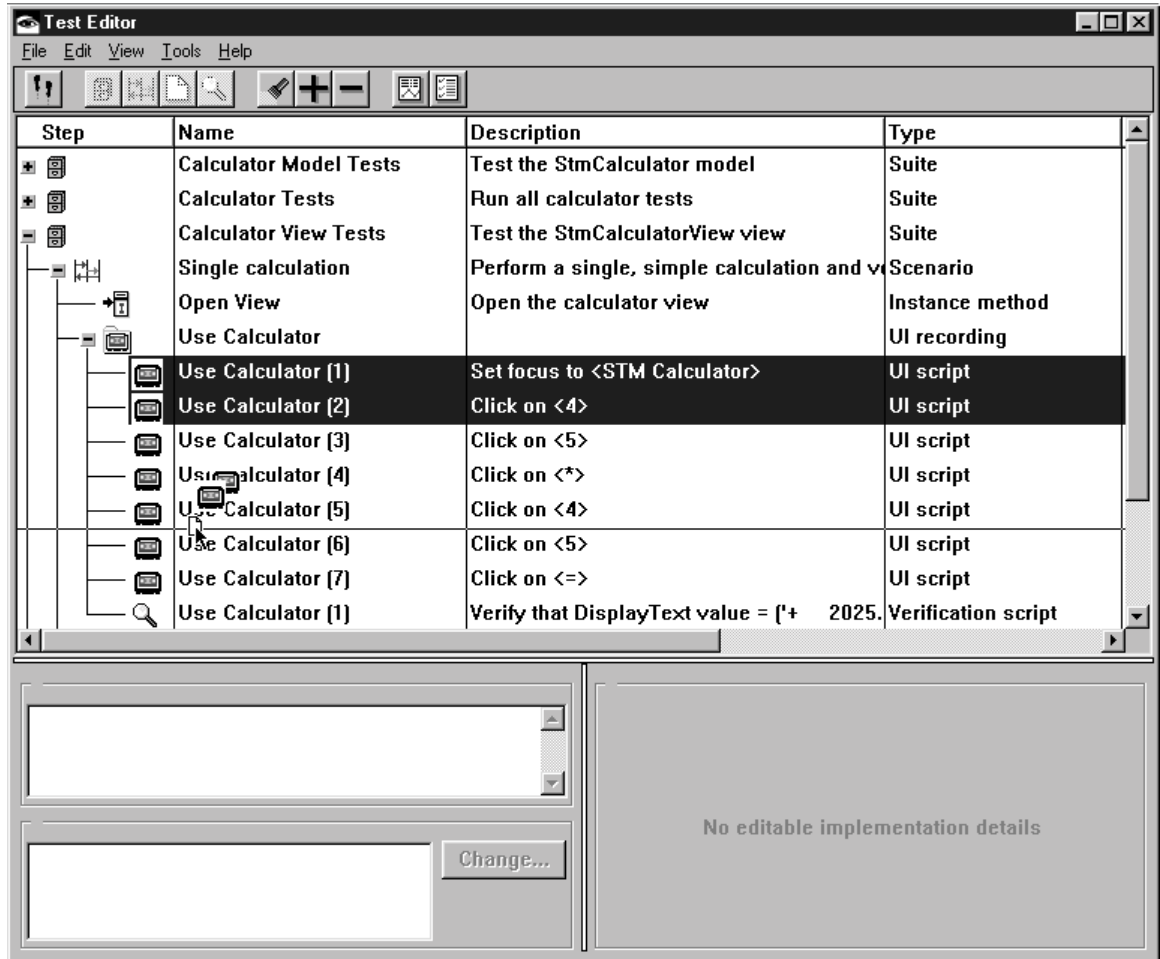
Try the following experiment. You'll need to load the Smalltalk Test Mentor Examples configuration map or STM-Calculator parcel into your image to try it. In the following example, you will select two steps and move them to a new location:

1. Open the Test Editor
2. Expand the **Calculator View Tests** suite and the **Single Calculation** scenario within it until you can see the steps that compose the **Use Calculator** step. You can press the Expand  button with the **Calculator View Tests** suite selected to do this.

3. Select the steps named **Use Calculator (1)** and **Use Calculator (2)**.
4. Press the mouse button that initiates item drag on your operating system⁵.

VisualAge:

1. Slowly move the mouse to move the item downward toward the step named **Use Calculator (6)**. You will see the item's icon moving along with the cursor and a line that indicates the insertion point.

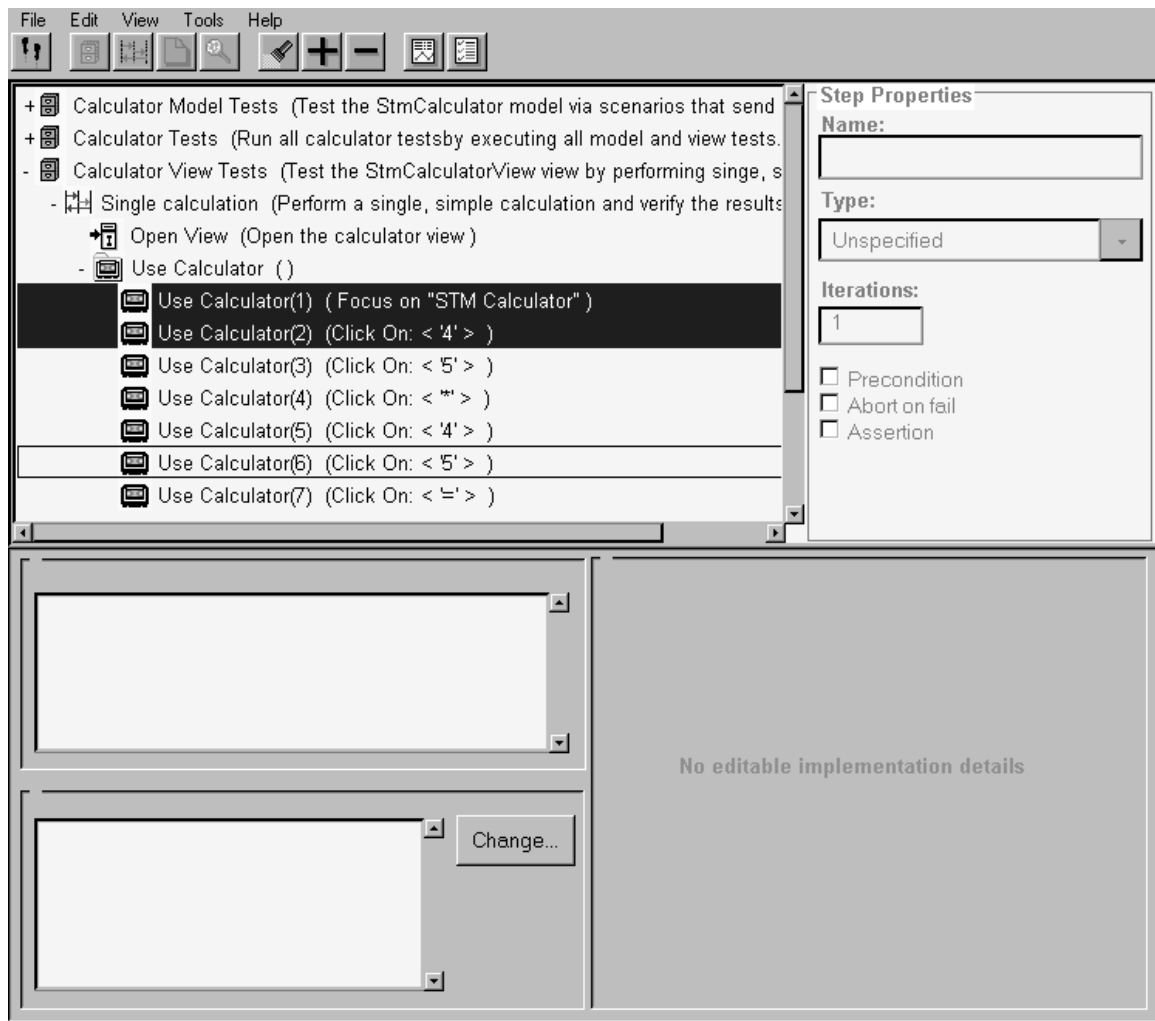


2. Stop moving the mouse when the insertion point line appears immediately after the item named **Use Calculator (6)**..
3. Release the mouse button - The **Use Calculator (1)** and **Use Calculator (2)** should now follow **Use Calculator (6)**.

VisualWorks:

1. Slowly move the mouse to move the item downward toward the step named **Use Calculator (6)**. You will see the item's icon moving along with the cursor. **Note: You must not release the mouse button between selecting the steps and starting to drag.**

⁵ Some versions of VisualAge Smalltalk have a defect in the base drag-drop support that requires that you place the mouse cursor over the first of the items to drag, otherwise the steps are dropped in the wrong order.



2. Stop moving the mouse when the cursor is over the item named **Use Calculator (6)**..
3. Release the mouse button - The **Use Calculator (1)** and **Use Calculator (2)** should now follow **Use Calculator (6)**.

Note 1: When you drop steps with the insertion point following a collection step or UI recording step, a prompt is displayed with the following question:

Drop item(s) INTO the collection step?

If you answer YES, the item will be moved immediately following the last item within the collection step. You may then move it again if you need the item to be placed elsewhere within the collection step.

If you answer NO, the item will be moved immediately following the collection step.

Note 2: The following restrictions should be noted:

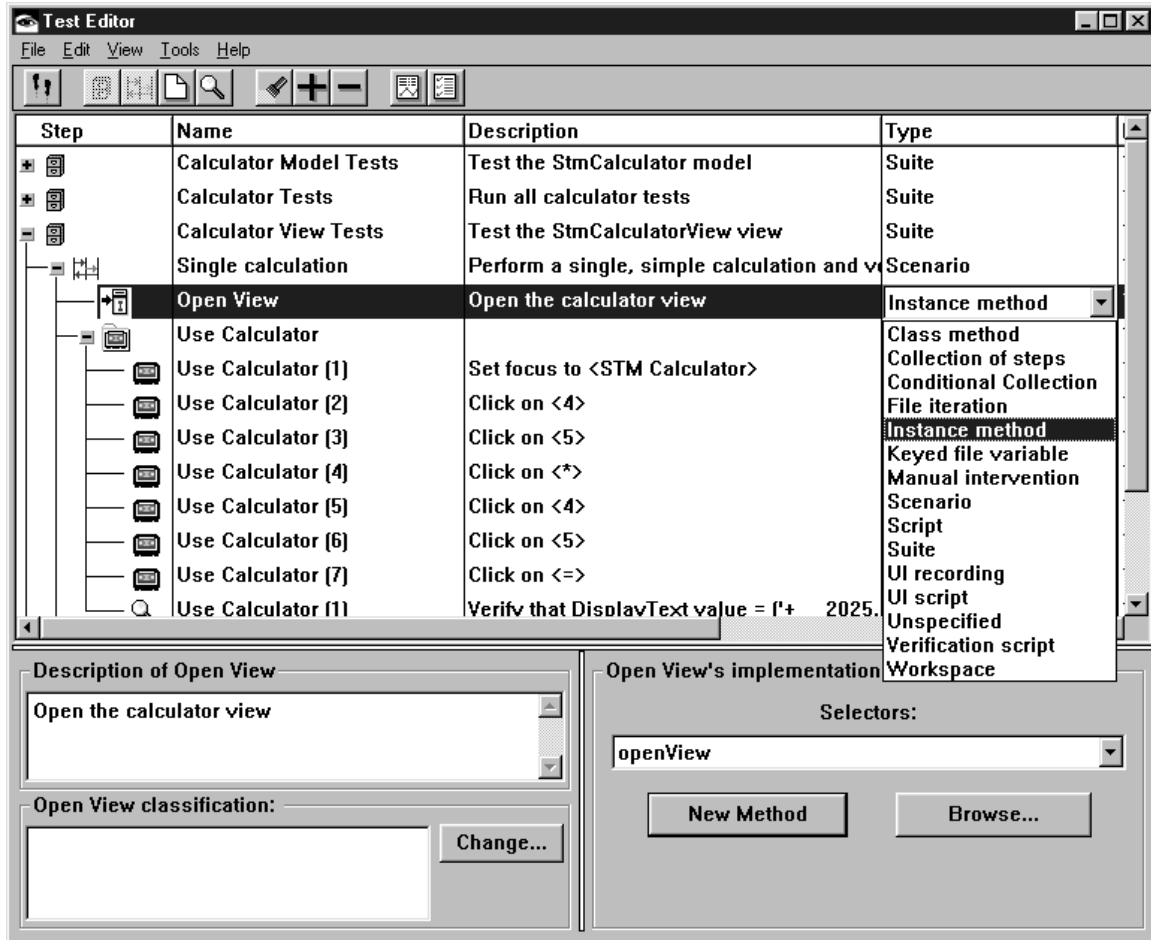
- You can not move a suite
- You can not move a scenario to another suite

How to change step types

You can change the type of a step from the Test Editor. To do this:

VisualAge:

1. Select the Type column of the step whose type you wish to change. The selected cell will become editable, that is, the cell will change to a drop-down list.

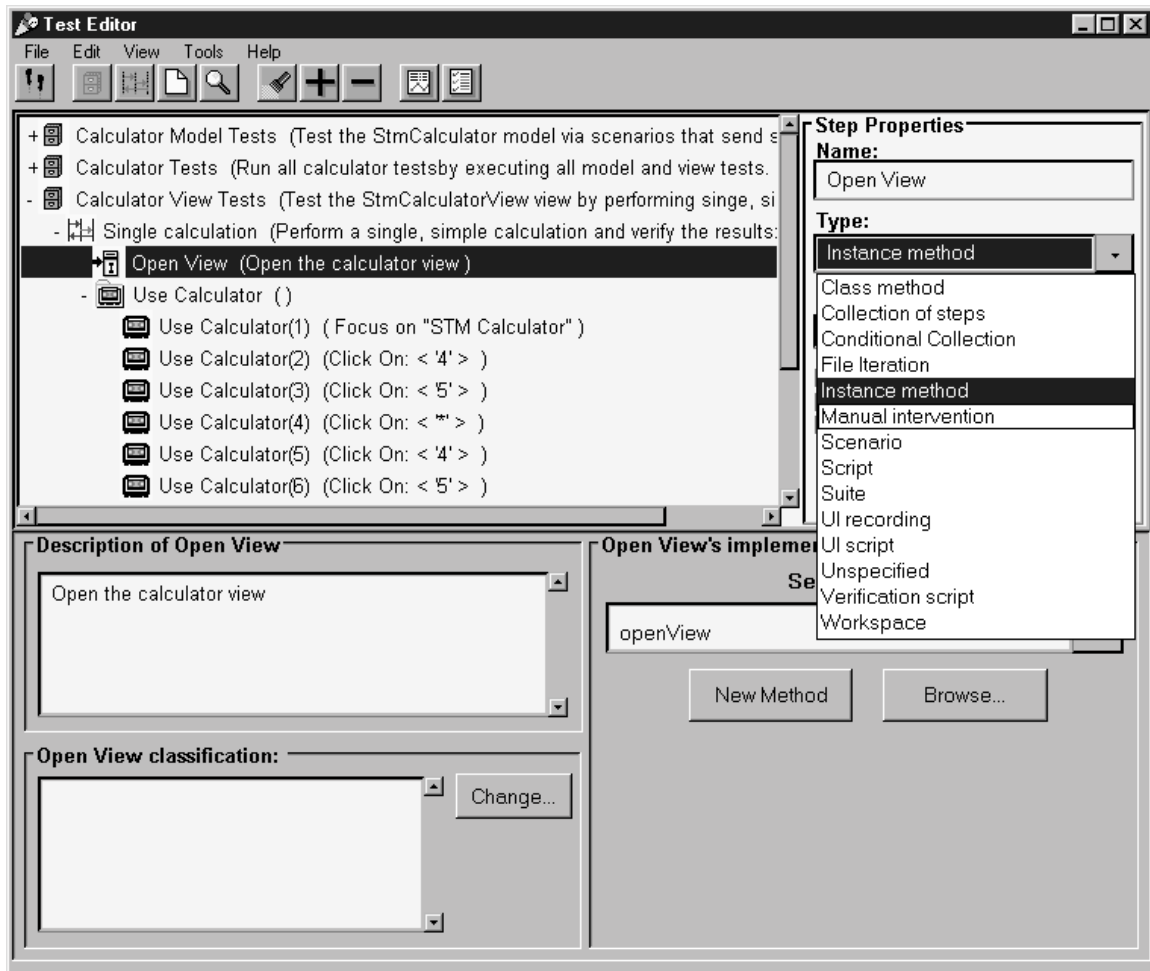


2. In the drop-down list, select the type of step that you wish the selected step to become.
3. Select any other cell or item to commit the change.

You can now change the specifics of the new type for the step in the Details View part of the editor.

VisualWorks:

1. Select the step to be changed



2. In the properties area of the list view, click on the Step Type drop down list and select the type of step that you wish the selected step to become.
3. Tab out of the list or select another control to accept the change.

How to change the number of times a step is executed

There are three ways to cause a step to be executed multiple times:

- You can set the value in the Iterations step property in the Test Editor for a step to the number of times to execute the step.
- You can repeat the execution of a script, user interface, user interface verification, instance method or class method step by sending #repeatIteration or #repeatIterationIf: from within the step. Each time this message is sent, the iteration count is decremented, forcing the currently executing iteration to be repeated once it has completed.
- You can embed one or more steps within a file iteration step so that they are executed one time for each row in the specified test data file.

How to temporarily disable execution of a step

If you are developing a test and wish to cause a step to not execute, but do not want to delete it, simply set the value of Iterations for the step to **0**.

How to delete suites, scenarios, or steps

To delete one or more suites, scenarios, or steps, select the items to delete in the Test Editor details view and perform one of the following:

- Select the Delete menu item from the Edit pulldown menu.
- Select the Delete menu item from the list view popup menu
- Press the Delete key (VisualAge only).

Before actually deleting an item or items, the Test Editor issues a confirmation prompt. If you are deleting a suite, a prompter will also be displayed with the following question:

Do you really want to delete the class: MySuite1

If you answer YES, the suite, as well as its class is deleted.

Note: If there are instances of the class in the image, you may get the following message:

Error: 132

MySuite1 cannot be removed because it has instances

If you receive this message, close down browsers that might reference that class's suite and try again. If you still cannot delete the class, execute the following code to remove all instances:

VisualAge:

```
MySuite1 allInstances do: [ :instance | instance become: nil ].
```


VisualWorks:

```
MySuite1 allInstances do: [ :instance | instance become: String new ].
```

How to record user interface interactions

You can manually encode steps to simulate user interface interactions but that can be very time consuming. The Smalltalk Test Mentor provides extensive facilities for recording user interface interactions to enable you to rapidly create tests.

User interface interactions are almost always recorded into steps within a User Interface Recording (UI Recording) step. There are two ways to record steps into a UI Recording step:

- You can cause an UI Recording step to be executed from Quick Runner. The Quick Runner recognizes empty UI Recording Steps and will capture any user interface interactions when it reaches the empty UI step.
- You can select the UI recording step and select the record  button from the UI recording step details view.

Note: Regardless of which mechanism you use, you should always remember that you can not record a view unless it has been opened **after** the Test Editor in which you are recording was opened.

How to capture user interface interactions during execution

It is a good idea to partition user interface interactions into multiple UI recording steps. This enables you to more easily manage updates to your views. For example, consider interaction with a complex view that contained a radio button set. If you recorded an entire interaction with that view into one UI recording step, you would have to re-record that interaction if the radio button set was changed to a group of check boxes. If you partitioned the interaction into several UI recording steps, you would only have to re-record the one that contained steps that simulated interactions with the radio button set.

The following example will lead you through creating a test that interacts with a simple view. The example test will be partitioned into three UI recording steps. You will need to load the Smalltalk Test Mentor Examples to try this example yourself.

1. Open a Test Editor.
2. Select the Calculator Views Tests suite.
3. Add a new scenario to the suite named **Operators tests**, then select the **Operators tests** scenario and add five steps to it. You can add the steps at the same time or individually. Create the steps like this:

Step name	Step type
Open calculator	Script
Add	UI Recording
Subtract	UI Recording
Multiply	UI Recording
Divide	UI Recording
Close calculator	UI Recording

4. Enter the following code into the first step. Don't forget to press Accept before moving on.

In VisualAge:


```
StmCalculatorView new openWidget.
```

In VisualWorks 2.5 or 3.0:

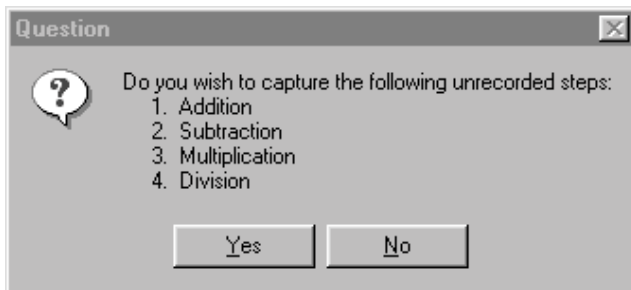
```
StmCalculatorView open
```

In VisualWorks 5i.4 or greater:

```
SilverMarkCalculator.StmCalculatorView open
```

5. Select the scenario, then press the Quick Run  button on the tool bar to run the scenario.

You will see the following prompt:



6. Answer Yes to continue. If you select No, the empty UI recording steps will be executed, but this accomplishes little.
7. The Test Editor window executes all steps up until the first UI recording step to be captured. For this example, this means executing the script step that opens the calculator. Once it reaches the first empty UI recording step, the Test Editor window selects that step and undergoes a slight transformation by adding a status area to the bottom of its window:

Test Editor File Edit View Tools Help

Step	Name	Description	Type
+	Calculator Model Tests	Test the StmCalculator model	Suite
+	Calculator Tests	Run all calculator tests	Suite
-	Calculator View Tests	Test the StmCalculatorView view	Suite
+	Single calculation	Perform a single, simple calculation and v	Scenario
-	Simple Calculations	Test simple calculations without cascading	Scenario
	Open View	Open the calculator view	Instance method
	Addition	1 + 2 = [3]	UI recording
	Subtraction	9 - 6 - 2 = [1]	UI recording
	Multiplication	5 * 5 * 2 = [50]	UI recording
	Division	100 / 4 = [25]	UI recording

Description of Addition

1 + 2 = [3]

Addition classification:

Change...



Addition 's implementation details:

— Recording —

Stop Recording

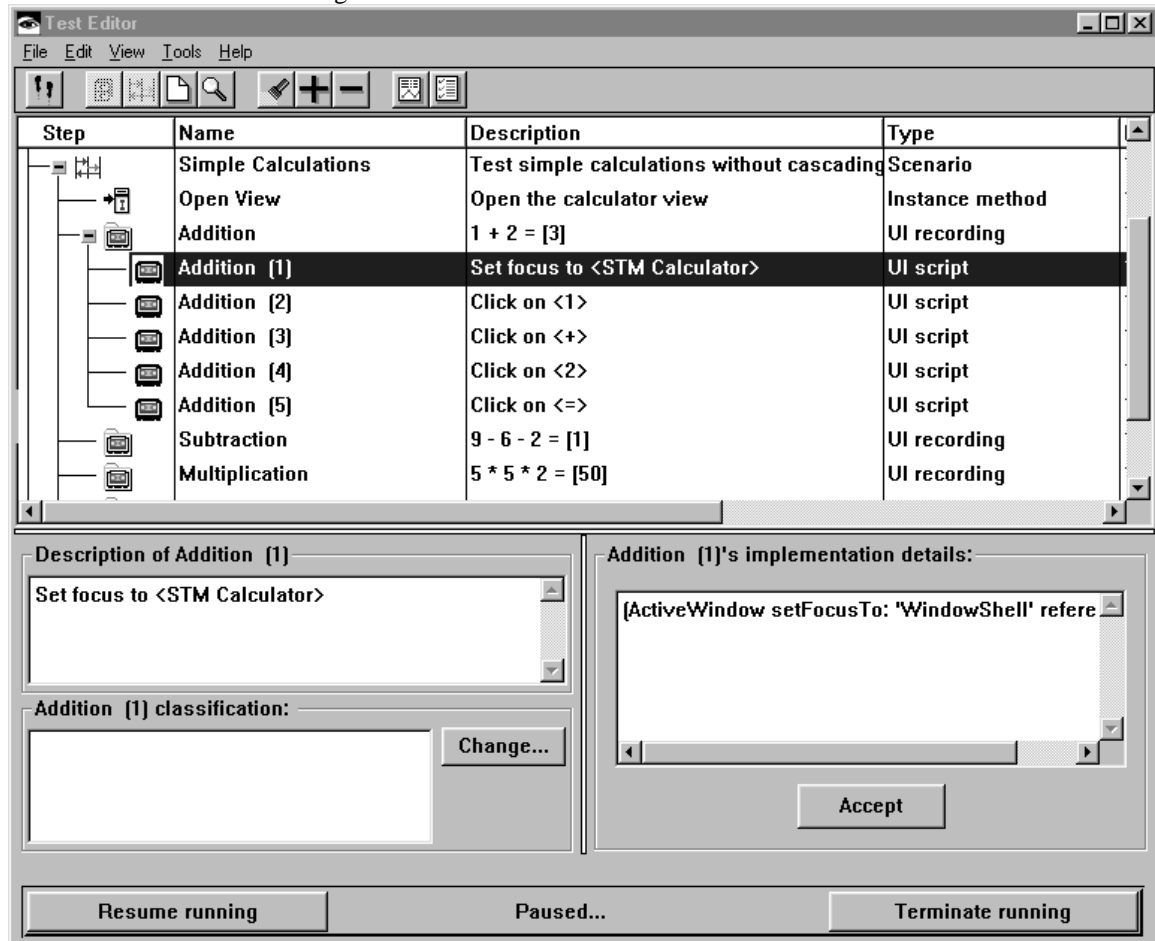
Recording...

The three elements of the status view are:

Status view element	Description
	<p>This button appears while the Smalltalk Test Mentor is recording user interface interactions into the currently selected UI Recording step. Press this button to stop recording and move on to execute the next step, if any.</p>
<p>Recording...</p>	<p>This shows the current status of recording.</p>
	<p>This animated icon also shows that the Smalltalk Test Mentor is currently recording user interface interactions.</p>

8. At this point the Smalltalk Test Mentor is recording user interface interactions. Any that it detects will be added to the UI recording step on whose behalf the steps are being recorded, in this case, the step named **Add**.
9. To give it something to record, use the calculator keys to perform the add operation: **1 + 2 =** then press the Stop Recording button. The recorded steps are added to the **Add** UI recording step and the status

section of the Test Editor changes:




At this point capture is paused. You can adjust the generated step or perform other Test Editor operations. In general, it is best to keep editor interactions to a minimum while paused during capture so as not to alter the structure of a test whose user interface steps are being recorded into. This is, however, an excellent time to add UI verification steps. This will be covered in the section titled, *How to verify Widget State*.

10. Press Resume running to continue capture for the next step.
11. Use the calculator keys to subtract one from the current total: $\text{= } 1 \text{ =}$ then press the Stop Recording button. The recorded steps are added to the **Subtract** UI recording step and the status section of the Test Editor changes as before.
12. Press Resume running to continue capture for the next step.
13. Use the calculator keys to multiply by five: $\text{* } 5 \text{ =}$ then press the Stop Recording button. The recorded steps are added to the **Multiply** UI recording step and the status section of the Test Editor changes as before.
14. Press Resume running to continue capture for the next step.
15. Use the calculator keys to divide by three: $\text{/ } 3 \text{ =}$ then press the Stop Recording button. The recorded steps are added to the **Divide** UI recording step and the status section of the Test Editor changes as before.
16. Press Resume running to continue capture for the next step.

17. Close the calculator view, then press the **Stop Recording** button. The recorded steps are added to the **Close calculator** UI recording step and the status section of the Test Editor changes as before.
18. Even though this is the last step to be captured, you still need to resume the test so it can complete. Press **Resume running** to do this. Once the test completes, the Test Results Browser is opened with the results of having run the scenario. You can dismiss this window.
19. Now look at the Test Editor window to see how your interactions with the calculator are represented in the form of UI steps within UI recording steps.

How to record directly into a User Interface Recording step

For simple cases, you may want to record user interface interactions directly into a UI recording step. You

can do this simply by pressing the **Record**  button inside a selected UI recording step's details view of the Test Editor. Any interactions with views opened after the start of recording are generated into UI script steps and inserted at the end of the steps within the selected UI recording step. The following example shows how to do this. You will need to load the Smalltalk Test Mentor Examples to try it yourself.

1. Open a Test Editor.
2. Select and expand the **Calculator Views Tests** suite and then the **Single Calculation** scenario within it.
3. Select the **Use Calculator** step.
4. Before you begin recording, you should open any views that you wish to record interactions with. For this example, launch the calculator (`StmCalculatorView`) by executing the following from a workspace:

In VisualAge:


```
StmCalculatorView new openWidget.
```

In VisualWorks 2.5 or 3.0:

```
StmCalculatorView open
```



In VisualWorks 5i.4 or greater:

```
SilverMarkCalculator.StmCalculatorView open
```

5. Press the **Record**  button inside the selected UI recording step's details view. The Test Editor window undergoes a slight transformation by adding a status area to the bottom of itself:

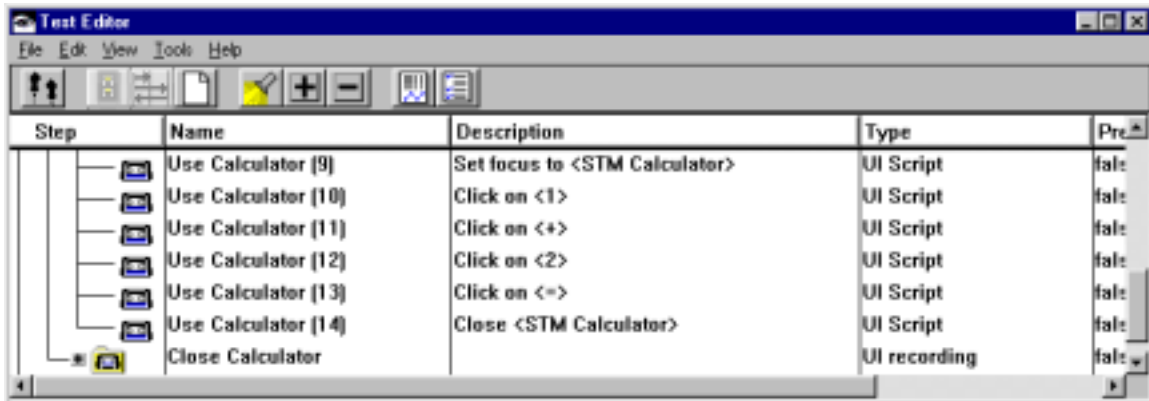


The three elements of the status view are:

Status view element	Description
	This button appears while the Smalltalk Test Mentor is recording user interface interactions into the currently selected UI Recording step. Press this button to stop recording.
Recording...	This shows the current status of recording.
	This animated icon also shows that the Smalltalk Test Mentor is currently recording user interface interactions.

- At this point the Test Editor is recording user interface interactions. Any interactions that it detects will be added to the UI recording step on whose behalf the steps are being recorded, in this case, the step named **Use Calculator**.
- With the calculator open, press the following keys: **1 + 2 =** and close the calculator.

8. Now press the Stop Recording button in the status area of the Test Editor.
9. Look at the steps within the **Use Calculator** step. The interactions you recorded are added at the end of the steps.



How to specify whether UI playback code is generated as widget or VisualAge part interactions (VisualAge only)

The output from recording a user interface interaction is a user interface step containing a block of code to execute that plays back that interaction. This code can be either directed toward widgets (Cw or Ew classes) or VisualAge visual parts (Abt classes).

In general, if your application user interface is developed using VisualAge parts, you should record at the VisualAge part level; especially if you use converters. To change the type of code generated:


1. Open the preferences view by selecting the Preferences... item of the Tools menu of the Test Editor or Test Browser.
2. Select the Recording tab.
3. Select the appropriate radio button and press Apply or OK.

Note 1: This must be done *before* recording begins.


Note 2: If you are experiencing playback problems you should try switching to a different mode and re-recording.

How to verify widget state

In addition to exercising your user interface widgets you should also verify that they are in an expected state as a result of various stimuli. You can do this with the Visual Verification Wizard. You can open this

at any time when a scenario or step is selected, or from a UI recording step by pressing the Verify UI 

button (or the F2 key in VisualAge). When you do this, the mouse pointer turns into a **?** in VisualAge, or

a  in VisualWorks. With the pointer in this state, press the mouse selection button over the widget you wish to verify, which opens the Visual Verification Wizard window for the selected widget. You then select which aspects of the widget you wish to verify and then indicate where to insert the new UI verification steps within the Test Editor list view. The following example shows how to add UI verification steps while capturing user interface interaction. You will need to load the Smalltalk Test Mentor Examples configuration map or STM-Calculator parcel to try it yourself.

1. Open a Test Editor.

2. Select the Calculator Views Tests suite.
3. Add a new scenario to the suite. Name it **Verify Add**. Select the **Verify Add** scenario and add two steps to it. You can add the steps at the same time or individually. Create the steps like this:

Step name	Step type
Open calculator	Script
Add	UI Recording

4. Enter the following code into the first step. Don't forget to press Accept before moving on.

In VisualAge:


```
StmCalculatorView new openWidget.
```

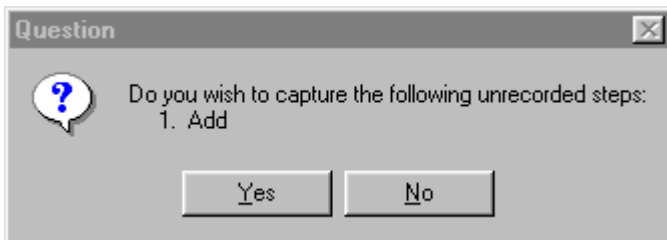
In VisualWorks 2.5 or 3.0:


```
StmCalculatorView open
```

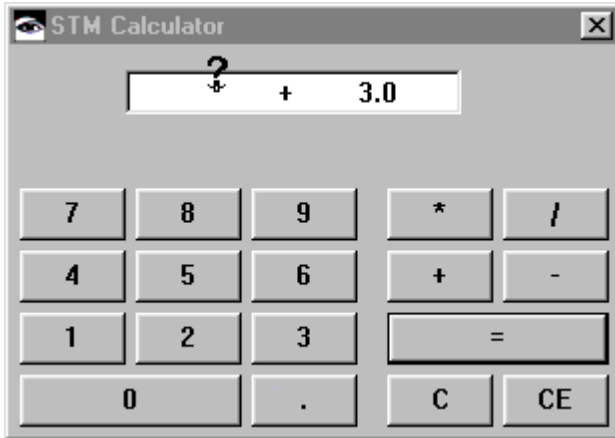
In VisualWorks 5i.4 or greater:

```
SilverMarkCalculator.StmCalculatorView open
```

5. Select the scenario and press the Quick Run  button on the tool bar to run the scenario. You will see the following prompt:

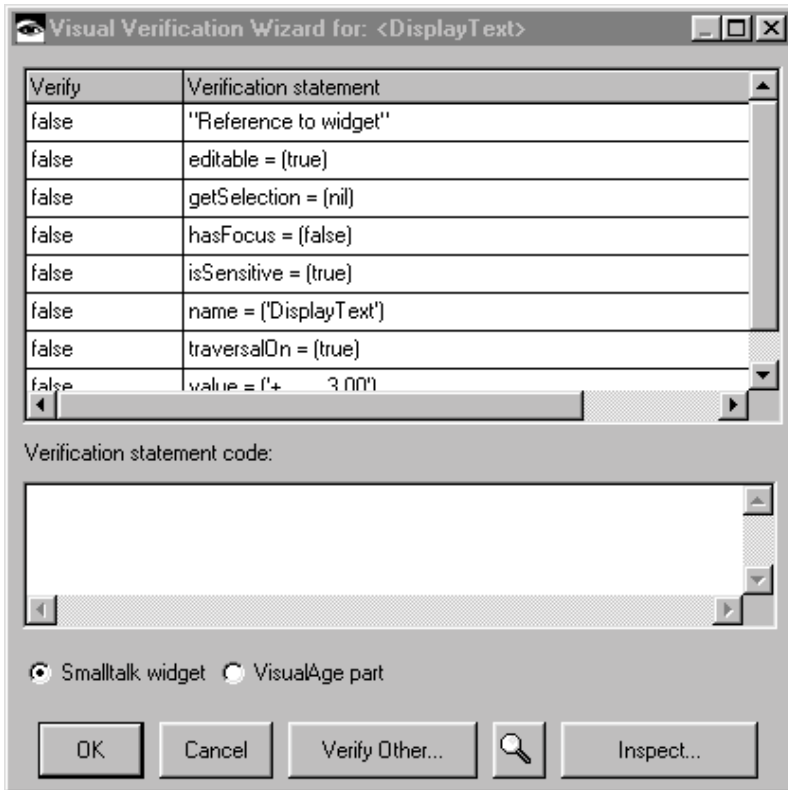


6. Answer Yes to continue.
7. The Test Editor window executes all steps up to the first UI recording step to be captured. For this example, this means executing the script step that opens the calculator. Once it reaches the first empty UI recording step, the Test Editor window selects that step and undergoes a transformation as described in the previous section titled *How to capture* user interface interactions during execution. At this point the Smalltalk Test Mentor is recording user interface interactions. Any that it detects will be added to the UI recording step on whose behalf the steps are being recorded. In this case, the step named **Add**.
8. To give it something to record, use the calculator keys to perform the add operation: **1 + 2 =**
9. At this point you can add UI verification steps to verify the state of the view under test.
10. Press the Verify UI  button on the Test Editor tool bar
11. Click over the result display of the calculator:

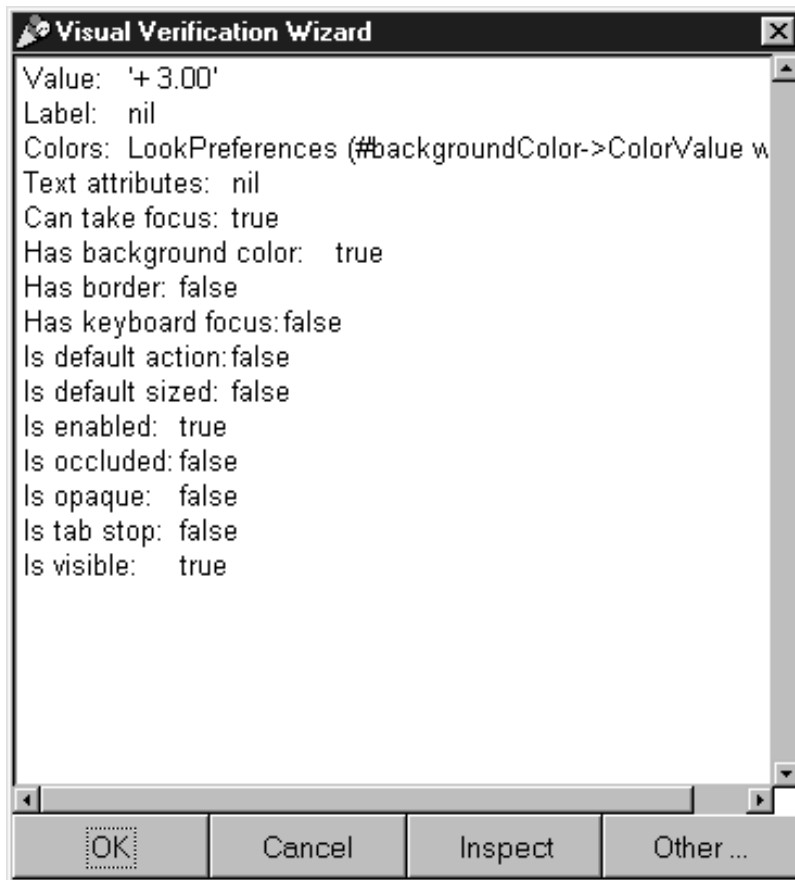


12. The following window appears:

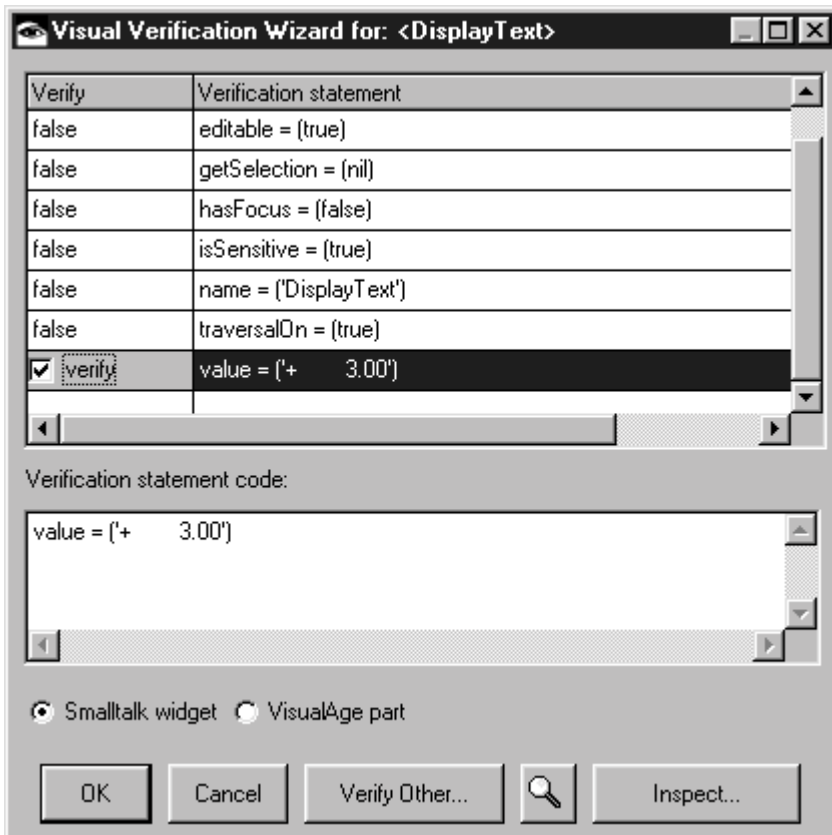
VisualAge:



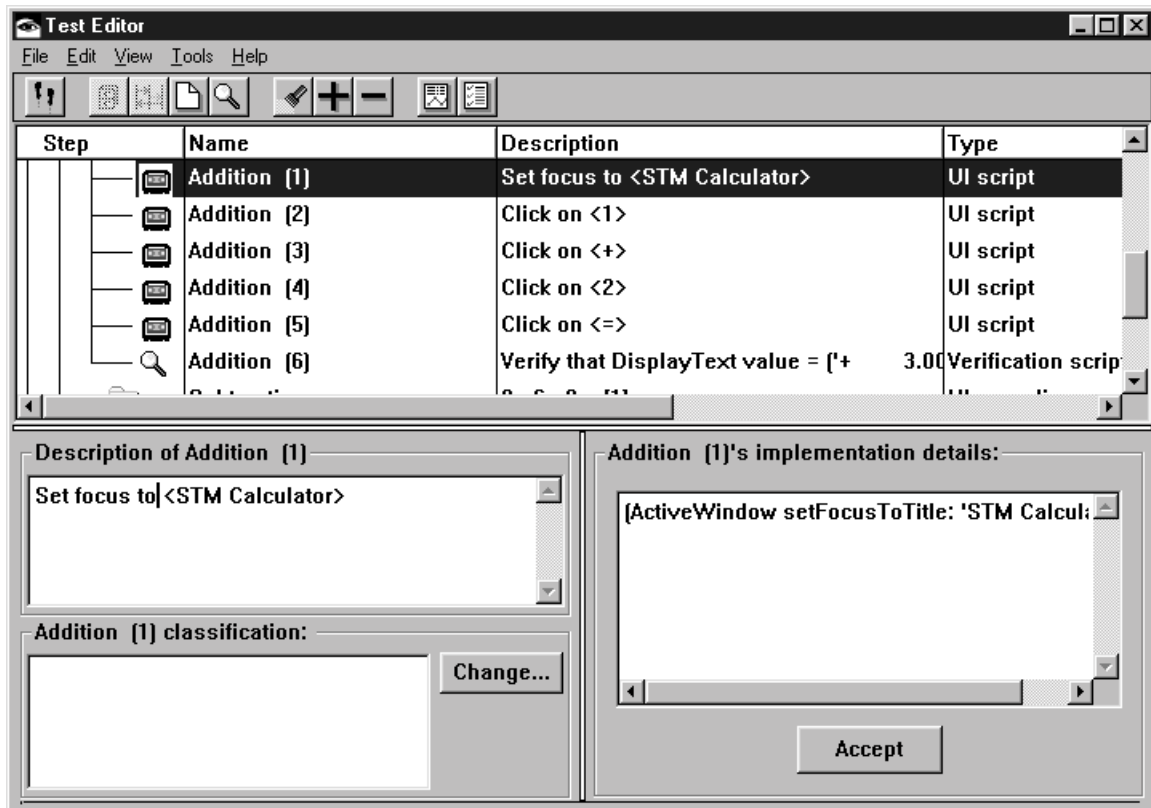
VisualWorks:



Check (VisualAge) or select (VisualWorks) the Verify cell whose corresponding Verification statement shows: **value = ('+ 3.0 ')**



13. Press OK.
14. You can continue recording more interactions with the view, and adding more verification steps as described here. For now, simply press Stop Recording. The Test Editor will look like this:



Notice that the step, **Add(6)** was added for the verification step. If you had checked more than one item to verify in the Verification Wizard window, more steps would have been created.

This concludes the example. You can add more verification steps or press Resume running to continue capturing.

How to view which widgets are in a window

You can use the Visual Verification Wizard to view a list of widgets owned by a window or shell. The following example shows you how. You will need to load the Smalltalk Test Mentor Examples configuration map or STM-Calculator parcel to try it yourself.

1. Open the Test Editor
2. Open the example calculator. The easiest way to do this is to execute

In VisualAge:



```
StmCalculatorView new openWidget.
```

In VisualWorks 2.5 or 3.0:

```
StmCalculatorView open
```

In VisualWorks 5i.4 or greater:

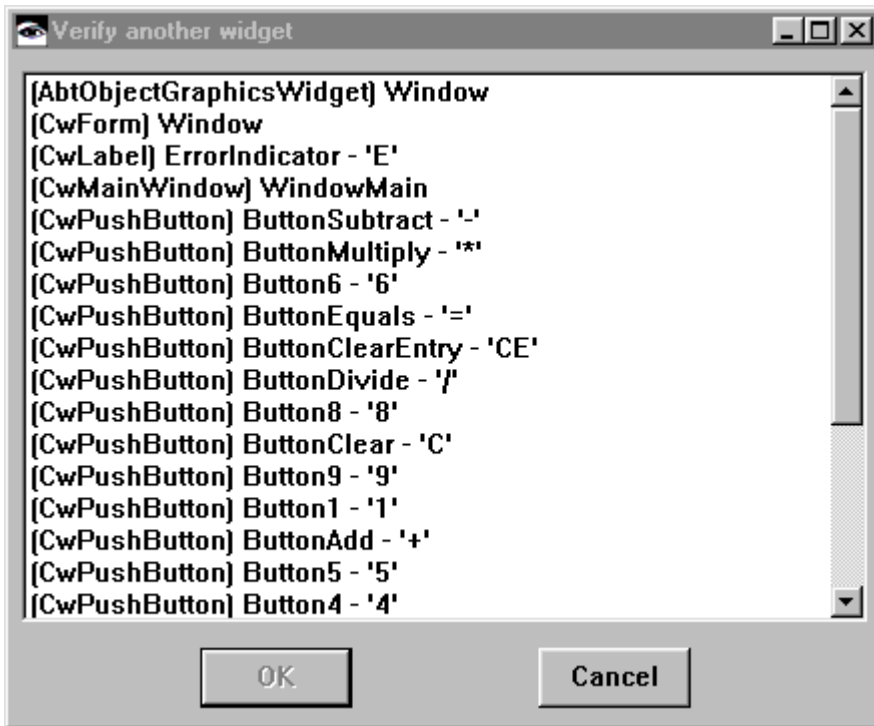
```
SilverMarkCalculator.StmCalculatorView open
```

3. Press the Verify UI  button in the tool bar and with the mouse pointer in the **?** (VisualAge) or  (VisualWorks) mode, click on any widget in the calculator.

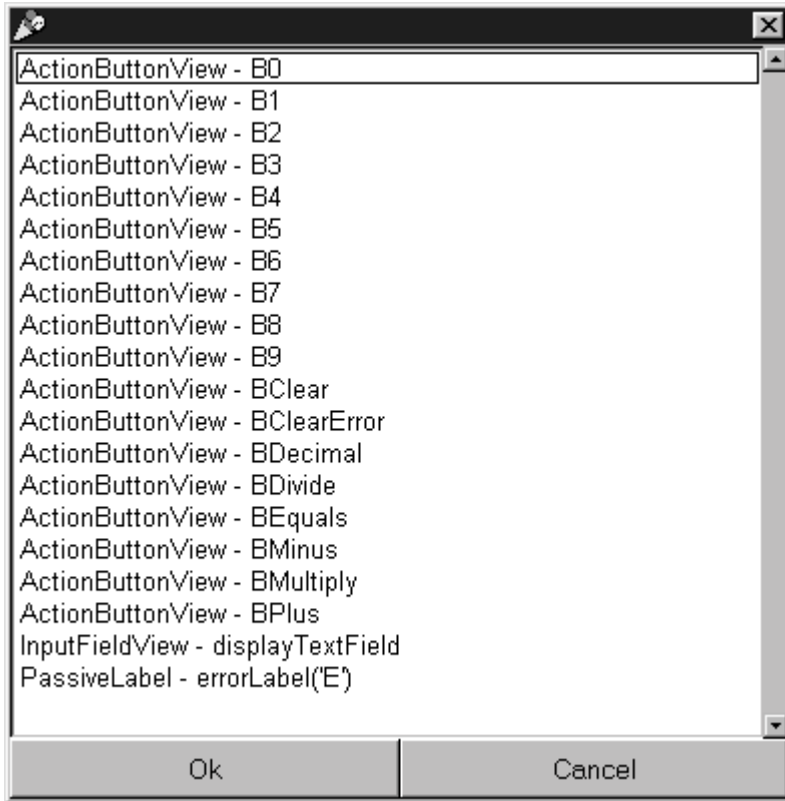
4. Press Verify Other...

A list of all the widgets in the window appears:

VisualAge:



VisualWorks:



5. This is a list of all the widgets in the Calculator view. Select any of them and press OK to view attributes for the selected widget that may be verified. You can also open a Smalltalk Inspector on the widget from the Visual Verification Wizard window.

How to apply a test data file to a set of recorded user interface interactions

You can embed steps within a file iteration step to cause those steps to be executed once for each row in the file. Each time a row is read, it is parsed to extract the names and values of the data items contained in the row, if any. For a description of how to structure a test data file, see the section in the *Smalltalk Test Mentor User Reference* on the *File Iteration step*.

When a data item is read, its value is assigned to a variable using the #varNamed:put: protocol. All you need to do is reference its value by name using #varNamed: within your step. For steps that perform user interface automation, you should replace the direct reference to literal variable values with references to variables.

For example, consider a user interface recording for a customer view that has generated code to fill in a name text widget:

```
(ActiveWindow widget: 'NameField')
    setValue: 'Munster, H.'
```

1. Create a data file that contains rows like the following:

```
Name=Munster, H.@Address=1313 Mockingbird lane@Weight<N>=324@Height<N>=120@Gender<C>=M@Smoker<B>=true
```

2. For each step that sets a value in the customer view, change the script to reference a variable as set by reading the file, like this:

```
(ActiveWindow widget: 'NameField')
  stmSetValue: (self varNamed: 'Name')
```

Running Tests

How to view existing tests

You can view tests with either the Test Editor or the Test Browser. Each presents a view of the suites that are present in the image. You can expand suites to show their component scenarios and you can expand scenarios to show their component steps.

If you are using the *Smalltalk Test Mentor - Runner* product, you will not have access to the Test Editor.

How to open the Test Browser

The Test Browser can be opened from either the System Transcript or the VisualAge Organizer.

- To open the Test Browser from the System Transcript, navigate to the Smalltalk Tools menu in VisualAge or the ENVY menu in VisualWorks and select SilverMark's Test Mentor Browser...
- To open the editor from the VisualAge Organizer, navigate to the Options menu and select Smalltalk Test Mentor Browser....


How to run a test

You run tests by first selecting the steps you wish to execute⁶ and then by launching the appropriate test runner.

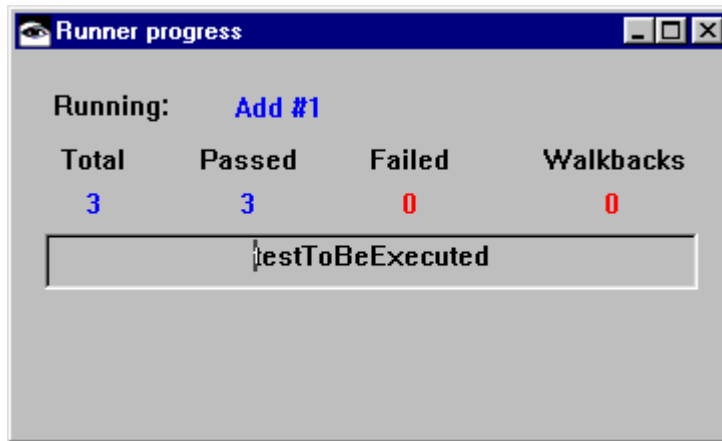
There are two types of test runner, If you do not need to pause steps or perform any debugging, you can use the Quick Runner. Otherwise, you should use the Test Runner.

How to use the Quick Runner

The Quick Runner is a simple view of current execution status that is displayed while steps are executed. To run a test using the Quick Runner, perform the following:

1. Select the steps you wish to run from either the Test Editor or the Test Runner
2. Perform one of the following:
 - Select the Quick Run menu item from the Tools pulldown menu.
 - Select the Quick Run menu item from the list view popup menu
 - Press the Quick Run  button on the tool bar.
3. The current execution status is shown in the following window as steps are executed

⁶ In VisualAge, the steps are executed *in the order in which you select them*



Note: If you are using a reasonably fast processor, some items will update too rapidly to see all values.

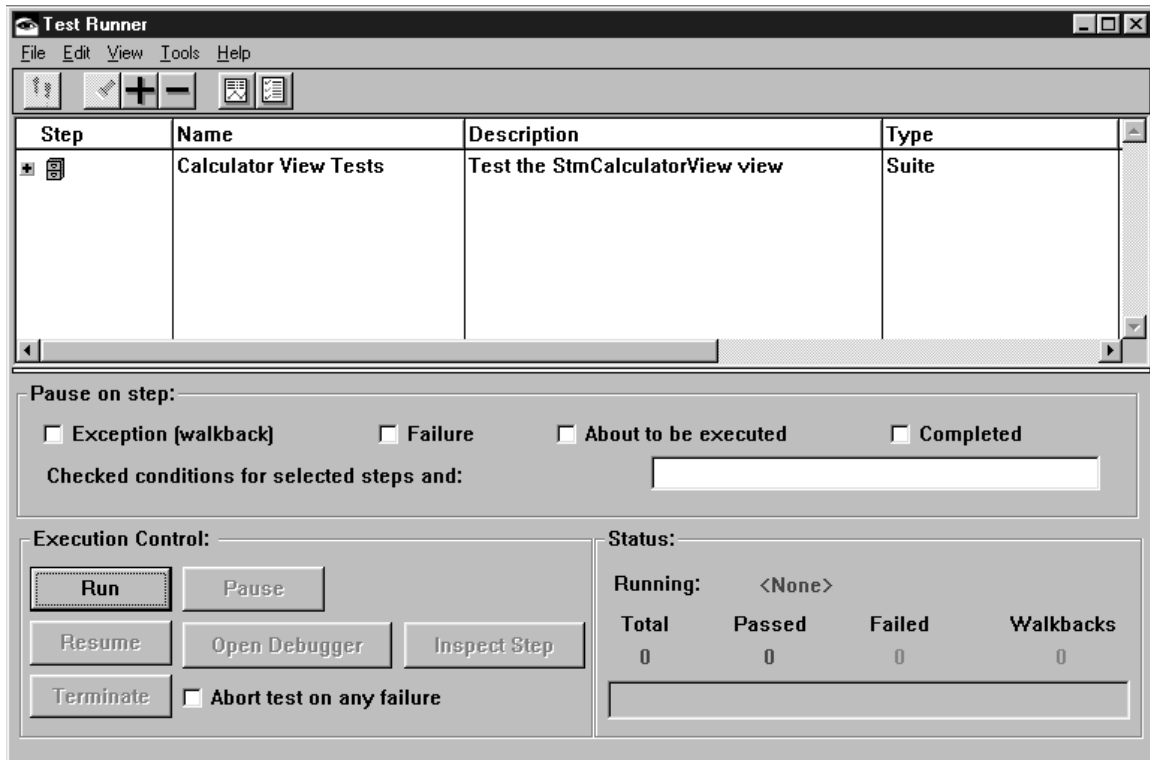
4. Once the execution of the last step has completed, the Test Results Browser is shown, with the results of the run. You can read about the Test Results Browser in the section under the heading, *Results Analysis*.

How to use the Test Runner

Like the Quick Runner, the Test Runner allows you to execute one or more steps, but it adds controls to allow you to pause and resume the execution of the steps. You may want to load the Smalltalk Test Mentor Examples configuration map or STM-Calculator parcel to follow the example.

To run a test using the Test Runner, perform the following:

1. Select the steps you wish to run from either the Test Editor or the Test Runner. As an example, select the **Calculator View Tests** suite.
2. Perform one of the following:
 - Select the Open Runner... menu item from the Tools pulldown menu.
 - Select the Open Runner... menu item from the list view popup menu
3. The Test Runner opens with the steps you selected in its list of steps to be executed.



4. To execute the steps, press the Run button.
5. The current state of the execution of the steps is displayed within the Status group box.
6. Once the execution of the last step has completed, the Test Results Browser is shown, with the results of the run. You can read about the Test Results Browser in the section under the heading, *Results Analysis*.

Note: If you run the same test from an open Test Runner, and have not closed the Test Results Browser, the execution results are added to the same, open Test Results Browser, rather than opening a new one.

To learn how to use the execution controls in the Test Runner, please refer to *How to pause and resume the execution of a test*, and *How to terminate execution of a test*.

How to stop a test from executing

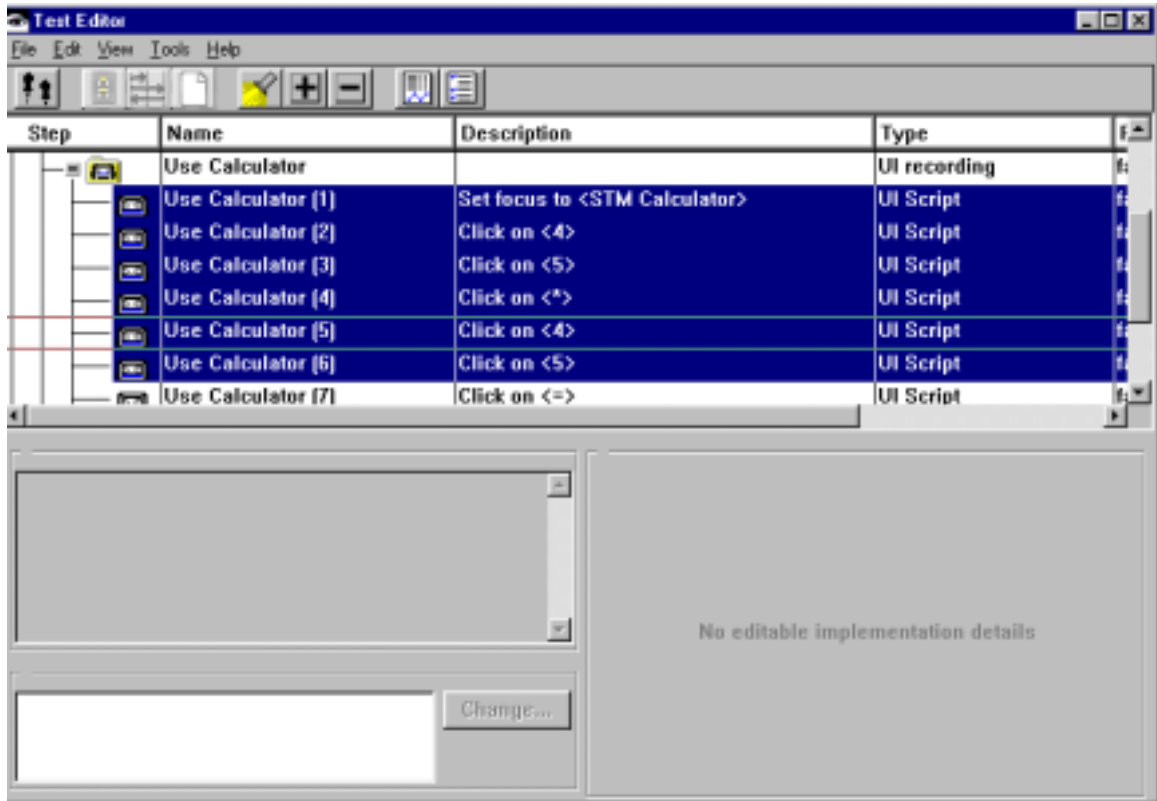
If you are using the Quick Runner, simply close the Status View. If you are using the Test Runner, press the Terminate button. When you close either the Status View or the Test Runner view while a test is running, the Test Results Browser opens on the steps that were executed up to the point at which the test was terminated.

How to run steps in a particular order (*VisualAge only*)

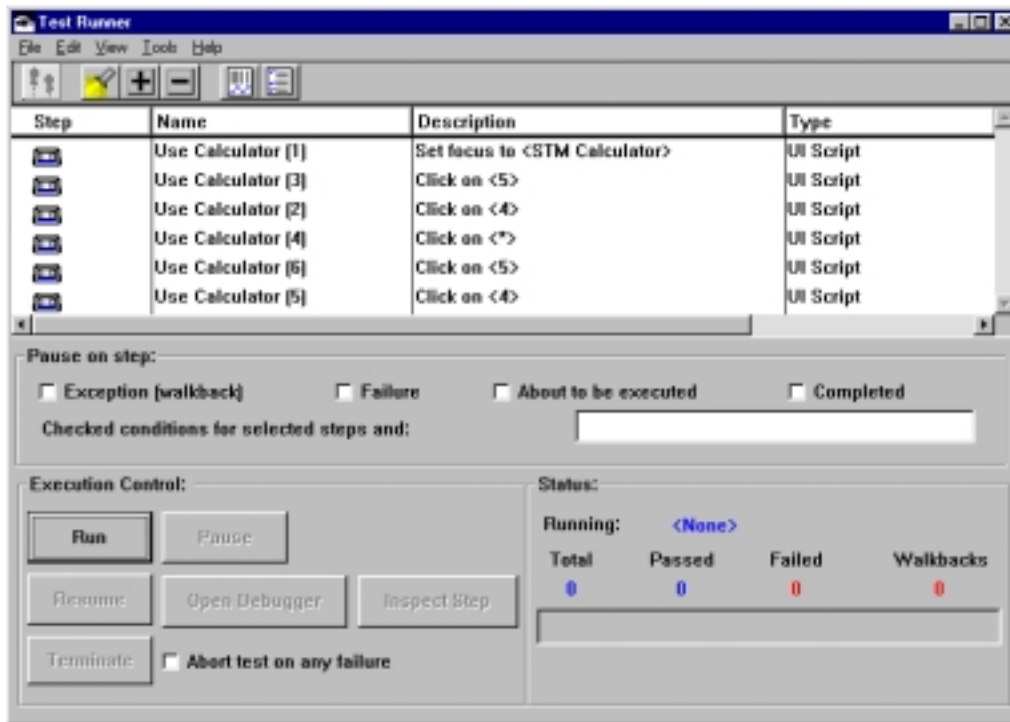
Steps are run in the order in which you select them from within the Test Editor or Test Runner. To see this, try the following example (you will need to have the Smalltalk Test Mentor Examples loaded):

1. Open the Test Browser (see *How to Open the Test Browser*).
2. Expand the Calculator View Tests Suite
3. Expand the Single Calculation
4. Expand the Use Calculator step
5. Multi-select the following steps in the given order:

- a. Open View
- b. Use Calculator (1)
- c. Use Calculator (3)
- d. Use Calculator (2)
- e. Use Calculator (4)
- f. Use Calculator (6)
- g. Use Calculator (5)



6. Select the Open Runner... item in the Tools menu.
7. In the Test Runner, you see the steps arranged in the selected order such that the steps will cause **54** to be multiplied by **54** rather than **45** by **45**:



How to pause and resume the execution of a test

You can use the Test Runner to control the execution of steps by pausing execution on various conditions of step execution. The conditions you can pause on are:

- Step about to be executed
- Step execution complete
- Step execution failure
- Step execution exception

You can pause any one or more of the above conditions for any step or selected step.

What you can do when a test is paused

When a step is paused, you can inspect the step or open a Smalltalk debugger on the executing test's process. You open the Smalltalk Debugger by selecting Open Debugger. This is most useful when you want to trace the execution of a script step, UI script step or verification script step, or for viewing the stack when an exception has been detected.

When you open the Smalltalk debugger on a process whose step is about to be executed, you may have to step over several preparatory message sends, to reach the point where the step is actually executed. This varies depending on the type of step.

If you place a halt in your code and cause the test to pause on an exception, you will not be able to step out of the halt from the debugger, because the test exception handling code has already trapped it. To trace the execution of a step, use the Test Runner pause-on-condition settings to pause the test prior to the step of interest, and step through it using the debugger. This means that **you do not need to embed 'self halt' in your test code to debug it.**

If you encounter an exception while tracing through a step with the debugger, the Test Runner will not trap the error because exception handling is disabled while the test's process is being executed under the debugger. This means that if you pause prior to executing a step that contains a halt, you can step through the halt with the debugger. Again, you should not need to embed halts in your tests.

Note 1: Alternately, you can step directly through exceptions by checking the Open debugger on exceptions check box on the Misc. tab of the Preferences notebook.

Note 2: If you close the Smalltalk Debugger that you opened from the Test Runner, the test terminates.

Note 3: You cannot press Resume from the Test Runner while the Smalltalk Debugger that you opened from the Test Runner is still open. If you wish to resume while the Smalltalk debugger is open, use the debugger's Resume button. This will close the debugger and resume execution of the test.

Note 4: Not all steps are useful to step through with the Smalltalk debugger. Suites, scenarios, and any of the collection steps, are of least interest because they simply contain code to iterate over the steps they contain. You will get the most benefit out of stepping through script (including user interface and user interface verification script), instance method, class method, and workspace steps.

Note 5: The top-level steps displayed by the Test Runner list view are held internally by a collection step that is not normally exposed. You may encounter this step if you pause during execution. When you run a test, this collection step is the first step to be executed, and the last one to complete. Its name is **ALL STEPS IN RUNNER**.

You can also open a Smalltalk inspector on the current step while a test is paused. See *The Guide to Programming* for information on step attributes.

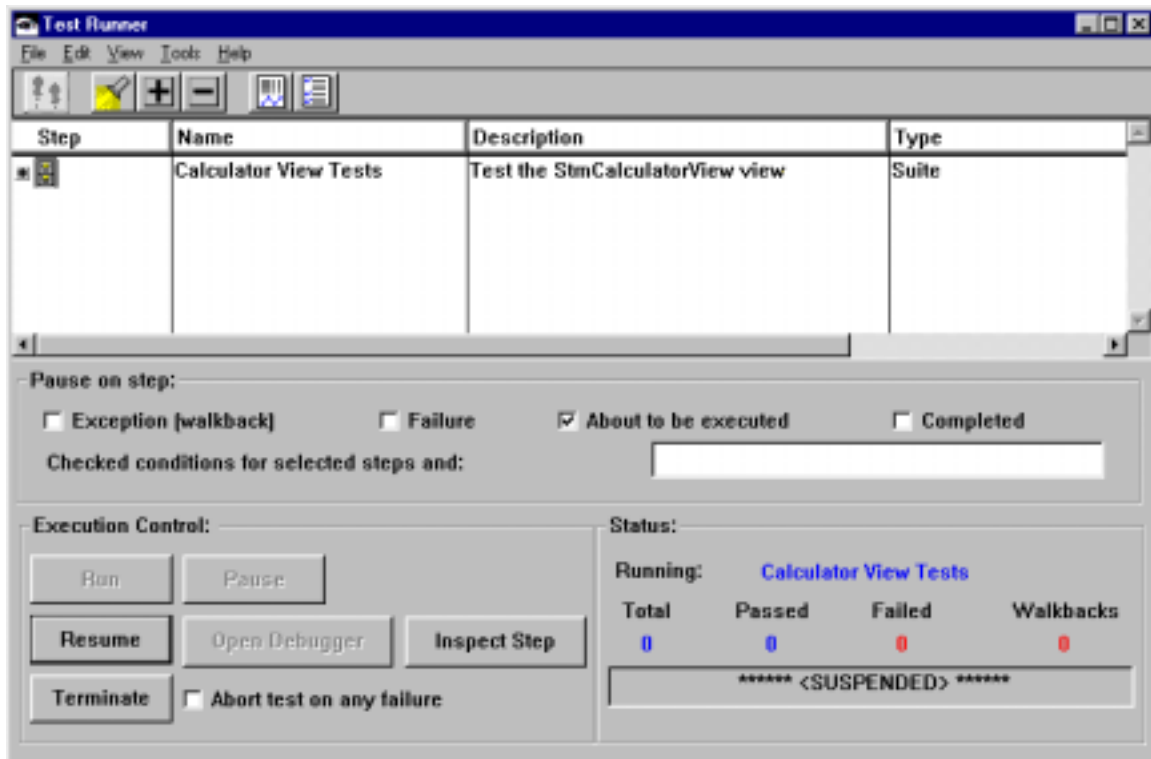
How to pause execution for any step

To pause execution on any step, make sure of following prior to pressing Run:

- No items in the Test Runner steps list are selected.
- The text field adjacent to the label, Checked conditions for selected steps and: is empty,
- One or more items in the Pause on step: group are checked

Try the following example. You will need to have the **Smalltalk Test Mentor - Examples** configuration map loaded.

- From the Test Editor or Test Browser, select the Calculator View Tests suite.
- Open the Test Runner by selecting the Open Runner... item in the Tools menu.
- Check the About to be executed check box in the Test Runner Pause on step: group.
- Press Run, then Resume. You need to press Resume this one time at the start to skip past a hidden step that holds all the steps in the runner view.
- You will see the following status appear:




Notice that only the Resume, Terminate, and Inspect Step buttons are enabled.

You can step through the rest of the steps in the suite by repeatedly pressing Resume.

How to pause execution for specific steps

To pause execution on specific steps, select one or more items in the Test Runner steps list, and/or enter the name of a step in the text field adjacent to the label, Checked conditions for selected steps:. Then check one or more items in the Pause on step: group prior, to pressing Run. You can also change these settings any time a test is paused.

Try the following example. You will need to have the Smalltalk Test Mentor Examples configuration map or STM-Calculator parcel loaded.

1. From the Test Editor or Test Browser, select the Calculator View Tests suite.
2. Open the Test Runner by selecting the Open Runner... item in the Tools menu.
3. Expand the list of steps by pressing the  button in the tool bar.
4. Select the steps named **Use Calculator (3)** and **Use Calculator (7)**.
5. Enter **Close Calculator** in the text field adjacent to the label, Checked conditions for selected steps.
6. Check the About to be executed check box in the Test Runner Pause on step: group.
7. Press Run and watch the status. The Test Runner will run the test up to the first item selected, then suspend execution.
8. Press Resume to continue execution to the next item selected.
9. Press Resume again to continue execution until the item entered in the text field (**Close Calculator**) is reached.
10. Press Resume again to complete execution.

Note: You can change any of the settings or selections described above at any time, whether steps are actively being executed or not. The changes will take place at the next execution state change.

How to terminate execution of a test while it is running

You can terminate the execution of a test while steps are being executed. To do this while steps launched from a Quick Runner are being executed, simply close the Quick Runner status window.

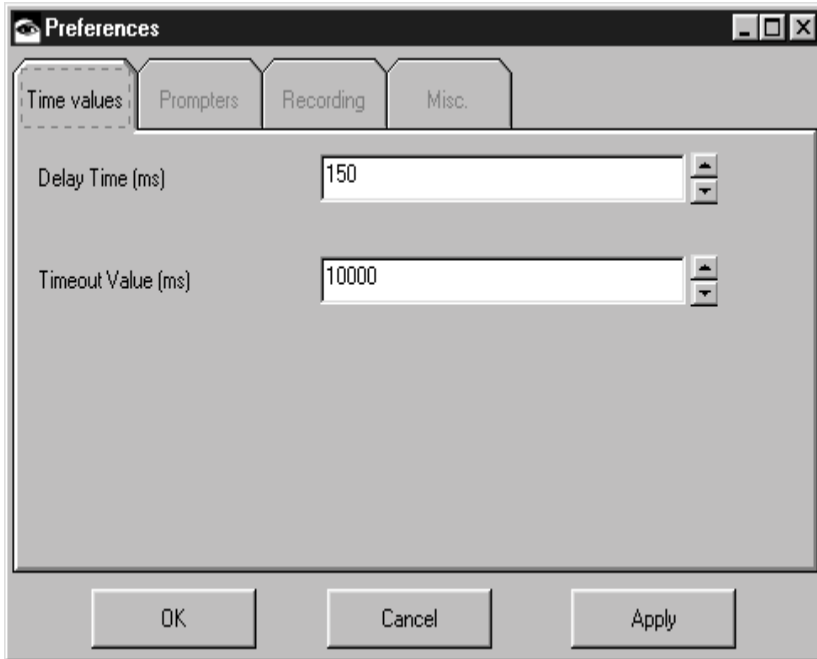
To terminate a test from the Test Runner, press the Terminate button while the test is executing or paused. You can then press Run again to begin execution of the steps from the beginning, or simply close the Test Runner window.

How to set delays and wait times

User interface steps have a built-in settling time delay for systems whose tests execute faster than the user interface can respond to automated requests.

Automated user interface interactions will also wait for widgets to become apparent if they cannot be found immediately. There is a maximum amount of time that the system will wait. Both these times can be set through the Preferences view.

1. To open the Preferences view, perform the following:
 - a. Select the Preferences... menu item from the Tools pulldown menu.
 - b. Select the Preferences ... menu item from the list view popup menu



All delay time values are in milliseconds.


Results Analysis

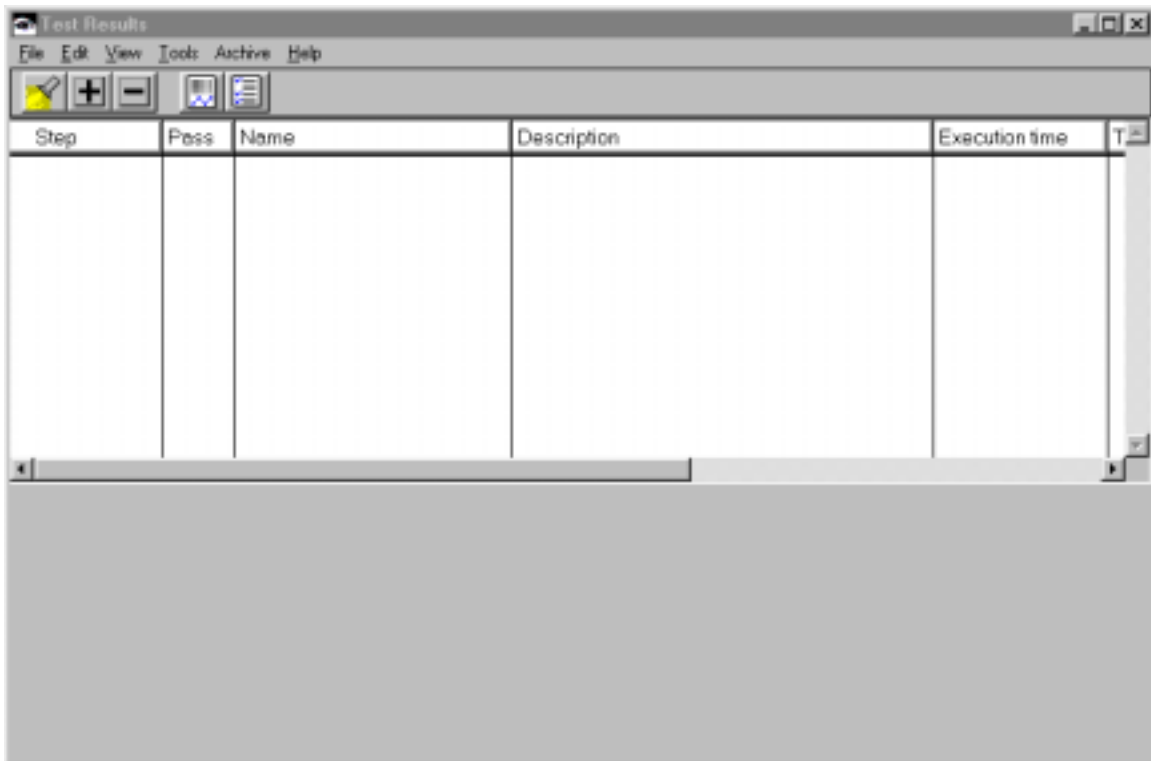
How to open a Test Results Browser

There are two ways that the Test Results Browser can be opened. The first, and most common, is automatically after the execution of a test. The second is to open an empty browser from one of the other Smalltalk Test Mentor views. The Test Results Browser can be opened from any of the following views:

- Test Editor
- Test Browser
- Test Results Browser

To open the Test Results Browser from one of these views, perform the following:

1. From one of the above views, perform one of the following:
 - Select the Results... menu item from the Tools pulldown menu.
 - Select the Results... menu item from the list view popup menu
 - Press the Results...  button on the tool bar.
2. An empty Results Browser is opened:



Because the browser has no results in it, you will have to load archived results to populate it. To find out how to do this, see *How to load Results from the results archive*.

How to calculate step statistics

You can use the Test Results Browser to perform a statistical comparison of selected steps. To do this, simply select several steps from the Test Results Browser. The result of the comparison is shown in the lower half of the browser.

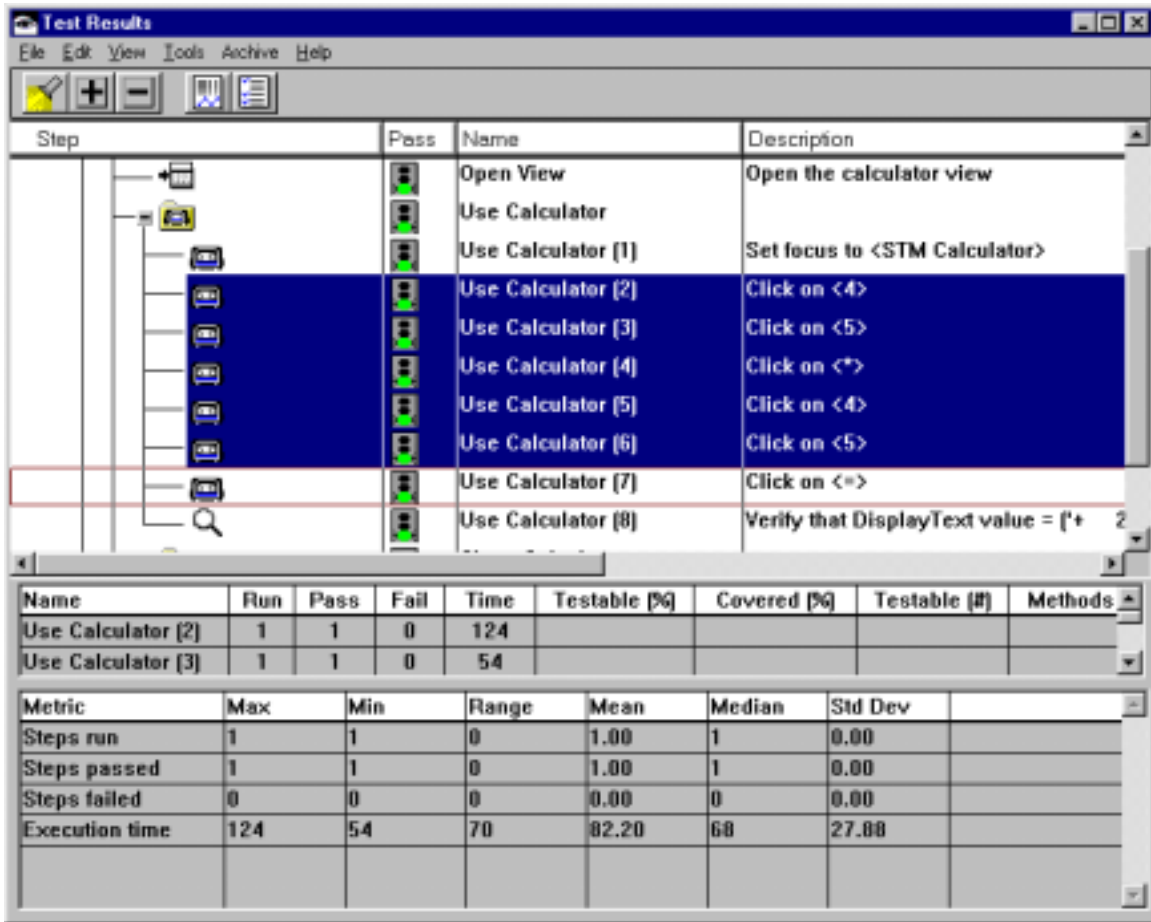
The screenshot shows the Test Results Browser window. The main table lists test steps with their execution times and types. Below it is a summary table for the test suite, and at the bottom is a metrics table.

Step	Pass	Name	Description	Execution time	Type
		Calculator View Tests	Test the StmCalculatorView view	2,653	Summary
		Single calculation	Perform a single, simple calculation and v	1,133	Scenario
		Open View	Open the calculator view	604	Interaction
		Use Calculator		355	UI
		Use Calculator (1)	Set focus to <STM Calculator>	37	UI
		Use Calculator (2)	Click on <4>	66	UI

Name	Run	Pass	Fail	Time	Covered (%)	Testable (%)	Methods (#)	Testable (#)
Calculator View Test	81	81	0	2,653	58.97	82.14	39	28
Calculator View Test	81	81	0	2,900	53.85	75.00	39	28

Metric	Max	Min	Range	Mean	Median	Std Dev
Steps run	81	81	0	81.00	81	0.00
Steps passed	81	81	0	81.00	81	0.00
Steps failed	0	0	0	0.00	0	0.00
Execution time	2,900	2,653	247	2,776.50	2,653	174.66
% Testable covered	82	75	7	78.57	82	5.05
% covered	59	54	5	56.41	59	3.63
Testable methods	28	28	0	28.00	28	0.00
Methods	39	39	0	39.00	39	0.00

The Test Results Browser makes no distinction between comparison of unrelated steps so you should be mindful of which steps you select. Typically you would compare the same suite or scenario executed against multiple versions of the system under test, however you may want to compare steps within the same scenario if they perform a similar function. For example, you may want to compare user interface steps within a scenario to characterize their execution times. The following shows a Test Results Browser with statistics for steps that enter values into the calculator under test shown.



How to compare steps for differences

You can compare any two steps and their sub-steps for differences. This is most useful when you want to rapidly determine if the results of a test execution differ substantially from that of a previous execution.

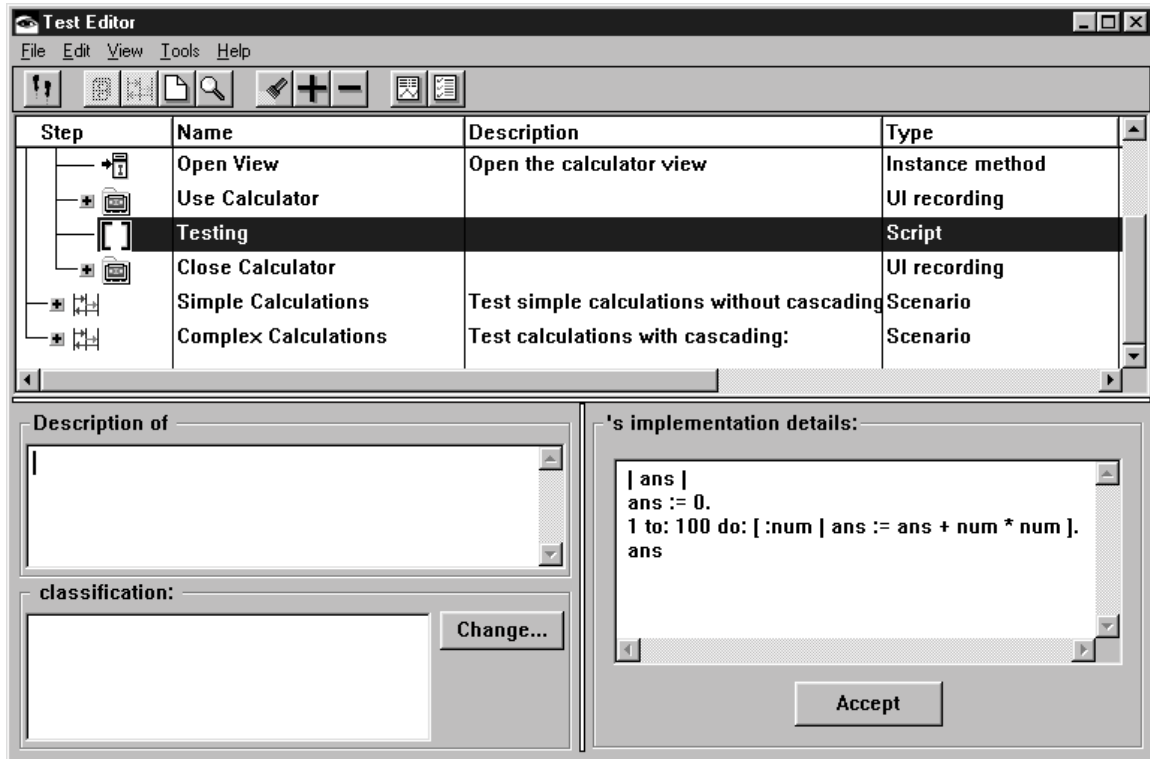
For example, to determine if a system under test has changed between test suite executions, perform the following (you will need to load the Smalltalk Test Mentor Examples configuration map or STM-Calculator parcel to follow this example):

1. Open the Test Editor
2. Expand the **Calculator View Tests** suite to show the steps in the **Single Calculation** scenario.
3. Add a script step named **Testing** between the **Use Calculator** and **Close Calculator** steps. You will use this step to vary the performance of the test, in order to have interesting values to compare.
4. Enter the following code for the script:

```
| ans |
ans := 0.
1 to: 100 do: [ :num | ans := ans + num * num ].
ans
```

5. Don't forget to press the A button.

The editor should look like this:



6. Select the **Single Calculation** scenario and select Open Runner... in the Tools menu open the Test Runner.
7. Press Run in the Test Runner to run the scenario.
8. Once the test completes, the Test Results Browser opens.
9. Now go back to the Test Editor and change the 'Testing' script to force an error in that step. The script should read like this:

```
self error: 'Oops'
```

10. Go back to the Test Runner and press Run to run the scenario.
11. Once the test completes, the new results are added to the already open Test Results Browser.
12. In the Test Results Browser, select the two results for the scenarios you just ran. You can see the different number of passed steps shown in the lower section of the Test Results Browser:

The screenshot shows a 'Test Results' window with a menu bar (File, Edit, View, Tools, Archive, Help) and a toolbar. The main area contains a table with the following data:

Step	Pass	Name	Description	Execution time
		Single calculation	Perform a single, simple calculation and v	121
		Single calculation	Perform a single, simple calculation and v	140

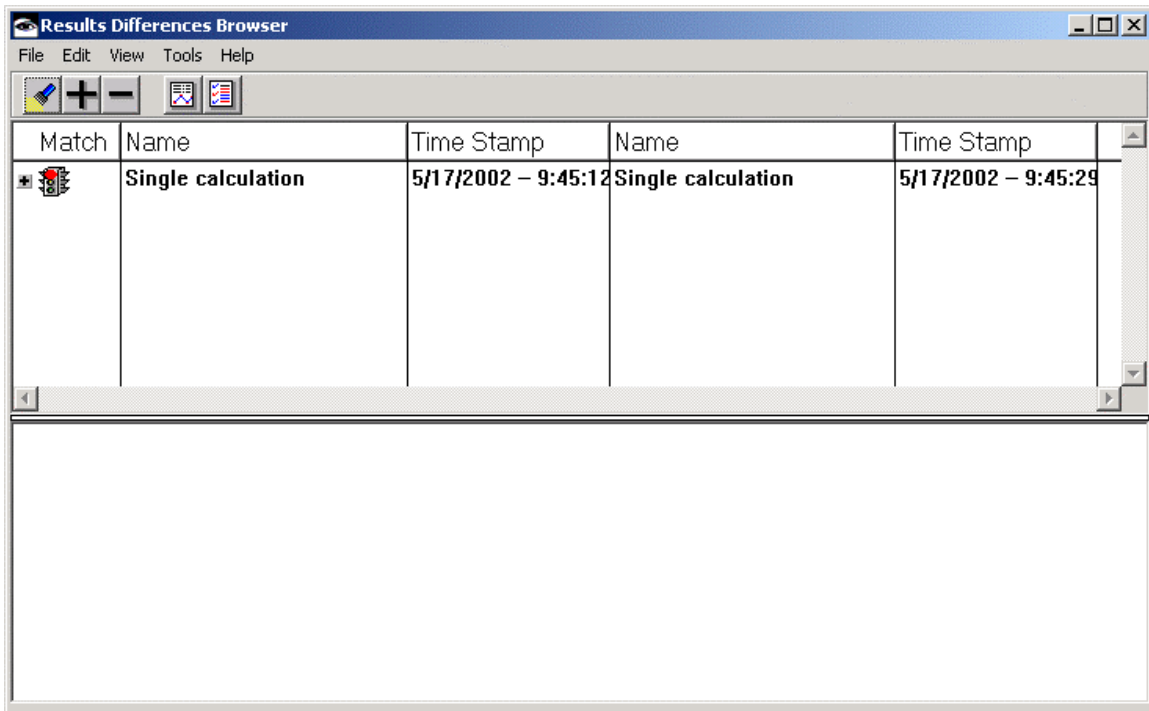
Below the main table is a summary table:


Name	Run	Pass	Fail	Time	Covered (%)	Testable (%)	Methods (#)	Testable
Single calculation	12	12	0	121				
Single calculation	12	11	1	140				


At the bottom is a metrics table:

Metric	Max	Min	Range	Mean	Median	Std Dev
Steps run	12	12	0	12.00	12	0.00
Steps passed	12	11	1	11.50	12	0.71
Steps failed	1	0	1	0.50	0	0.71
Execution time	140	121	19	130.50	121	13.44

13. Open the Results Differences Browser by selecting the Differences... item in the Test Results Browser Tools menu.
14. Enter **500** in response to the **Enter maximum allowable percent difference between numeric values** prompt, to specify that any numeric values within 500% of each other are acceptable, because for this experiment, execution time mismatches are not of interest.
15. The following Results Differences Browser is opened:



16. Each row shows the results of comparing two results items. The name and time stamp of each item is shown, as well as an icon to indicate whether the comparison was successful → , or

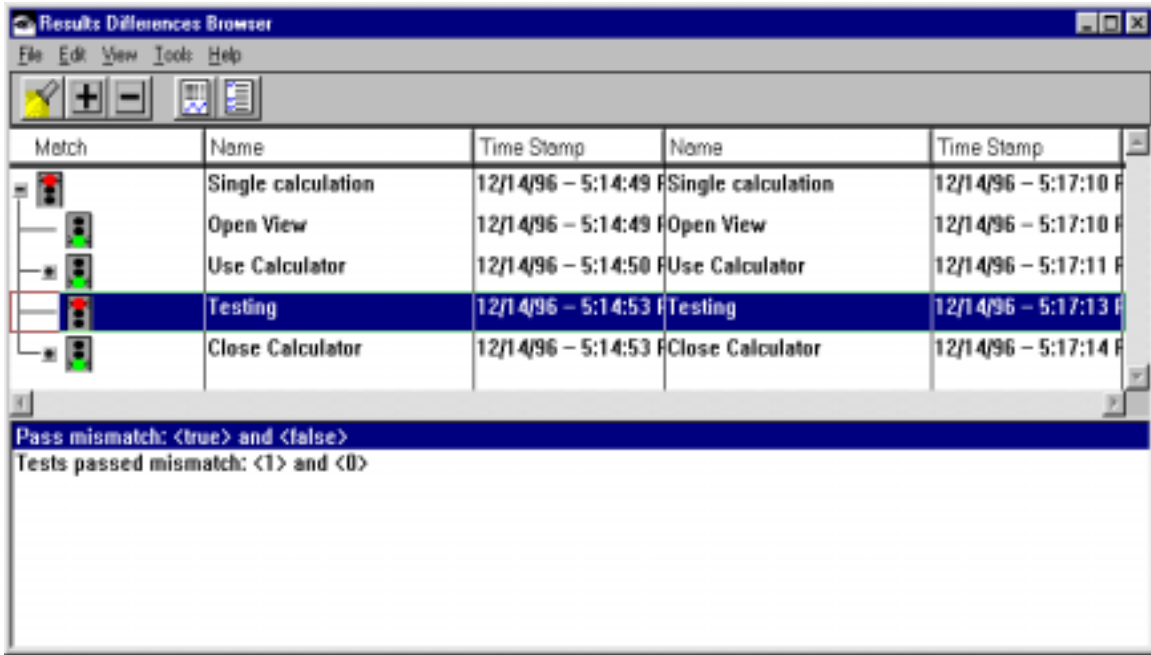
not → .

17. You see that the icon for the top-level comparison indicates failure. This means that there was a mismatch between the selected results or results that they contain were found. If you select the item, you see the following displayed in the lower section of the Results Differences Browser:

Pass mismatch: <true> and: <false>

18. This means that in comparing the two results for the execution of the **Single Calculation** step, one was found to have passed and the other to have failed. Thus, they failed the comparison by their *pass* values.

19. To find more mismatches, you can use the Find dialog to search. For this example, there are few enough items that you can simply expand the top-level item. If you do this, you can see that, as would be expected, the results for the Testing step did not match. This isn't surprising considering that you changed the script before running the scenario the second time.




You see that the comparison failed for two reasons:

- One result represented a passed step and the other represented a failed step
- One result represented a one passed step and the other represented zero passed steps.

How to analyze method coverage metrics

When you execute a suite that has one or more applications or namespaces listed for its Applications or Namespaces Covered property, method coverage metrics are automatically calculated. While the Test Results Browser is open after execution, you can use the Method Coverage Browser to view details of which applications, classes and methods within them were executed as a result of executing the suite.

To better understand what the Method Coverage Browser can do for you, try the following example. You will need to load the Smalltalk Test Mentor Examples configuration map or STM-Calculator parcel to do so.


1. From the Test Editor or Test Browser, select the **Calculator View Tests** suite.
2. Press the Quick Run  button on the tool bar to run the suite. Once execution of the suite completes, the Test Results Browser is opened.
3. Select the Coverage analysis... item of the Tools menu in the Test Results Browser. The Method Coverage Browser is opened:

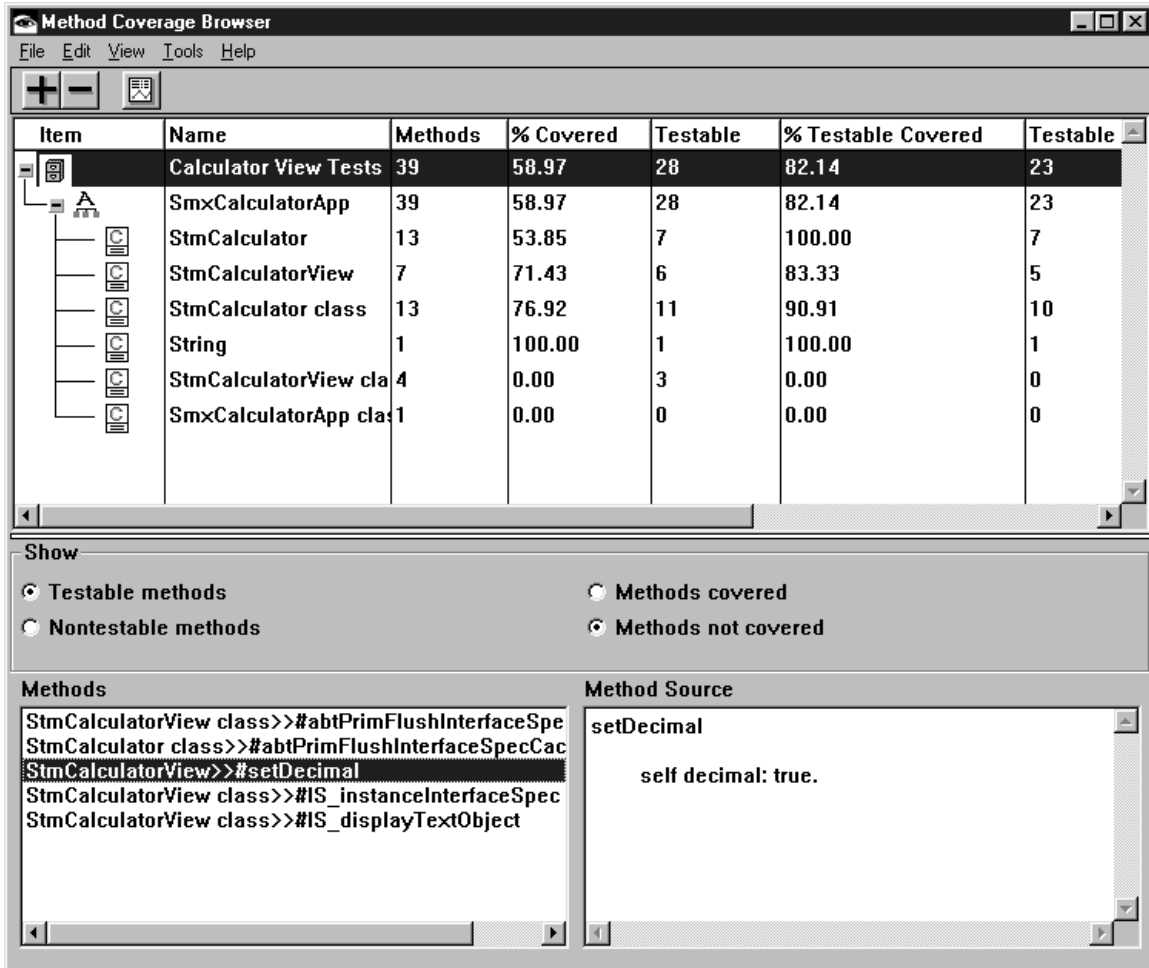
The screenshot shows the Method Coverage Browser application. At the top is a menu bar with 'File', 'Edit', 'View', 'Tools', and 'Help'. Below the menu bar is a toolbar with a '+' button (Expand), a '-' button (Collapse), and a document icon. The main area contains a table with the following data:

Item	Name	Methods	% Covered	Testable	% Testable Covered	Testable
	Calculator View Tests	39	58.97	28	82.14	23
	SmxCalculatorApp	39	58.97	28	82.14	23
	StmCalculator	13	53.85	7	100.00	7
	StmCalculatorView	7	71.43	6	83.33	5
	StmCalculator class	13	76.92	11	90.91	10
	String	1	100.00	1	100.00	1
	StmCalculatorView cla	4	0.00	3	0.00	0
	SmxCalculatorApp cla	1	0.00	0	0.00	0

Below the table is a 'Show' section with four radio buttons: 'Testable methods' (selected), 'Nontestable methods', 'Methods covered', and 'Methods not covered'. Below that is a 'Methods' list view showing several entries, with 'StmCalculatorView>>#setDecimal' selected. To the right of the 'Methods' list is a 'Method Source' window showing the code for the selected method:

```
setDecimal
    self decimal: true.
```

4. Try selecting the **Calculator View Tests** item and pressing the Expand  button. The applications that the suite is composed of and the classes within them are shown.
5. You can select the items in the list view to see details of coverage. You can also select whether you wish to view methods that were covered within the selected item, or methods that were not covered. Try selecting **StmCalculatorView** in the list view, then select the Methods not covered radio button, and then **StmCalculatorView>>#setDecimal**. You should see the following:



6. Because there is only one application tested by the example suite, this is not the most interesting example. Try the Method Coverage Browser on one of your suites.

How to store results into the results archive (ENVY only)

You can store the results of executing a suite in an archive for later viewing or comparison. To do this, simply select one or more suites in the Test Results Browser and select the Store results into archive item of the Archive menu. By doing this, you cause a persistent form of the selected suite or suites to be stored in the code library.

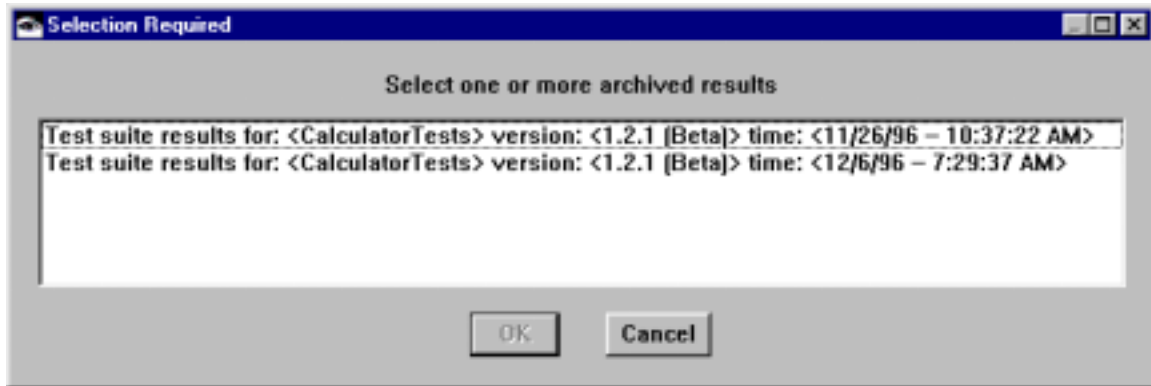
Note: You can only store results for a versioned suite. That means that the class in which a suite is defined must be versioned before the results of executing the suite can be stored in the archive. You can disable this restriction with settings on the Misc. tab of the Preferences notebook, but we do not recommend generally doing so.

How to store results into a file

You can store the results of executing a suite in a file for later viewing or comparison. To do this, simply select one or more suites in the Test Results Browser and select the Store results into file item of the Archive menu. When you do this you will be prompted for a file name.

How to load results from the results archive

You can load test results previously stored in the results archive by selecting the Load results from archive... item of the Archive menu, in the Test Results Browser. When you do this, you are prompted for which results to load:



Select the results you wish to load and press OK.

Note: In order to load a suite's results from the results archive, the version of the class in which the suite is defined must be loaded into the image first.

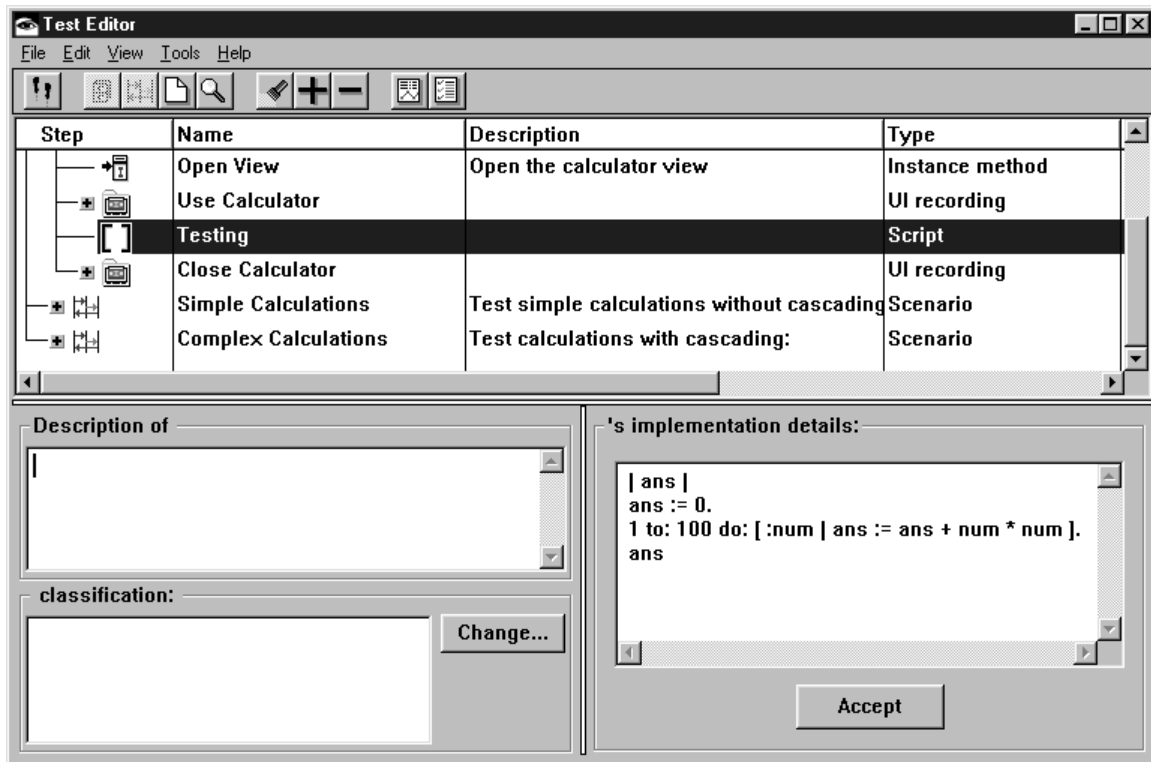
How to view results from an external tool

You can export results to an external tool such as Microsoft's Excel™. The following example shows how to do this. You will need to load the Smalltalk Test Mentor Examples and have access to Microsoft Excel to try the example yourself.

1. Open the Test Editor
2. Expand the **Calculator View Tests** suite to show the steps in the **Single Calculation** scenario.
3. Add a script step named **Testing** between the **Use Calculator** and **Close Calculator** steps. You will use this step to vary the performance of the test.
4. Enter the following code for the script:

```
| ans |  
ans := 0.  
1 to: 100 do: [ :num | ans := ans + num * num ].  
ans
```

5. Don't forget to press the Accept button.
6. The editor should look like this:



7. Select the **Calculator View Tests** scenario and select Open Runner... in the Tools menu to open the Test Runner.
8. Press Run in the Test Runner to run the scenario.
9. Once the test completes, the Test Results Browser opens.
10. Now go back to the Test Editor and change the **Testing** script to run for a longer period of time, pressing Accept to accept the change. The script should read like this:

```
| ans |
ans := 0.
1 to: 1000 do: [ :num | ans := ans + num * num ].
ans
```

11. Go back to the Test Runner and press Run to run the scenario.
12. Once the test completes, the new results are added to the already open Test Results Browser.
13. Now go back to the Test Editor and change the **Testing** script to run for an even longer period of time, pressing Accept to accept the change. The script should read like this:

```
| ans |
ans := 0.
1 to: 5000 do: [ :num | ans := ans + num * num ].
ans
```

14. Go back to the Test Runner and press Run to run the scenario.
15. Once the test completes, the new results are added to the already open Test Results Browser.
16. In the Test Results Browser, select the three results for the scenarios you just ran.

17. From the Test Results Browser select the Export Results... item from the File menu. A file dialog opens. Name the file **test.csv**. If you look at the file, it should have the same format as this:

```
Name, 'Calculator View Tests', 'Calculator View Tests', 'Calculator View Tests'
Run, 13, 13, 13
Pass, 13, 13, 13
Fail, 0, 0, 0
Time, 3156, 4054, 36971
Testable (%), 66.6666666666667, 66.6666666666667, 66.6666666666667
Covered (%), 38.0952380952381, 38.0952380952381, 38.0952380952381
Testable (#), 21, 21, 21
Methods (#), 12, 12, 12
-----< Statistics >-----

Metric ,Max,Min,Range,Mean,Median,Standard Deviation
Steps run, 13, 13, 0, 13.0, 13, 0.0
Steps passed, 13, 13, 0, 13.0, 13, 0.0
Steps failed, 0, 0, 0, 0.0, 0, 0.0
Execution time, 36971, 3156, 33815, 14727.0, 4054, 19269.1009909648
% Testable
covered, 66.6666666666667, 66.6666666666667, 0.0, 66.6666666666667, 66.6666666666667, 1.1012082
4 659276e-6
% covered, 38.0952380952381, 38.0952380952381, 0.0, 38.0952380952381, 38.0952380952381, 0.0
Testable methods, 12, 12, 0, 12.0, 12, 0.0
Methods, 21, 21, 0, 21.0, 21, 0.0
```

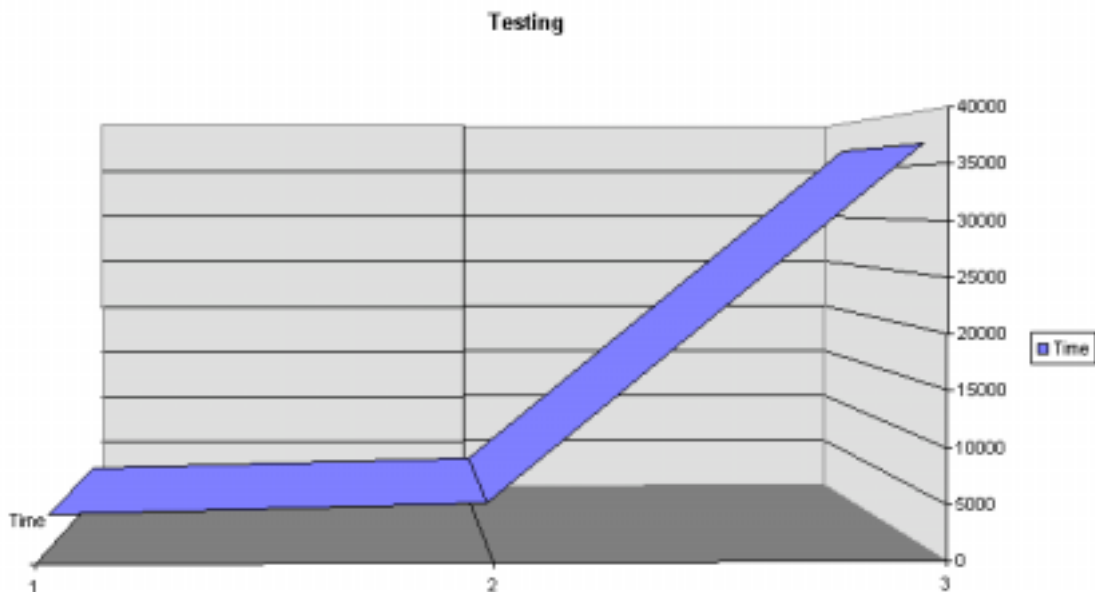
18. Now open Excel.

19. Use the Open... item of Excel's File menu to open a file prompter. Select **test.csv**.

20. Excel should show something like this:

	A	B	C	D	E	F	G	H	I
1									
2									
3	Name	'Calculator	'Calculator	'Calculator	View Tests'				
4	Run	13	13	13					
5	Pass	13	13	13					
6	Fail	0	0	0					
7	Time	3156	4054	36971					
8	Testable (%)	66.66667	66.66667	66.66667					
9	Covered (%)	38.09524	38.09524	38.09524					
10	Testable (#)	21	21	21					
11	Methods (#)	12	12	12					
12	-----< Statistics >-----								
13									
14	Metric	Max	Min	Range	Mean	Median	Standard Deviation		
15	Steps run	13	13	0	13	13	0		
16	Steps passed	13	13	0	13	13	0		
17	Steps failed	0	0	0	0	0	0		
18	Execution time	36971	3156	33815	14727	4054	19269.1		
19	% Testable covered	66.66667	66.66667	0	66.66667	66.66667	1.10E-06		
20	% covered	38.09524	38.09524	0	38.09524	38.09524	0		
21	Testable methods	12	12	0	12	12	0		
22	Methods	21	21	0	21	21	0		
23									

You can now use Excel's facilities to perform further analysis on your results. For example, you can use Excel's charting facilities to create a graph of execution time. To do this, simply select the items in row seven of the spreadsheet and use the Charting Wizard to create it. A detailed explanation of how to use Excel is beyond the scope of this document. Here's a possible graph of execution time:



How to load test results from a file

To load test results that have been stored in a file, simply perform the following:

1. Open a Test Results Browser (see *How to Open a Test Results Browser* for instructions on how to do this).
2. Perform one of the following:
 - Select the Load results from file... menu item from the Archive pulldown menu.
 - Select the Load results from file... menu item from the list view popup menu
3. Select a valid test results file from the file selection prompter.
4. The results are shown in the Test Results Browser. You can now store them to the archive for later analysis.

Testing strategies

How to design test cases

The purpose of this section is to show you how to design effective tests using the Smalltalk Test Mentor.

The number one item to remember is that you should design test cases as part of the design of the system you intend to test. If you are using a use-case driven design process, you will find that your test cases fall naturally out of your use cases. One of the worst mistakes an organization can make is to begin designing tests late in the development cycle. That's when testability and other problems begin to surface. This is also the time when schedule pressures are at their greatest and typically, testing is often compressed as a result.

Test Case Structure

Assuming you have designed a set of scenarios, the question then becomes how to implement them. The first item to consider is how the tests are to be structured. Using Smalltalk with Envy gives you several alternatives.

Structuring Tests by Adding Test Methods

By far the simplest way to structure test cases is to simply add test methods to the classes you intend to test. This has the distinct disadvantage of polluting your classes with test methods. You can use method categorization to differentiate test methods but there is still no way to prevent your test methods from being packaged with your runtime image.

Structuring Tests by Adding Extensions

One of the more powerful features of Smalltalk is the ability to extend classes with methods from different Applications, if you are using ENVY/developer, or Parcels if you are using VisualWorks without ENVY/developer. These additional methods are *under the control of an application or parcel other than the one in which the class is defined*. This is called extension. You can apply this technique to the structure of your tests. In general, it is best to create a test application or parcel separate from the application or parcel that controls the classes you wish to test. That way, you can exclude your test code from the code that is included in your deployed runtime image. Once you have set up a separate test application or parcel, you can then add test methods to your classes under test, but from within your test applications or parcels.

Using ENVY/developer

To add method extensions using ENVY/developer, select your test application in the Applications Manager. Next, you must give your test application visibility to the application containing the classes you wish to test. From the Applications Manager Prerequisites list, select the Change... menu item. Use the dialog that opens to add the application you wish to test to the prerequisites of your test application. Open a class hierarchy browser for the class you want to extend, and select the Add/Extension... menu item for the class. When prompted, select the application to extend the class into. If the application is not in the list, you will need to make sure the visibility is set correctly and that there is an open edition of the application. You can now add new test methods to your class under test.

In VisualAge you can also do this from the VisualAge Organizer when your test application is selected. First you must give your test application visibility to the application containing the classes you wish to test. From the VisualAge Organizer Applications menu, select Prerequisites. Use the dialog that opens to add the application you wish to test to the prerequisites of your test application. Then, from the Parts menu, select New... and then Extension... Enter the class you wish to add test methods to. You can now add new test methods to your class under test.

Using VisualWorks without ENVY/developer

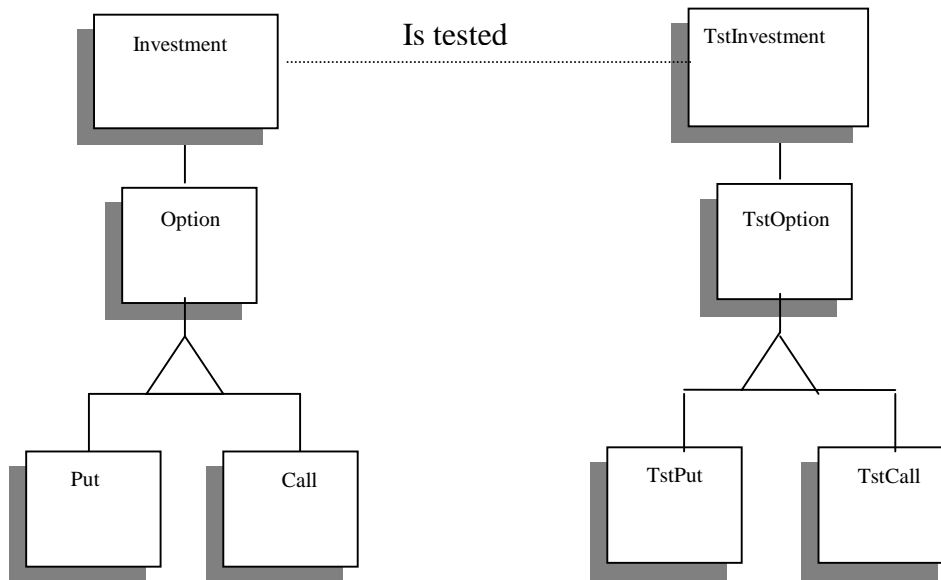
To add method extensions in VisualWorks without ENVY/developer, simply create a parcel for your tests, add then use the browsers to add the test methods to the parcel.

Structuring Tests by Adding Shadows

As shown above, you can use class extensions to add test methods without adding them to the body of methods that are ultimately shipped to your customers. You can use applications and parcels as a

mechanism for partitioning your test methods. Still, the above technique does not help you actually structure your tests in a logical manner. A major drawback to using extensions to add testing behavior is that you are limited to how much testing behavior you can actually add to your classes under test. You can not change the shape of a class under test by adding class or instance variables, or you end up violating the spirit of extensions. Also, if you are trying to reuse test case behavior through inheritance, you end up having to place that behavior at a very high level in the Smalltalk hierarchy so that all your classes under test can make use of that behavior. Creating *test shadows* can help overcome those limitations.

A test shadow is simply a class or application that shadows a class or application under test. A test shadow class contains all test specific code for testing a particular class. Typically test shadow classes are structured along the same or similar hierarchies as the classes they test.



By using test shadow classes, you are free to create a pure testing hierarchy under which all your test shadow classes can inherit. Of course your test shadow classes can also inherit testing behavior specific to their classes under test, following the way behavior in the classes under test is inherited.

Extensions then become the preferred mechanism for adding introspective behavior to classes under test that do not expose their internal states. For example, if you were trying to create a shadow test class for a class that has a **sum** instance variable, but that instance variable has no get method, you are free to simply create a get method for the class under test in the shadow test class's application, as an extension to the class under test.

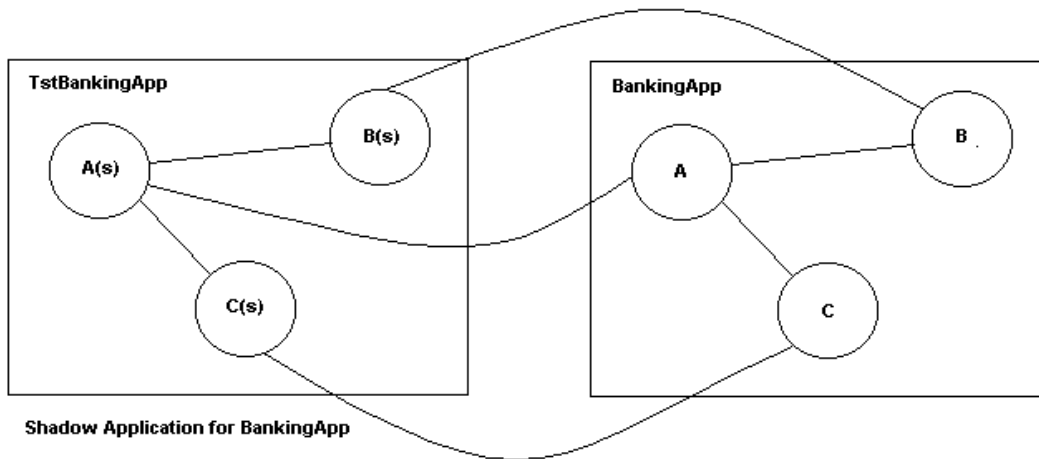
The main disadvantage to using shadow test classes is that they typically require more maintenance than either of the above mechanisms. If your system's hierarchy changes, the shadow class's hierarchy must change along with it. Also, there are simply more classes in the image to manage.

Shadow Classes and the Smalltalk Test Mentor

You should create your test shadow classes as subclasses of `StmSuite`. In fact, when you create a suite for the purpose of testing a class within your system, you are creating a test shadow class for your class under test. As you build up your hierarchy of test suites, you build your hierarchy of test shadows. A suite then contains one or more scenarios to implement use cases for the class under test.

Cluster testing

Few, if any, non-trivial systems are composed of instances of a single class. Systems are typically composed of clusters of collaborating classes, often called subsystems. The shadowing technique is still valid under this circumstance, but the scope of the cluster shadow is considerably larger than that of a single class. The cluster shadow class's responsibility is to orchestrate interactions with the component shadow classes. Because many ENVY/developer applications and VisualWorks parcels contain clusters themselves, it is often simplest to create a shadow of an application to orchestrate tests between shadows of its component classes. This is shown in the following diagram:



How to reuse tests

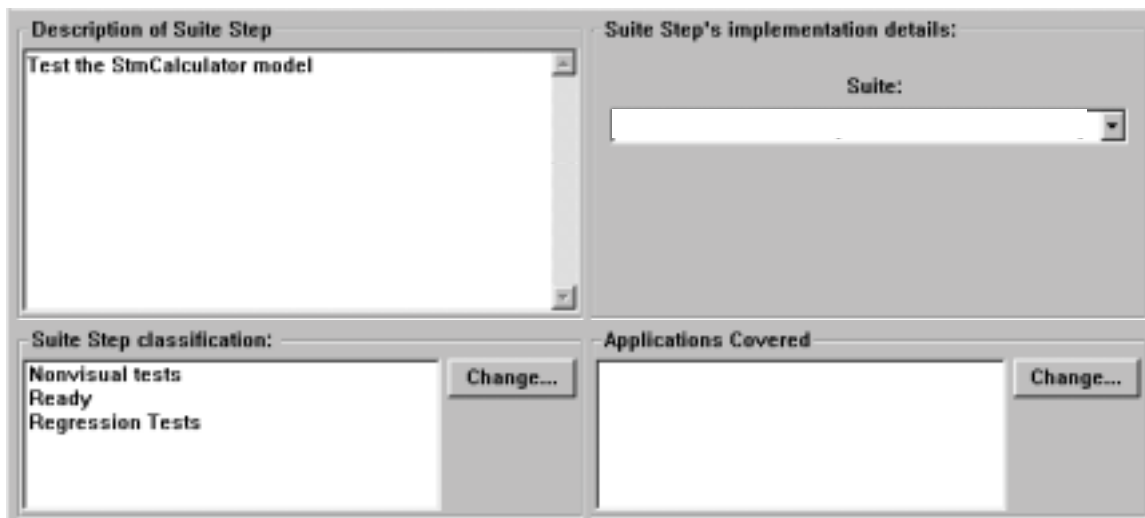
The Smalltalk Test Mentor supports reuse through reference to reusable units and inheritance.

Reuse through reference

There are two units of reusability that can be referenced within tests - the suite and the scenario. You can embed references to suites and scenarios within scenarios in the form of suite steps and scenario steps that reference suites and scenarios, respectively. For example, if you are testing a system that requires a user to log in to it, you can create a login scenario that will stand on its own as a particular usage of the system login function. You can then reuse that scenario within other scenarios that require logging in to the system as part of setting up.

To create a step that references a suite, perform the following:

1. Either add a new step with a type of **Suite**, or change the type of an existing step to **Suite**.
2. When you select the suite step that you just changed or selected, you will see something like the following:



3. Use the drop-down list to select a suite. When this step is executed, it will cause all scenarios within the specified suite to be executed.
4. To create a step that references a scenario, perform the following:
5. Either add a new step with a type of **Scenario**, or change the type of an existing step to **Scenario**.
6. When you select the scenario step that you just changed or selected, you will see something like the following:



7. Use the Suite drop-down list to select a suite and the Scenario drop-down list to select a scenario within that suite. When this step is executed, it will cause the selected scenario within the specified suite to be executed.

Caution: If you need to share objects between steps that are executed under different suites and scenarios, use the object registration APIs as described in *How to keep track of objects under test*.

Reuse through inheritance

Because tests and their supporting structure are expressed as methods within suite classes, you can use subclassing to alter or override test behavior. For example, consider an abstract class that defines an interface for performing certain transformations, where those transformations are implemented within a hierarchy of concrete subclasses. You can create a single suite class that defines tests on those interfaces and then define subclasses of that suite class that provide the stimuli and verification implementation specific to their corresponding concrete hierarchy. In this way, you define the scenarios and steps only once, with specificity provided by supporting methods throughout the hierarchy.

You can also override the specifications for suites and scenarios themselves through subclassing. When you create a suite class as a subclass of another suite class, the new suite's behavior is identical to the suite it subclasses because it inherits all specifications from it. As soon as you make a change in the new suite, it generates its own method or methods to express the change. Once you do this, changes in the superclass suite may or may not be reflected in the subclass, depending on the extent of the code that was generated. You can learn more about which generated methods define which aspects of a test in the section titled *Smalltalk Test Mentor Programmer's Guide*.

How to use invariants

An invariant is a state within the system under test that is expected to never vary, regardless of the stimulus applied to the system. The Smalltalk Test Mentor allows you to specify special steps to be run after each scenario, as well as after each top level step within a scenario.


The classic example of using an invariant is in the case of the queue. Consider a test of a queue. You might add items and remove items but the size returned by a queue should never be less than zero. That is an invariant. You might implement a scenario for adding and removing queue elements, specifying a scenario invariant, in the following way. This assumes a suite named 'Queue Suite' defined in class QueueSuite.

1. Define a new scenario named 'Add & remove' in 'Queue Suite'.
2. Select the new scenario, if it isn't already selected.
3. Press the New Method button.
4. Enter **addRemoveInvariant** at the prompt, and press OK.
5. Press the Browse... button.
6. Enter the following code for the QueueSuite>>addRemoveInvariant method in the class browser.

```
^(self varNamed: 'queue') size < 0 not
```
7. Create three script steps:
8. Change the name of the first script to 'Create queue ' and enter the following text as code the script:

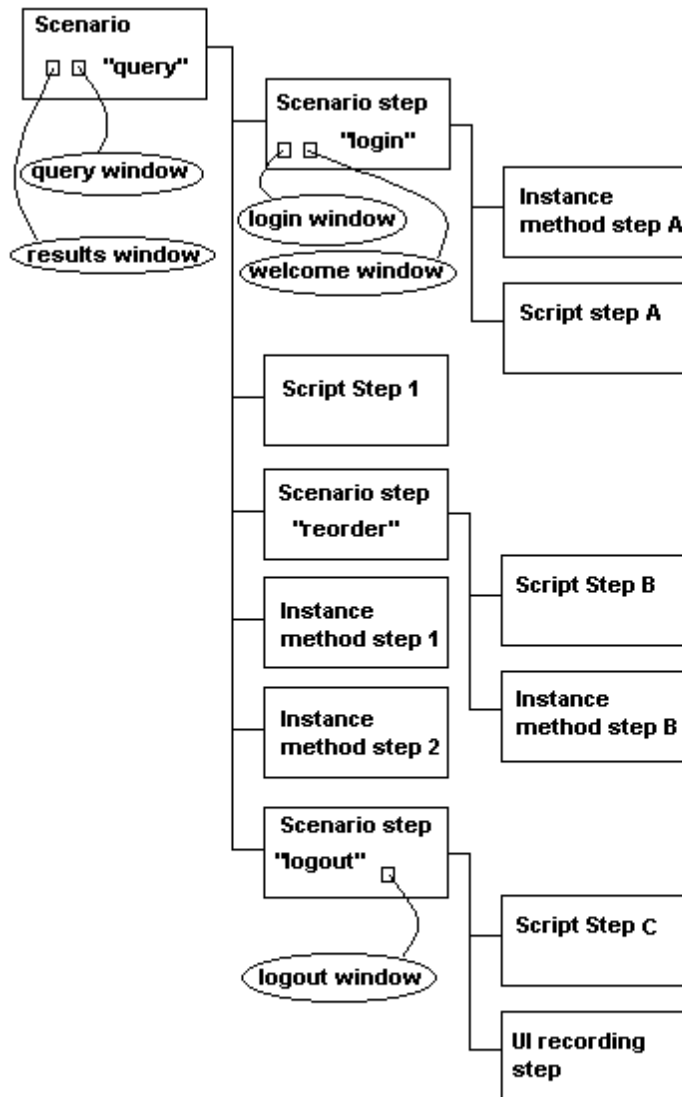
```
"Use OrderedCollection as a queue"  
^self varNamed: 'queue' put: OrderedCollection new
```
9. Change the name of the second script to 'Add ' and enter the following text as code the script:

```
^(self varNamed: 'queue') add: 'something'.
```
10. Change the name of the third script to 'Remove ' and enter the following text as code the script:

```
^(self varNamed: 'queue') remove: 'something'.
```
11. Select the 'Add & remove' scenario and Press the Quick Run  button on the tool bar.
12. Look at the results browser. Notice that the results of having executed an instance variable step with the name 'Add & remove_invariant' are inserted after each step. This is a result of your having specified the #addRemoveInvariant invariant selector for the scenario.

How to keep track of objects under test

When you reuse suites or scenarios during a test, steps within the test are executed under varying instances of suites or scenarios. If you use instance variables to hold objects under test, you will not be able to access all instance variables from all steps. Consider the following scenario:



This example consists of a scenario composed of several steps, three of which are scenarios as referenced by scenario steps. Remember that each suite or scenario operates under the aegis of its own instance of the class in which it is defined. This means that the three scenarios each operate as unique instances, separate from each other, regardless of whether they are defined within the same class.

For the purpose of explanation, assume that the **query** scenario needs to keep track of the **query window** and the **results window** objects under test. The **login** and **logout** referenced scenarios keep track of other windows as objects under test. The **reorder** scenario requires access to the **results** window but does not have direct addressability to it because that window is held by an entirely different instance.

To alleviate this type of restriction, the Smalltalk Test Mentor provides a well-known area for storing objects during test execution. You can use the interface to this area to store objects under a key,

independent of the instance of `scenario` or `suite` that a step is executing within. Use either `#var:named:` or `#varNamed:put:`⁷ to store an object and `#varNamed:` to retrieve it.

Continuing with the above example, the step that creates the **results window** might perform the following in order to make it accessible to any other step within the test.

```
self
  var: self queryWindow resultsView
  named: 'Query Results'.
```

This assumes that `#resultsView` is a message that when sent to an instance of the query window, returns the current results window.

If the **reorder** scenario needs to access the **results window** in order to perform its operations, it would use the following to access that object:

```
self varNamed: 'Query Results'
```

This would be performed from either a script step or an instance method step.

Note: This replaces `#registerObject:named:` and `#registeredObjectNamed:` provided in version 1.0 of the Smalltalk Test Mentor. The new selectors were provided in the interest in saving keystrokes, however, the old protocol is still supported.

How to design for step failures

The whole point of testing is to find faults in the system under test. If tests are not finding faults, either the system under test is fault-free, which is unlikely, or the test is incomplete. Given the assumption that faults will be found, you should design tests to handle them.

If the execution of a test produces a failed step⁸, all subsequent steps could be executed, the test could be immediately aborted, or steps may be executed to 'clean up' the test before completing. The Smalltalk Test Mentor supports all three possible outcomes.

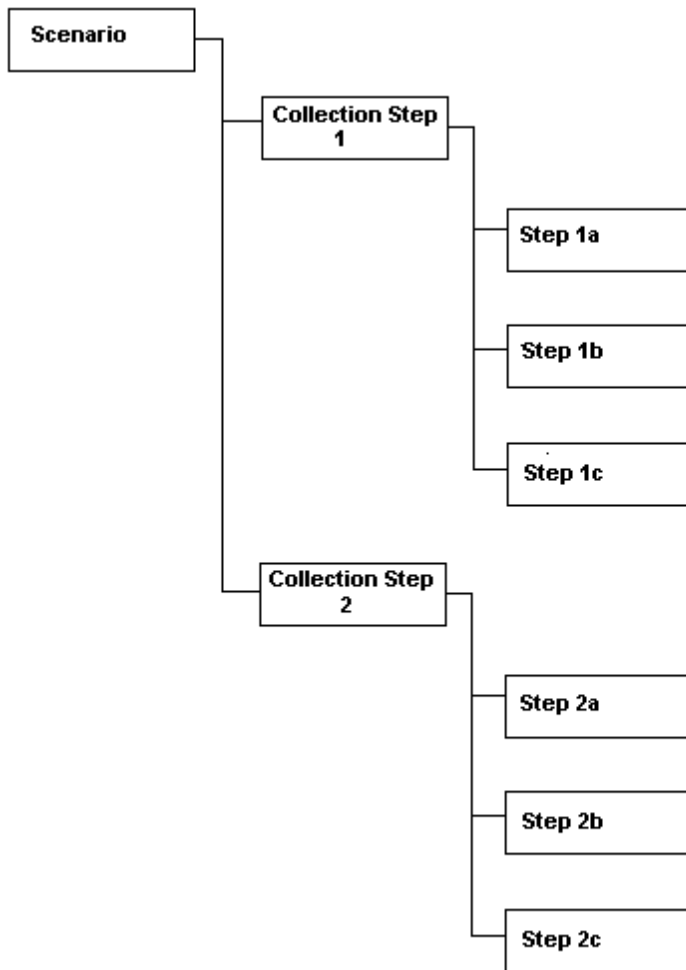
The *abort on fail*, and *precondition* attributes of steps are used to control execution on failures. The two mechanisms implied by those attributes differ in their levels of granularity. First a refresher on the meaning of *abort on fail* and *precondition* attributes.

An *abort on fail* step attribute value of *true* for a given step means that the given step will cease running any sub-steps if any one of those sub-steps fail. A *precondition* step attribute value of *true* for a given step means that the successful execution of the step is a precondition for the execution of any subsequent peer steps. A subsequent peer step is simply the next step after the given step that shares the same parent step or scenario.

Here are some examples based on the following configuration of steps, within a scenario.

⁷ Both `#var:Named` and `#varNamed:put:` perform identically. You should choose whichever one you feel most comfortable with. `#varNamed:put:`, although lengthier to type, is more consistent with the VisualAge subpart access protocol, if that sort of thing matters to you.

⁸ A failed step is a step that has found a fault. If one assumes that the purpose of testing is to find faults, then a failed step really represents a successfully located error in the system under test.



How to allow a test to proceed after a failed step

If you don't change the *abort on fail*, or *precondition* attributes of any steps from their default value of *false*, the test will run through all the steps regardless of the outcome of each step. Thus, using the default value of false guarantees that all steps will be run. The disadvantage is that your tests may end up running for much longer than necessary if there is no need to run after a failed step and one is encountered early in a test. The condition in the system under test that caused the first failed step may cause many more. Failures due to exceptions take a long time to process because the walkback stack has to be converted to a string and logged.

The following table shows the settings required to ensure all steps of the example scenario above are run, regardless of failures:

Step or scenario name	Abort on fail	Precondition
Scenario	false	false
Collection Step 1	false	false
Step 1a	false	false
Step 1b	false	false
Step 1c	false	false
Collection Step 2	false	false
Step 2a	false	false
Step 2b	false	false
Step 2c	false	false

How to abort a test after a failed step

For this example, assume that steps **1b**, **2a**, and **2b** all fail for one reason or another. Also assume the following attribute settings for the example steps:

Step or scenario name	Abort on fail	Precondition
Scenario	false	false
Collection Step 1	true	false
Step 1a	false	false
Step 1b	false	false
Step 1c	false	false
Collection Step 2	false	false
Step 2a	false	false
Step 2b	false	true
Step 2c	false	false

Because Collection **Step 1**'s *abort on fail* setting is set to *true*, any failure of its immediate sub-steps will cause the rest of its sub-steps to not be executed. In this case, step **1c** would not be executed.

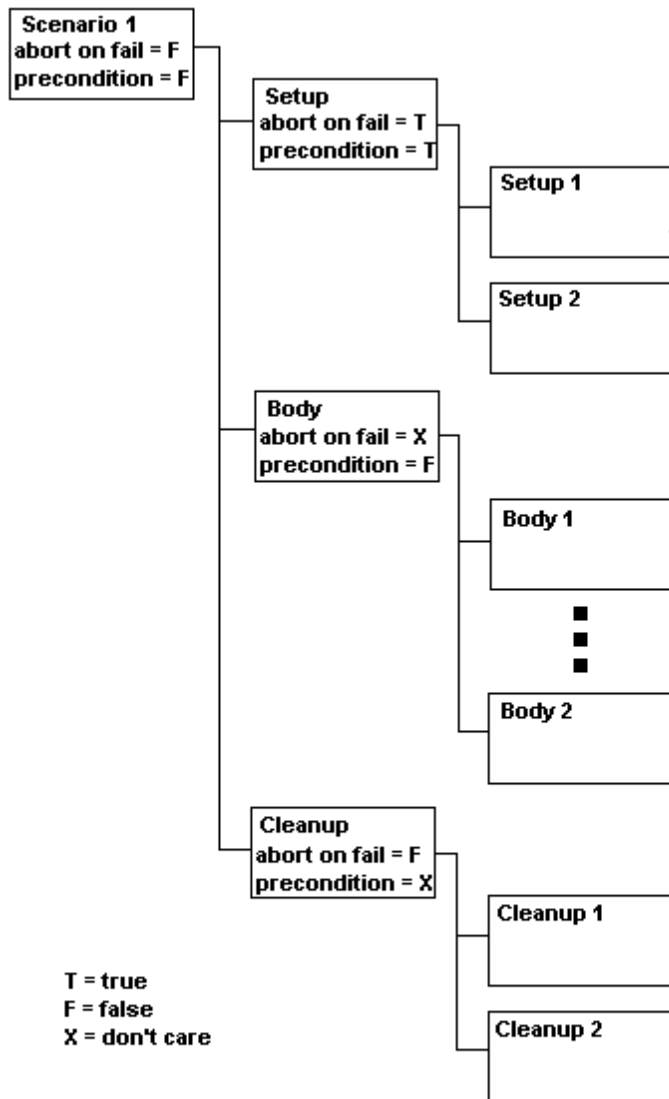
Because Collection **Step 2**'s *abort on fail* setting is set to *false*, and step **2a**'s *precondition* attribute is set to *false*, the failure of that step has no effect on the execution of the steps. Because step **2b**'s successful execution is a precondition to the execution of any subsequent steps, (in this case, **2c**) step 2c will not be executed.

There is one more way to abort a test on the failure of a step. You can use the Abort test on any failure checkbox in the Test Runner Execution control: group to cause the test that is being run to prematurely terminate on the occurrence of a failed step during the execution of the test.

How to 'clean up' after a failed step

Unless it is unavoidable, most tests should leave the Smalltalk image or system under test in the same state upon completion as it was in prior to that test's execution. Adherence to this practice makes running sequences of tests practical. Otherwise, tests require a great deal of defensive setup steps to prepare the system, prior to the execution of the test proper. That is not to say that there should not be a precondition step or steps at the beginning of a test to ensure that the system is in the correct state prior to test execution, but the onus should be on tests to clean up after themselves rather than relying on subsequently executed tests to perform that task. A good example of cleaning up would be ensuring that any windows opened during the execution of a test were closed.

The following diagram illustrates a possible configuration of steps for a scenario to accommodate both verifying that the system is in the correct state, and to clean up whether or not failures were encountered:



In this configuration, the scenario (**Scenario 1**) is partitioned into three sections:

Setup, in which the state of the system under test is verified to be correct for executing the test and any preliminary setup work is performed. The successful execution of this step is a precondition to executing the **Body** step.

Body, in which the main steps of the test are executed. It is important that the **Body** step not be a precondition to the execution of the **Cleanup** step, because the **Cleanup** step, in this example, must be executed under all conditions. Whether **Body** or any of its sub-steps aborts on failure, and the value of the precondition attribute for any of its sub-steps, is specific to the particular test being performed and is not of interest here.

Cleanup, in which the system under test or image is placed into a known state suitable for running subsequent tests. It is difficult to definitively state whether sub-steps of the **Cleanup** should all be run or should abort on any step's failure. For this example, it is assumed that all **Cleanup** sub-steps must be run, regardless of failure.

How to test asynchronous systems

Many systems do not operate on a single thread, but instead, spawn asynchronous processes that may complete at any time. Consider the example of a multi-tier system that allows a user to submit multiple requests to a server without knowing when responses to those requests will return, or in which order. A test that automates this could easily cause the requests to be made but would somehow have to be able to test for and handle responses that may return at any time. Another example is a test that ensures that a server system written in Smalltalk can carry out a task and still be responsive to asynchronous requests or interrupts.

These types of situations can be difficult to test because of their inherent non-determinism. The Smalltalk Test Mentor supplies special APIs for handling these types of situations.

The key to testing asynchronous systems is to construct usage scenarios that are as deterministic as possible, but still allow for uncertainty. You should design tests that place the system under test into a state where an asynchronous state is expected to be entered at some time and then wait for that state before proceeding, within a reasonable time.

In the first example above, you would design a test that causes one or more requests to be sent and then waits for all responses to be returned. For example, if responses manifest themselves in the form of widgets changing state, perhaps from containing no items to containing some, the test would need to wait until all the expected widget states are entered. Once this has happened, the test can proceed and verify the responses, perhaps by examining actual contents of the widgets to ensure the correct responses have been returned. For example:

1. Open application main view
2. Launch query view #1
3. Enter query criteria #1 and submit request
4. Launch query view #2
5. Enter query criteria #2 and submit request
6. Optionally test some other aspect of the system to ensure that the system remains responsive while requests are outstanding.
7. Wait for query responses to return. Fail if time exceeds maximum allowable. Optionally fail if responses do not return in the expected order, if ordering is important.
8. Verify response from query #1
9. Verify response from query #2

The key factor is to be able to make the test wait for one or more asynchronous state transitions to occur during a set time, without tying up system resources. The API for doing this is:

```
Object>>#checkCondition: conditionBlock
        interval: checkIntervalInMilliseconds
        timeout: timeoutInMilliseconds
```

This operates by sending #value to the condition block once on entry. If it does not return **true**, it repeatedly gives up time slices, sending #value to the condition block once every check interval until either the condition block returns **true** or the timeout value is exceeded.

How to make the best use of metrics

Metrics are only as good as the way in which they are used. The Smalltalk Test Mentor offers a set of metrics tailored to give you a view of the stability and maturity of the system under test, point out potential performance problems and suggest whether there are areas of the system under test that require further testing.

Execution time

Execution time is automatically measured for each suite, scenario and step. It is not uncommon for small changes to introduce subtle performance penalties that would otherwise go unnoticed. At a minimum, when you execute a suite, you should compare the execution time of it to that of a baseline run that has been archived. If the times are substantially different, you can use the Differences Browser to compare the times of the scenarios and steps within it to determine the source of the anomaly.

You can use classifications to tag similar steps, and then use the Find dialog to select items with those classifications, within most browsers. You can do this within the Test Results Browser to locate similar results. When multiple results are selected in the Test Results Browser, you are shown a statistical comparison of the results' metrics, including execution times.

Consider the example of a system that tests a transaction-based system. You can classify each step that causes a similar transaction (where similar means having the same expected transaction turnaround time) the same way. You can then select all of those steps' results and view a comparison of their execution times.

Code coverage

There is some amount of controversy concerning the general topic of code coverage metrics. There are those who argue that code coverage metrics should include a measurement of the total number of paths through each method. Although thorough, many logic errors may remain undetected. Consider the following:

```
(a > 1 & b = 0) ifTrue: [ x := x / a ].
(a = 2 | x > 1) ifTrue: [ x := x + 1 ].
```

Values of a=2, b=0, and x=3, ensure 100% code path coverage, yet some logic errors would yield the same results for those values:

- If '&' should be '|', the result would be the same
- If 'x > 1' should be 'x > 0', the result would be the same.

Method code path metrics fall short In Object-Oriented programs, where inter-object messaging adds a larger dimension to paths. Also, method code path metrics typically do not address the state of the system under test.

The Smalltalk Test Mentor does provide method coverage metrics, but the purpose is not to show you what *has* been covered in a system under test, but rather to show what *has not* been covered, and to indicate where more testing needs to be done. When you look at code coverage metrics in the Method Coverage

browser, you should be looking for classes with low coverage values and considering which usage scenarios you have overlooked in designing your test cases.

Executed/Passed/Failed steps

Unlike code coverage metrics, there is little controversy surrounding this metric. When you execute a test, you are given the total number of steps run, the number that passed and the number that failed. These numbers include any transient steps, as well.

Because some tests may abort on failure, you should be mindful of the total number of steps executed. This will tell you whether your test ceased executing prematurely. A test that completes prematurely may register a low number of failures.

If you are running tests where failures are common, particularly in the early stages of development, it is easy to overlook new failures. In these cases you should always compare results to a previous run. The Differences browser will show you if any new failures have occurred. It will also show you where previous errors have been fixed. This is important if the number of new failed steps and the number that no longer fail balance. In this case you won't see any difference if you just look at the totals.

Combining metrics

In general, metrics should be interpreted in light of one another. Most metrics, when taken by themselves, do not show the whole picture. For example, a decreasing number of total failures from week to week may indicate increasing product maturity, but not if the percent of methods covered is also decreasing. In this case you probably have a situation where older problems are being fixed, but new code, possibly indicated by an increase in the total number of methods in the system under test, that isn't covered by tests, is being introduced.

Miscellaneous

How to Search

Most of the Smalltalk Test Mentor views have a search facility built in. With it, you can select steps, results or differences, as appropriate, that meet some criteria. The following example shows you how. You will need to load the Smalltalk Test Mentor Examples configuration map or STM-Calculator parcel to try it yourself.

1. Open the Test Browser
2. Select the Find... item in the Edit menu.

The following dialog appears:

The screenshot shows a 'Find' dialog box with the following elements:

- Search criteria:**
 - Name:** [Empty text field]
 - Description:** [Empty text field]
 - Classification:** Ready, Regression Tests, Visual Tests
- Match Case:**
- Match Unclassified:**
- Find and select from:**
 - All Suites
 - Selected item
- Find Next** button
- Find All** button
- Direction:**
 - Forward
 - Backward
- Close** button

Note: The items listed under Classification: may be different in your Smalltalk image depending on which tests you have loaded.

3. Type **division** in the text field next to the Name: field. Make sure all check boxes, selections and text fields are set as shown above.
4. Press the Find All button.
5. You will see *all* steps that have "**division**" in their name selected.
6. Now press the Find Next button. You will see the second item with "**division**" in its name selected.

7. Press the Find Next button again. You will see the third item with "**division**" in its name selected.
8. Now select the Backward radio button and then press the Find Next button. The item from step #7 is selected.
9. The All suites and Selected item radio buttons indicate the scope of a search. Try this:
 - a. Press the Find All button. Again, all steps with "**division**" in their name are selected.
 - b. Now select the step named "**Division**" in the **Simple Calculations** scenario and press Find all. Notice that only the steps contained by "**Division**" are now selected.

How to adjust the proportional size of the upper and lower views within a window *(VisualAge only)*

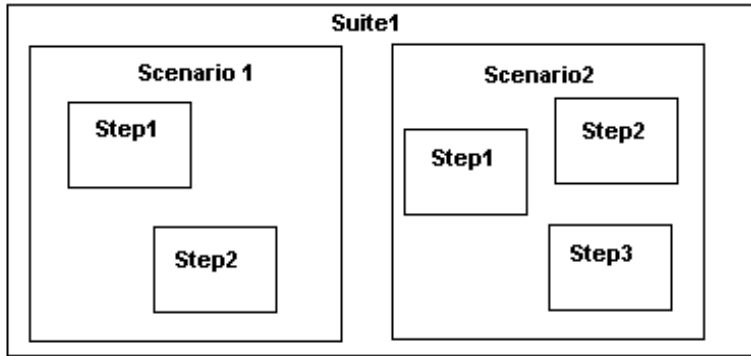
On Windows and OS/2 you can easily adjust the amount of room the upper and lower views use within the Test Editor, Test Browser, Test Runner, Test Results Browser, Differences Browser and Method Coverage Browser. To do this:

1. Move the mouse pointer over the divider between the upper and lower sections until the cursor turns into a vertical double arrow: ⇕.
2. With the left mouse button down, drag the divider until you have adjusted the size the way you like it and let go of the mouse button

The Smalltalk Test Mentor Programmer's Guide

Test Case Specification

Tests are defined by a containment hierarchy of Suites, Scenarios and Steps. A description of using this hierarchy can be found in the *Smalltalk Test Mentor Users' Guide*.



Specifying a Suite

Each separate subclass of `StmSuite` represents a single suite. Each suite is differentiated by its specification. The specification defines attributes of a suite such as its name, description, and execution controls. A specification for a suite is an instance of `StmSuiteSpec` that is returned by sending the message, `#spec`, to a suite's class. Here is a typical implementation of `#spec`, as generated by the Test Editor for an example suite:

```
spec
  "Run all calculator tests"

  ^(super newSuiteSpec
    name: 'CalculatorTests';
    description: 'Run all calculator tests';
    classification:
      (OrderedCollection new
        add: 'Build Verification';
        add: 'Regression';
        yourself);
    iterations: 1;
    abortOnFail: false;
    isAssertion: false;
    isPrecondition: false;
    scenarioSpecSelectors:
      (OrderedCollection new
        add: #modelViewTests_Spec;
        yourself);
    applicationsCovered:
      (OrderedCollection new
        add: #StmCalculatorApp;
        yourself) ).
```

The attributes for an instance of `StmSpec` are defined in the following table:

Name	Get Selector	Set Selector	Description
Abort on fail	<code>#abortOnFail</code>	<code>abortOnFail: aBoolean</code>	True if the execution of a suite is to be aborted on a scenario's failure and False if execution should continue.
Applications covered	<code>#applicationsCovered</code>	<code>#applicationsCovered: aCollection</code>	A collection of applications or VisualWorks name space names specified by their symbols. This collection is used when monitoring method coverage during test suite execution.
Classification	<code>#classification</code>	<code>#classification: aCollection</code>	A collection of strings, each of which is a classification for the suite.
Description	<code>#description</code>	<code>#description: aString</code>	A prose description of the suite.
Invariant selector	<code>#invariantSelector</code>	<code>#invariantSelector: aSymbol</code>	The selector for an instance method within the suite to be executed after each of the suite's scenarios, when the suite is executed. It should contain code to verify some condition of the system that the suite tests, that should always be true.
Iterations	<code>#iterations</code>	<code>#iterations: anInteger</code>	The number of times to execute the suite
Assertion	<code>#isAssertion</code>	<code>#isAssertion: aBoolean</code>	True if the step is expected to return a Boolean value to indicate whether the step has passed.
Precondition	<code>#isPrecondition</code>	<code>#isPrecondition: aBoolean</code>	True, if the execution of any tests subsequent to executing the suite's, is predicated on the successful execution of the suite.
Name	<code>#name</code>	<code>#name: aString</code>	The name of the suite.
Scenario specification selectors	<code>#scenarioSpecSelectors</code>	<code>#scenarioSpecSelectors: aCollection</code>	A collection of selectors in the suite's class specify methods that return specifications for the scenarios contained within the suite.

Specifying a Scenario

A scenario represents a discrete collection of steps within a suite. A scenario is defined by its specification and its steps. Each of those is defined by a separate class method within a suite's class.

The scenario spec

The specification for a scenario is similar to that of a suite. A specification for a scenario is an instance of `StmScenarioSpec` that is returned by a scenario's specification method. A suite has one scenario specification method for each scenario contained within it. The `#scenarioSpecSelectors` attribute of a suite specification returns the selectors for all of a suite's scenario specifications. Each selector represents its respective scenario's specification. Here is an example of a scenario specification method:

```

longCalculation_Spec
    "Test a long calculation"

    ^(self newScenarioSpec
        name: 'Long Calculation';
        description: 'Test a long calculation';
        classification:
            (OrderedCollection new
                add: 'Build Verification';
                add: 'Regression';
                yourself);
        abortOnFail: false;
        iterations: 1;
        isAssertion: false;
        isPrecondition: false;
        invariantSelector: #calculatorModelInvariant;
        scenarioSelector: #longCalculation )

```

Scenario specifications provide the following attributes:

Name	Get Selector	Set Selector	Description
Abort on fail	#abortOnFail	abortOnFail: aBoolean	True if the execution of a scenario is to be aborted on one of its steps' failure and False if execution should continue.
Classification	#classification	#classification: aCollection	A collection of strings, each of which is a classification for the scenario.
Description	#description	#description: aString	A prose description of the scenario.
Iterations	#iterations	#iterations: anInteger	The number of times to execute the scenario
Invariant selector	#invariantSelector	#invariantSelector: aSymbol	The selector for an instance method within the scenario's class to be executed after each of the scenario's steps, when the scenario is executed. It should contain code to verify some condition of the system that the scenario tests, that should always be true.
Precondition	#isPrecondition	#isPrecondition: aBoolean	True, if the execution of any tests subsequent to executing the scenario's, is predicated on the successful execution of the scenario.
Name	#name	#name: aString	The name of the scenario.
Scenario selector	#scenarioSelector	#scenarioSelector: aSymbol	The selector for a method that returns an instance of the scenario's class populated with the scenario's steps.

Scenario Steps

The steps for a scenario are returned by a class method that is referenced by the scenario specification's #scenarioSelector attribute. Here is an example of a method that returns the steps for a scenario:

enterDigit

```
"Enter a single digit and verify results"
| scenario_Instance |
scenario_Instance := self new.
^scenario_Instance
"-----< Step: Open View >-----"
addStepFromReceiver: scenario_Instance
selector: #openView
spec:
    (scenario_Instance newTestStepSpec
     name: 'Open View';
     description: 'Open the calculator view';
     iterations: 1;
     abortOnFail: false;
     isAssertion: false;
     isPrecondition: false );

"-----< Step: Use Calculator >-----"
addStepFromSteps: ( StmUIRecordedSteps new
spec:
    (scenario_Instance newTestStepSpec
     name: 'Use Calculator';
     description: '';
     iterations: 1;
     abortOnFail: false;
     isAssertion: false;
     isPrecondition: false );

"-----< Step: Use Calculator (1) >-----"
addStepFromRecording: [ (ActiveWindow setFocusToTitle: 'STM Calculator') ]
spec:
    (scenario_Instance newTestStepSpec
     name: 'Use Calculator (1)';
     description: 'Set focus to <STM Calculator>';
     iterations: 1;
     abortOnFail: true;
     isAssertion: false;
     isPrecondition: false );

"-----< Step: Use Calculator (2) >-----"
addStepFromRecording: [ (ActiveWindow widget: 'Button4') click ]
spec:
    (scenario_Instance newTestStepSpec
     name: 'Use Calculator (2)';
     description: 'Click on <4>';
     iterations: 1;
     abortOnFail: false;
     isAssertion: false;
     isPrecondition: true );

"-----< Step: Use Calculator (3) >-----"
addStepFromVerification: [ (ActiveWindow widget: 'DisplayText')
value = ('+ 4.0 ') ]
spec:
    (scenario_Instance newTestStepSpec
     name: 'Use Calculator (8)';
     description: 'Verify that DisplayText value = ('+ 4.0
''');
     iterations: 1;
     abortOnFail: false;
     isAssertion: false;
     isPrecondition: false )

);

"-----< Step: Use Calculator (4) >-----"
addStepFromSteps: ( StmUIRecordedSteps new
spec:
```

```

        (scenario_Instance newTestStepSpec
          name: 'Close Calculator';
          description: '';
          iterations: 1;
          abortOnFail: false;
          isAssertion: false;
          isPrecondition: false );
    "-----< Step: Close Calculator (1) >-----"

    addStepFromRecording: [ (ActiveWindow setFocusToTitle: 'STM Calculator') ]
    spec:
      (scenario_Instance newTestStepSpec
        name: 'Close Calculator (1)';
        description: 'Set focus to <STM Calculator>';
        iterations: 1;
        abortOnFail: false;
        isAssertion: false;
        isPrecondition: false );

    "-----< Step: Close Calculator (2) >-----"
    addStepFromRecording: [
      (ActiveWindow setFocusToTitle: 'STM Calculator') destroyWidget ]
    spec:
      (scenario_Instance newTestStepSpec
        name: 'Close Calculator (2)';
        description: 'Close <STM Calculator>';
        iterations: 1;
        abortOnFail: false;
        isAssertion: false;
        isPrecondition: false )
  )

```

The scenario steps methods are typically generated by the Test Editor, although you are free to create them without the editor, or alter those that have been already generated. In general, any changes you make in a generated method will be reflected in the editor provided you adhere to the expected structure. The structure consists of the creation of an instance of the scenario's class, followed by the population of that instance with steps.

Use the following methods to add steps:

Adding Base Steps

The following methods are provided by the StmExecutionApp application.

#addStepFromBlock: <aBlock> spec: <aStmSpec>

This adds a script step based on the source in aBlock, with the specification, aStmSpec.

#addStepFromClassSelfTest: <aClass> selector: <aSymbol> spec: <aStmSpec>

This adds a class method step based on the given class, aClass, and class method selector with the specification, aStmSpec.

#addStepFromManualText: <aString> spec: <aStmSpec>

This adds a manual intervention step based on the prompt text, aString, with the specification, aStmSpec.

#addStepFromReceiver: anObject selector: <aSymbol> spec: <aStmSpec>

This adds a step based on the source in a given receiver, and selector, with the specification, aStmSpec. This is usually generated for instance method steps, where the instance of the scenario's class is specified as the receiver.

#addStepFromSteps: <aStmCollectionStep>

This adds a collection of steps held by an instance of StmCollectionStep. StmCollectionStep instances respond to the same #addStepFrom... messages as StmScenario, so you add steps to an instance of StmCollectionStep the same way as you do a scenario. This enables you to create nested collections of

steps. For example, the following excerpt from a scenario is composed of three top-level steps; Setup, Body, and Cleanup, all of which implement the bulk of their function as sub-steps:

```
| scenario_Instance |
scenario_Instance := self new.

^scenario_Instance
"-----< Step: Setup >-----"
addStepFromSteps: ( StmCollectionStep new
spec:
    (scenario_Instance newTestStepSpec
        name: 'Setup';
        description: '';
        abortOnFail: true;
        iterations: 1;
        isAssertion: false;
        isPrecondition: true );
"-----< Step: Verify Not logged in >-----"
"
    addStepFromReceiver: scenario_Instance
    selector: #verifyNotLoggedIn
    spec:
        (scenario_Instance newTestStepSpec
            name: 'Verify Not logged in';
            description: 'Verify that the current logged in users
database
does not include 'TestUser1'.';
            iterations: 1;
            abortOnFail: false;
            isAssertion: false;
            isPrecondition: false );

);
"-----< Step: Body >-----"
addStepFromSteps: ( StmCollectionStep new
spec:
    (scenario_Instance newTestStepSpec
        name: 'Body';
        description: '';
        iterations: 1;
        abortOnFail: false;
        isAssertion: false;
        isPrecondition: false );
"-----< Step: Log In >-----"
    addStepFromSteps: ( StmUIRecordedSteps new
spec:
        (scenario_Instance newTestStepSpec
            name: 'Log In';
            description: 'Open log in dialog with
User = 'TestUser1'
Password = 'TestPassword1';
            iterations: 1;
            abortOnFail: false;
            isAssertion: false;
            isPrecondition: false ));
"-----< Step: Verify logged In >-----"
    addStepFromReceiver: scenario_Instance
    selector: #verifyUserIsLoggedIn
    spec:
        (scenario_Instance newTestStepSpec
            name: 'Verify logged In';
            description: 'Verify that current logged in users database
includes 'TestUser1'.';
            iterations: 1;
            abortOnFail: false;
            isAssertion: false;
            isPrecondition: false );
```

```

"-----< Step: Verify Not logged In >-----"
"
    addStepFromReceiver: scenario_Instance
    selector: #verifyNotLoggedIn
    spec:
        (scenario_Instance newTestStepSpec
         name: 'Verify Not logged in';
         description: 'Verify that the current logged in users
database
does not include 'TestUser1'.'.');
        iterations: 1;
        abortOnFail: false;
        isAssertion: false;
        isPrecondition: false );

"-----< Step: Log Out >-----"

    addStepFromSteps: ( StmUIRecordedSteps new
    spec:
        (scenario_Instance newTestStepSpec
         name: 'Log Out';
         description: 'Log out user 'TestUser1''
';

        iterations: 1;
        abortOnFail: false;
        isAssertion: false;
        isPrecondition: false ))

);

"-----< Step: Cleanup >-----"

    addStepFromSteps: ( StmCollectionStep new
    spec:
        (scenario_Instance newTestStepSpec
         name: 'Cleanup';
         description: '';
         iterations: 1;
         abortOnFail: false;
         isAssertion: false;
         isPrecondition: false );

"-----< Step: Remove Instances >-----"

    addStepFromReceiver: scenario_Instance
    selector: #removeInstances
    spec:
        (scenario_Instance newTestStepSpec
         name: 'Remove Instances';
         description: 'Invoke banking system test routine to
remove any stray instances left over after
walkbacks';
        iterations: 1;
        abortOnFail: false;
        isAssertion: false;
        isPrecondition: false )

)

```

Notice that user interface recording steps are added in the same manner as a collection of steps. That is because they exist in the same hierarchy.

#addStepFromTestScenarioClass: <aClass> selector: <aSymbol> spec: <aStmSpec>

This adds a scenario step based on the given class, aClass, and scenario selector, aSymbol with the specification, aStmSpec. Note that when this step is executed, the given specification will override the specification of the referenced scenario.

#addStepFromTestSuiteClass: <aClass> spec: <aStmSpec>

This adds a suite step based on the given class, aClass, with the specification, aStmSpec. Note that when this step is executed, the given specification's control attributes will **not** override those of the referenced suite. The reason for this is that it is permissible for the results of having executed a referenced suite from a suite step to be analyzed or archived in the same manner as the referenced suite by itself. Therefore, it is not advantageous to allow execution parameters such as 'abort on fail' or 'precondition' to be changed from a suite step.

#addStepFromWorkspace: <fileName> spec: <aStmSpec>

This adds a workspace step based on the given file name and path as a String, and specification. Note: Because the compiler can not be packaged in a runtime image, this step will only run correctly in a development image.

Adding User Interface Playback Steps

The following methods are provided by the StmExecutionVisualsApp application. You must package this application.

#addStepFromRecording: <aBlock> spec: <aStmSpec>

This adds a script step suitable for playing back a recorded user interface interaction, based on the source in aBlock, with the specification, aStmSpec.

#addStepFromVerification: <aBlock> spec: <aStmSpec>

This adds a script step based on the source in aBlock, with the specification, aStmSpec. It is assumed that the block provided evaluates to a Boolean.

Steps

Each step type is implemented by a subclass of StmStep. This section describes each type of step and its public attributes.

Class method step - StmClassMethodStep

This is a subclass of StmInstanceMethodStep. For execution, StmInstanceMethodStep may be used in its place. This class exists mainly to provide edit-time behavior.

Attribute Name	Get selector	Set selector	Description
Receiver	#receiver	#receiver:	The receiver for messages to the class - the class itself.
Selector	#selector	#selector:	The selector that the step sends to the class.

Instance method step - StmInstanceMethodStep

This provides behavior to send a message to an object as a test step. The object is typically an instance of the scenario containing the step.

Attribute Name	Get selector	Set selector	Description
Receiver	#receiver	#receiver:	The receiver for the message. In code generated by the test editor, this is always the instance of the scenario that contains the step.
Selector	#selector	#selector:	The selector that the step sends to the receiver.

Suite Step - StmSuiteStep

This step provides behavior to cause a suite to be executed as a step.

Attribute Name	Get selector	Set selector	Description
Applications covered	#applicationsCovered	#applicationsCovered:	A SequenceableCollection of symbols, each representing an application's symbol.
Instance	#instance		The instance of the class specified by the suite class attribute.
Invariant selector	#invariantSelector	#invariantSelector:	The selector for the suite's invariant.
Steps	#steps		An instance of StmCollectionStep whose steps are each a scenario step - one for each scenario within the suite.
Suite class	#suiteClass	#suiteClass:	The suite's class

Scenario Step - StmScenarioStep

This step provides behavior to cause a scenario to be executed as a step.

Attribute Name	Get selector	Set selector	Description
Instance	#instance		The instance of the class specified by the suite class attribute.
Invariant selector	#invariantSelector	#invariantSelector:	The selector for the scenario's invariant.
Steps	#steps		An instance of StmCollectionStep whose steps are each a step within the scenario.
Scenario selector	#scenarioSelector	#scenarioSelector:	A selector for the method that returns an instance of the scenario's class, fully populated with steps.
Suite class	#suiteClass	#suiteClass:	The suite's class

Workspace step - StmWorkspaceStep

This step provides behavior to cause a workspace to be read from a file, compiled, and executed as a step. Tests with this step will not work in a packaged image because the compiler is not present.

Attribute Name	Get selector	Set selector	Description
File name	#fileName	#fileName:	The name (and optionally, path) of the file containing the workspace to be executed, specified as a String.

Manual intervention step - StmManualStep

This step provides behavior to raise a prompter for specified text to be presented, as a step.

Attribute Name	Get selector	Set selector	Description
Text	#text	#text:	The text, specified as a String, to be used to prompt for manual intervention.

Collection step - StmCollectionStep

This step provides behavior to contain a collection of steps. Executing this step causes the contained steps to be executed.

Attribute Name	Get selector	Set selector	Description
All steps	#allSteps		A recursively generated list of all steps contained by the collection step and its children, returned as an OrderedCollection.
Steps	#steps		The steps immediately contained by the collection step, returned as an OrderedCollection.

User interface recording step - StmUIRecordedStep

This step provides the same execution-time behavior as StmCollectionStep. It exists to provide edit-time behavior for recording steps.

Attribute Name	Get selector	Set selector	Description
All steps	#allSteps		A recursively generated list of all steps contained by the collection step and its children, returned as an OrderedCollection.
Steps	#steps		The steps immediately contained by the collection step, returned as an OrderedCollection.

File Iteration step - StmCollectionOnTextFileStep

This step extends StmCollection step to execute the steps within it once for each row in a specified text file.

Attribute Name	Get selector	Set selector	Description
All steps	#allSteps		A recursively generated list of all steps contained by the collection

			step and its children, returned as an OrderedCollection.
Steps	#steps		The steps immediately contained by the collection step, returned as an OrderedCollection.
File	#file	#file:	The path and file name of the text file containing the test data.
Separator	#separator	#separator:	The character used to separate data items.
Assignment	#assignment	#assignment:	The character used to denote assignment of a value to a variable.

The selector, `#readingTextFile:separator:assignment:spec:` is usually used to initialize an instance of this step.

Script step - StmBlockStep

This step provides behavior to execute a Smalltalk script, as contained within a zero argument block, as a step.

Attribute Name	Get selector	Set selector	Description
Block	#block	#block:	A block whose source string is a Smalltalk script to be executed.

User interface step - StmRecordedStep

This provides the same behavior as `StmBlockStep`, with the additional behavior of delaying after execution of its block. The delay is specified by sending `StmExecutionApp>>#settlingTime:` an integer number of milliseconds to delay.

Attribute Name	Get selector	Set selector	Description
Block	#block	#block:	A block whose source string is a Smalltalk script to be executed.

User interface verification step - StmVerificationStep

This provides the same behavior as `StmRecordedStep`, with the exception that it is always configured as an assertion. This means that execution of the block should return a Boolean. The returned value is used to indicate whether the verification, and thus the step, passed or failed.

Attribute Name	Get selector	Set selector	Description
Block	#block	#block:	A block whose source string is a Smalltalk script to be executed.

Changing the execution time behavior of a step

At step execution time, subclasses of `StmContext` are created to manage the execution and creation of results for steps. Each subclass of `StmStep` references a corresponding subclass of `StmContext`, as specified by their respective `#contextClass` class methods. The actual execution of a step is performed by a

corresponding implementation of `StmContext>>#basicPerformTest`. If you wish to change the execution behavior of a step, you should subclass its context class and override `#basicPerformTest`.

Suite Operations

You can query a suite for its specification, its scenarios, execute a suite or execute a selected scenario. The following class methods implement behavior available to subclasses of `StmSuite` for performing these and other operations:

#performTest

This causes the receiver suite's scenarios to be executed and returns an instance of `StmTestResults`. See *Test Results* for a detailed explanation of the structure of the returned results.

#performTestStoreResultsInto: <fileName>

This causes the receiver suite's scenarios to be executed, returns an instance of `StmTestResults`, and as a side effect, writes the test results to the given file and path as a `String`. See *Test Results* for a detailed explanation of the structure of the returned results.

This is the recommended interface for executing a packaged suite. You should send this message to the suite you wish to execute as your image startup expression. For example,

```
MyTestSuite performTestStoreResultsInto: 'mytest.res'.
```

See *Smalltalk Test Mentor Users' Guide* for the procedure to load results from a file into a Test Results Browser.

#report

This returns a string containing a description of the suite and all its steps. Following is an extract of a report for one of the examples shipped with the product:

```

----- Suite----- :
Name: Calculator View Tests
Description: Test the StmCalculatorView view
Iterations: 1
Precondition: false
Assertion: false
Abort on Fail: false
  ----- Scenario-----
  Name: Single calculation
  Description: Perform a single calculation and verify results
  Iterations: 1
  Precondition: false
  Assertion: false
  Abort on Fail: false
    ----- Instance method-----
    Name: Open View
    Description: Open the calculator view
    Iterations: 1
    Precondition: false
    Assertion: false
    Abort on Fail: false
    ----- UI recording-----
    Name: Use Calculator
    Description:
    Iterations: 1
    Precondition: false
    Assertion: false
    Abort on Fail: false
      ----- UI Script-----
      Name: Use Calculator (1)
      Description: Set focus to <STM Calculator>
      Iterations: 1
      Precondition: false
      Assertion: false
      Abort on Fail: false
      .
      .
      .

```

#scenarioNamed: <aString>

This returns an instance of the named scenario, that is, it returns an instance of the scenario's class populated with the scenario's steps and specification. If more than one scenario exists with the same name, the first one found will be returned. Scenarios are returned in order of their specification selectors in the containing suite's specification.

#scenarioNamed: <aString> ifAbsent: <failBlock>

This returns an instance of the named scenario, that is, it returns an instance of the scenario's class populated with the scenario's steps and specification. If the scenario name is not found, #value is sent to the fail block. If more than one scenario exists with the same name, the first one found will be returned. Scenarios are returned in order of their specification selectors in the containing suite's specification.

#scenarioNames

This returns the names of all scenarios defined by a particular suite. The names are found by extracting the names from all scenario specifications listed by the #scenarioSpecs attribute of the suite's specification, in the order that they appear.

#scenarios

This returns all scenarios defined by the receiver suite, that is, an instance of the suite whose steps are scenario steps, each of whom represent a respective scenario. The steps are in the order that the scenario's specification selectors appear in their suite's specification.

#scenarioSpecNamed: <aString>

This returns the scenario specification whose #name attribute is the same as aString.

#scenarioSpecNamed: <aString> ifAbsent: <failBlock>

This returns the scenario specification whose #name attribute is the same as aString. If a scenario with the given name is not found, a #value message is sent to failBlock.

#scenarioSpecs

This returns an ordered collection of the instances of StmScenarioSpec returned by sending the selectors designated by the receiver suite specification's #scenarioSpecSelectors attribute.

#spec

This returns an instance of StmSuiteSpec, the receiver suite's specification.

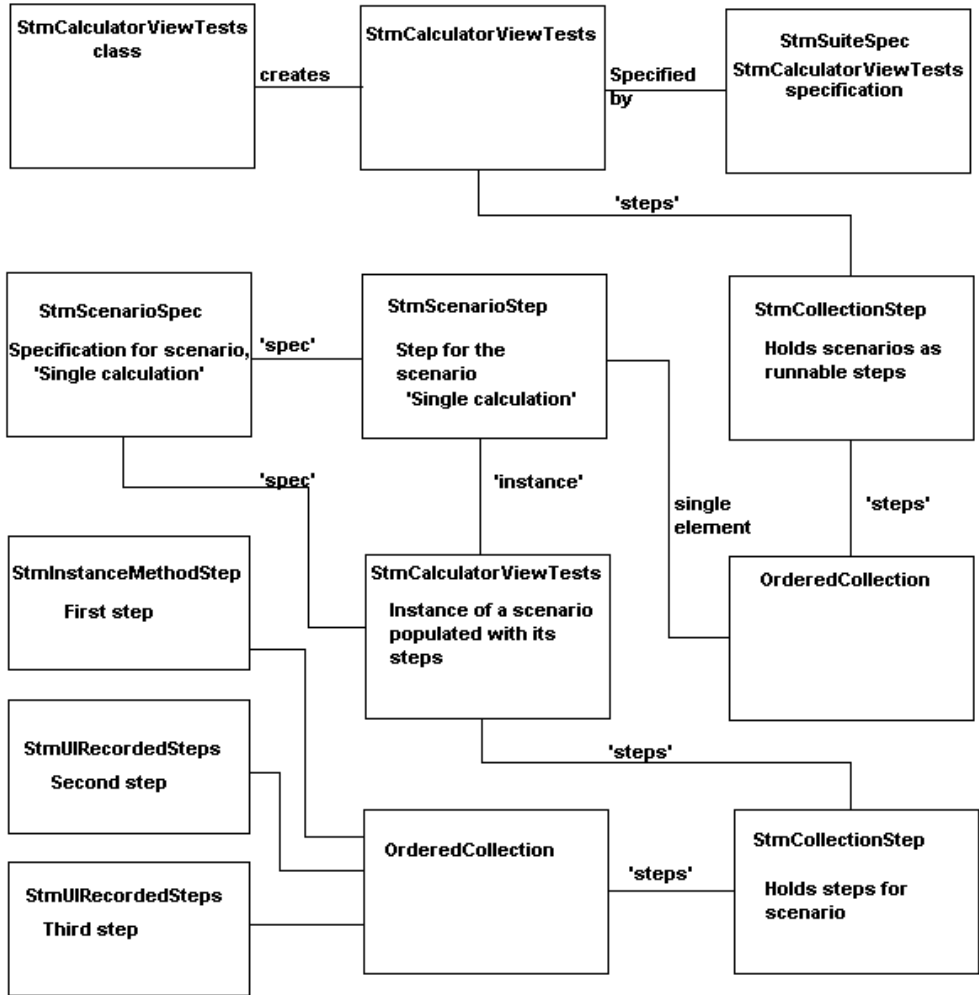
Test Case Structure

Now that the different components of a test case have been described, let's put them together by following through an example with Smalltalk inspectors. To follow this, you'll need to load the [Smalltalk Test Mentor Examples](#) configuration map into your image.

Action	Observation
StmCalculatorViewTests scenarios inspect	Inspector opens on an instance of StmCalculatorViewTests. Notice the two instance variables; spec and steps.
Open an inspector on the spec instance variable.	You see an instance of the suite's specification. You can close this inspector.
Open an inspector on the steps instance variable	You see an instance of StmCollectionStep. This is a ready-to-run step for execution of the scenarios.
Open an inspector on the steps instance variable	You see an OrderedCollection inspector with one element, an instance of StmScenarioStep. This step represents the single scenario defined by the calculator view tests Suite.
Open an inspector on the first element of the steps ordered collection	You see a StmScenarioStep inspector. Notice that the value of the instance variable called <i>instance</i> is nil.
Open an inspector on the spec instance variable of the StmScenarioStep	You see a StmScenarioSpec inspector. If you browse the instance variables, you can see the name, description and various execution controls of the scenario.
From the StmScenarioStep inspector text area, execute the expression: <code>instance inspect</code>	You see an inspector on an instance of StmCalculatorViewTests. This is the instance of the class created in behalf of the scenario.
Open an inspector on the steps instance variable of the recently created instance of StmCalculatorViewTests	You see an instance of StmCollectionStep. This is a ready-to-run step for execution of the scenario.

StmCalculatorViewTests.	
Open an inspector on the steps instance variable	You see an OrderedCollection inspector with three elements, each an instance of a subclass of StmStep. Each of these represents a step for the scenario.
Open an inspector on the second element, an instance of StmUIRecordedSteps.	You see a StmUIRecordedSteps inspector. StmUIRecordedSteps is a subclass of StmCollectionStep, so it holds a collection of sub-steps, each of which is an instance of StmRecordedStep.
Look at the following methods in StmCalculatorViewTests class: #spec #singleCalculation_spec #singleCalculation	You should be able to map the code you see to the instances of that code that you just inspected.

The structure described above can also be described in the following diagram (some less interesting aspects are intentionally left out):



Step Execution

A scenario is instantiated by creating an instance of the suite in which the scenario is contained and initializing it to contain instances of the scenario's steps. This happens automatically within the Test Browser or Editor as scenarios are traversed. You can also create an instance of a scenario by sending `#scenarioNamed: <nameOfScenario>` to a scenario's class. In this section, the term *'instance of a scenario'* should be considered synonymous with *'instance of a scenario initialized to contain instances of a scenario's steps'*.

When steps are executed, it is from within the context of a particular scenario, which as described above, is an instance of a subclass of `StmScenario`. In some cases this context may effect the way the step is executed. For example, sending the message, `#self`, from within the method specified for an instance method step can have different results depending on the receiver. In this case the receiver is an instance of the scenario in which the instance method step is defined.

The instance can be determined from the following table.

Step Type	Receiver for #self
Instance Method step	An instance of the scenario that contains the step.
Any of the Script steps	An instance of the scenario that contains the step.
Workspace step	nil

Suite and scenario steps are executed as their own instances. For example, if scenario A contains a scenario step B, and B contains an instance method, C, the receiver for any messages that C sends to `self` will be B, not A.

Execution APIs

When an instance method, or script step is executed, it is evaluated within the context of an instance of the scenario in which it is contained. The Smalltalk Test Mentor provides several methods for accessing and effecting the current state of the test, as well as a registry for accessing objects across steps that execute from within different scenario's contexts.

Test Variables

If you are using instance method or script steps within a test, you will probably need to keep some sort of state information or hold objects under test in the form of instance variables. In tests that span multiple scenario instances it is not possible to make those instance variables available to all steps because some may be operating from within different instances. In this situation, you should use the test variable APIs provided. They allow you to place named objects in a common area maintained by the test step dispatcher, which has visibility to all steps.

You can also use the test variable APIs if you simply want to avoid the bother of defining instance variables for your tests.

Note: These APIs replace #registerObject:named: and #registeredObjectNamed: from version 1.0 of the Smalltalk Test Mentor. The old selectors were simply too cumbersome to type. Those selectors are still supported, however.

#varNamed: <name> put: <object>

or

#var: <object> named: <name>

Either of these registers the given object under the specified name. The name used can be any object, although typically an atom, String or Symbol. Using either of selector has the same effect and choosing one over the other would be guided by personal preference.

#varNamed: <name>

This returns the object registered under the given name.

Execution Control

You can use the following methods to control the execution of your test or create and execute new, *transient*⁹, steps.

#performTest: <aBlock>

This will create a new step that executes the given block, as sub-step of the currently executing step. For example, sending #performTest: [Transcript cr; show: 'Hello'] from within a script step will show up in the Results Browser as a sub-step of the script step.

#performTest: <aBlock> name: <nameString> description: <descriptionString>

Like #performTest:, this creates a new step that executes the given block, as sub-step of the currently executing step. Additionally, the step will have the given name and description. For example, sending

⁹ A transient step is a step that only comes into existence during the execution of a test.

```
#performTest: [Transcript cr; show: 'Hello']
  name: 'Testing'
  description: 'Test this thing out'
```

from within a script step will show up in the Results Browser as a sub-step of the script step with the given name and description.

#failStep

Cease the current step's execution and register it as having failed.

#verify: <aBlock>

This will create a new step that executes the given block, as sub-step of the currently executing step. The block is expected to answer a Boolean indicating whether the verification condition contained within the block has been met. The block's returned value is used to indicate whether the #transient step has passed.

#verify: <aBlock> name: <nameString> description: <descriptionString>

Like #verify::, this creates a new step that executes the given block, as sub-step of the currently executing step. Additionally, the step will have the given name and description.

State

Use the following methods to query or effect the current state of a test.

#currentStep

Returns an instance of StmStep or a subclass, representing the currently executing step.

#currentIteration

Returns the number for the currently executing iteration of the current step

#currentResults

Returns an instance of StmTestResults for the currently executing step. The state of the returned object will depend on what point during the execution of a step this message is sent.

#currentStackDump

Returns a text dump of the current process execution stack.

#previousResults

This returns an instance of StmTestResults for the previously executed step.

#previousValue

This returns the value returned by the execution of the previous step, or nil if this is sent from the first step to be executed.

#pass: <aBoolean>

This specifies whether the currently executing step has passed or failed. A step is assumed to have passed unless one of the following happens:

A child step, whether transient or otherwise, has failed (#pass attribute of its results = false)

The currently executing step raises an exception

The #pass: message is sent with an argument of false.

Iteration control

When a step is dispatched, it executed for the number of times specified by its iterations attribute. You can cause it to be executed for additional iterations.

#repeatIteration

When sent during the execution of an iteration of a step, causes the current iteration to be repeated again once it has completed.

#repeatIterationIf: <conditionBlock>

When sent during the execution of an iteration of a step, causes the current iteration to be repeated again once it has completed, but only if the condition block returns **true** when sent #value.

Message logging

You can log messages to a file in such a way as to be protected from losing those messages if the Smalltalk virtual machine should generate a system exception. When a message is logged, the file is opened, the message written and the file closed again.

#logMsg: <aString>

Log a string to a file. The file name is derived from the currently executing scenario name.

#log: <aString> toFileNamed: <fileNameString>

Log a string to a file named by *fileNameString*.

Waiting on condition

Often during testing, a response to a stimulus may not appear immediately. In these cases, the test must wait a reasonable time for the condition to be met before proceeding. Use the following to wait for an asynchronous state to be met:

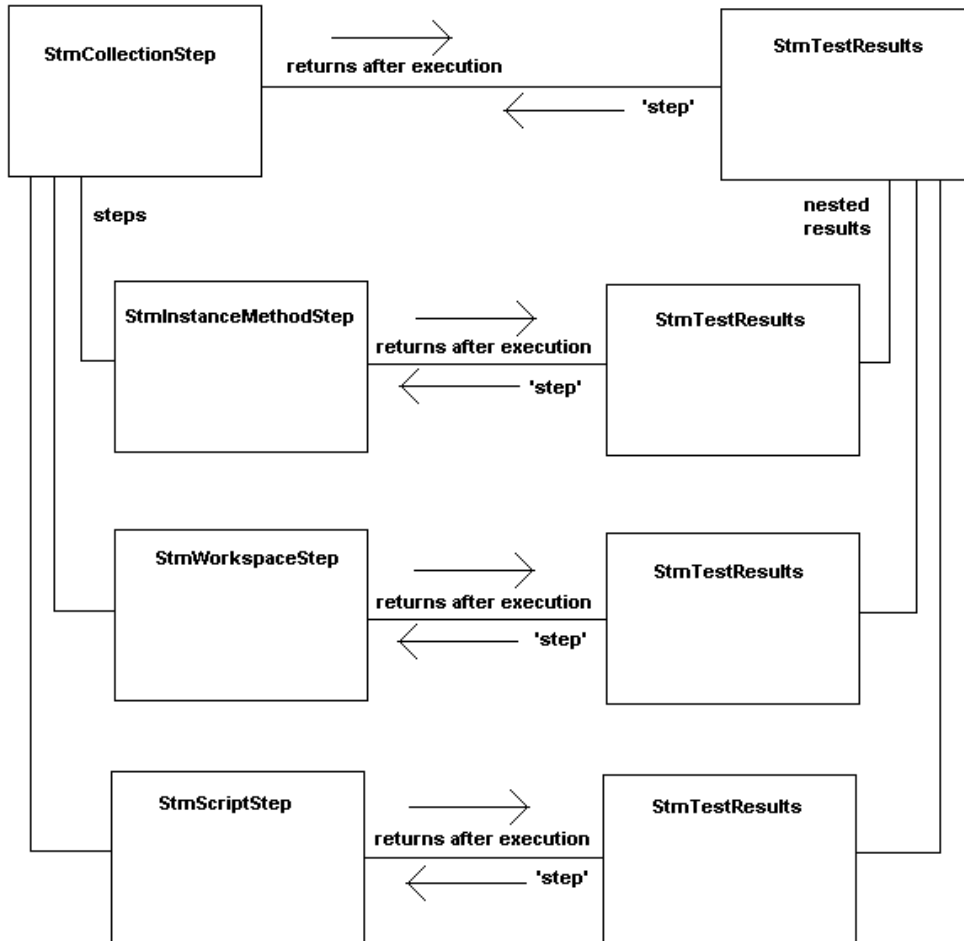
#checkCondition: conditionBlock
interval: intervalInMilliseconds
timeout: timeoutInMilliseconds

This operates by sending #value to the condition block once on entry. If it does not return **true**, it repeatedly gives up time slices, sending #value to the condition block once every check interval until either the condition block returns **true** or the timeout value is exceeded.

Test Results

When each step executes, an instance of `StmTestResults` is created. This class describes the results of having executed a step. Instances of `StmTestResults` are always created as a reflection of the steps within the test they execute. If a step has three sub-steps, the results it returns will have three nested results.

The following diagram shows the relationship between a step with three arbitrary sub-steps and the results that they produce:



The following table describes attributes of test results:

Name	Get Selector	Set Selector	Description
Abort on fail	#abortOnFail		True if the execution of a step was to be aborted on one of its sub-steps' failure and False if execution should have continued.
Classification	#classification		A collection of strings, each of which is a classification for the step that was executed.
Client data	#clientData	clientData: anObject	You can place any data you like here. Typically, this would be

			set from within a step to add additional information on the execution or failure of a step.
Description	#description		A prose description of the executed step.
Execution time	#executionTime		The time that it took for the step to execute. This is measured via Time>>#millisecondsToRun: on the most specific part of running the step. The value for this will be zero for steps that generate walkbacks, or manual intervention steps.
Assertion	#isAssertion		True if the execute step was expected to return a Boolean value to indicate whether the step had passed.
Iteration	#iteration		The iteration of the step that produced the results.
Iterations	#iterations		The total number of iterations specified by the that produced the results.
Precondition	#isPrecondition		True, if the execution of any tests subsequent to executing the step's, was predicated on the successful execution of the step.
Name	#name		The name of the executed step.
Nested Results	#nestedResults		A collection of instances of StmTestResults that represent the results for any sub-steps of the step represented by the results.
Number of methods	#numberOfMethods		This is only used by suite steps. It is the total number of methods controlled by the applications listed by the suite's #applicationsCovered attribute. The value is zero for any step that is either not a suite step or a suite step with no value specified by #applicationsCovered
Number of methods entered	#numberOfMethods Entered		This is only used by suite steps. It is the total number of methods controlled by the applications listed by the suite's #applicationsCovered attribute that were entered during execution. This number is always less than or equal to #numberOfMethods. The value is zero for any step that is either not a suite step or a

			<p>suite step with no value specified by <code>#applicationsCovered</code></p>
Number of testable methods	<code>#numberOfTestableMethods</code>		<p>This is only used by suite steps. It is the total number of methods controlled by the applications listed by the suite's <code>#applicationsCovered</code> attribute that were testable. This number is always less than or equal to <code>#numberOfMethods</code>.</p> <p>The value is zero for any step that is either not a suite step or a suite step with no value specified by <code>#applicationsCovered</code>.</p> <p>A method is not testable if it is compiled inline or it is a user or a system primitive.</p>
Pass	<code>#pass</code>	<code>#pass: aBoolean</code>	<p>This specifies whether the step executed successfully (true) or a failure of some sort was detected (false). A failure can be <i>hard</i>, such as caused by a walkback (<code>reason = ##exception</code>), or soft, where only this value is set.</p>
Reason	<code>#reason</code>	<code>reason: atom</code>	<p>This holds the reason for a steps' execution failure. This is automatically set on detection of an exception (<code>##exception</code>) and on the failure of a step that is an assertion (<code>isAssertion = true</code>).</p>
Signal	<code>#signal</code>		<p>For steps that caused an exception to be raised, this is the instance of <code>Signal</code> that was passed to the exception block.</p>
Stack	<code>#stack</code>		<p>The string form of the stack, at the time of the exception, or nil if no exception was detected.</p>
Step	<code>#step</code>		<p>The step whose execution caused the given results to be created.</p>
Number of tests passed	<code>#testsPassed</code>		<p>The number of tests that have passed. A step is counted as having passed if it can be considered to perform some test, as opposed to simply containing other tests, an <code>pass=true</code>.</p>

Number of tests run	#testsRun		The number of tests that have run. A step is counted as having run if it can be considered to perform some test, as opposed to simply containing other tests.
Text	#text	#text: aString	You can use #text to describe the circumstances of the execution of a step in any way you require. It is not otherwise used by the Smalltalk Test Mentor.
Time stamp	#timeStamp		The date and time that the step that produced the results was executed.
Returned value	#value		The value returned by the executed step.

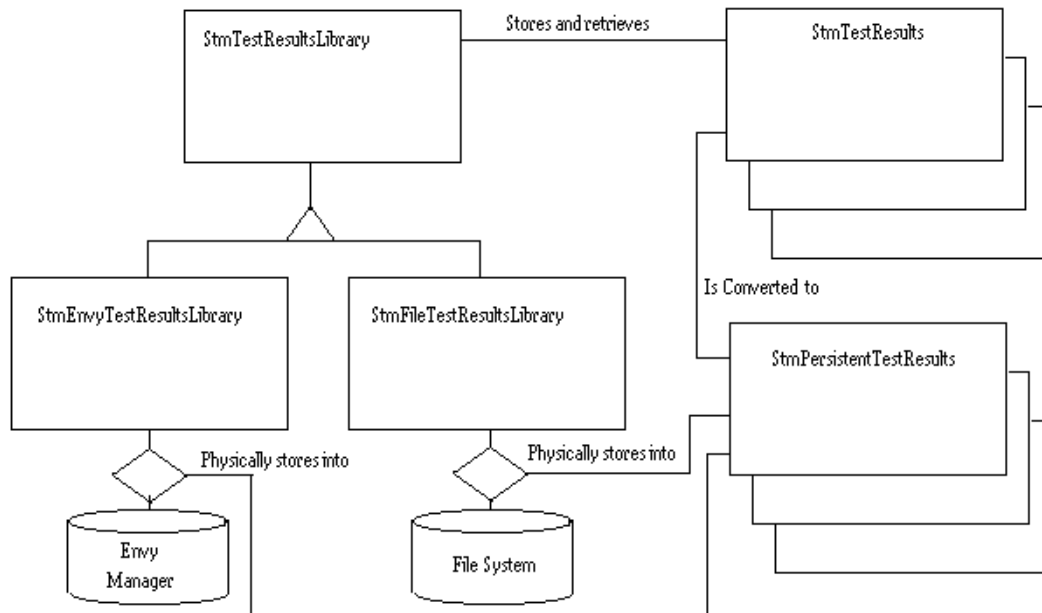
Persistent Test Results

Results are stored persistently in lightweight objects that are instances of `StmPersistentTestResults`. The attributes stored are: *clientData*, *executionTime*, *nestedResults*, *numberOfMethods*, *numberOfMethodsEntered*, *numberOfTestableMethods*, *pass*, *reason*, *results*, *signature*, *spec*, *stack*, *suiteClass*, *suiteClassVersion*, *text*, *timeStamp*, and *value*.

Adding New Types of Persistent Repository

The Smalltalk Test Mentor is capable of storing the result of executing a suite in either the Envy manager or a file. The code for doing this is structured so that you can add your own persistent repositories with a minimal amount of effort.

The objects for storing results are described in the following diagram:



To add another repository, simply subclass `StmTestResultsLibrary`. Use or override the following methods to add the required behavior:

#materializeResults: uniqueIdentifier

This takes a unique identifier for the results and returns an instance of `StmPersistentTestResults`, fully populated with the stored results. You will probably need to override this to access your repository. If this is called as a result of sending a `#retrieveResults:` message, the persistent results returned will be converted into instances of `StmTestResults` to be returned by `#retrieveResults`:

#storeResults: anStmTestResults

This stores the given results and is necessarily very repository dependent. You will need to prompt for or calculate a unique identifier for the results here.

Record and playback limitations and notes

Record and playback at the object level offers many advantages over recording individual mouse movements, clicks, and keyboard events. Primarily, it provides a very concise and reliable means to communicate with widgets. There are several limitations to using this approach. These are listed below:

Interactions with GraphicalObjects such as CwDrawingAreas are not recorded because they are primarily mouse and keyboard event driven.

Interactions with AbtHotSpotView are not recorded because it is a type of graphical object. If you need to play back a certain message to an AbtHotSpotView, create a script step that uses the VisualAge protocol for retrieving a subpart. For example:

```
( <MyView> subpartNamed: 'someHotSpot' ) click
```

or use the Smalltalk Test Mentor widget retrieval protocol:

```
(ActiveWindow widget: 'someHotSpot') userData click
```

Because most menu playback occurs underneath the covers you will not be able to see drop-down or pop-up menus appear, cascade, or toggle. Forcing menus to pop-up can cause the single Smalltalk thread to wait for an OS event to occur to release it. The Smalltalk Test Mentor works around this limitation by sending messages directly to the buttons held by these menus, without forcing the menus to appear or pop-up.

If a VisualAge Composition Editor is open in addition to any other instances of the view it defines, user interface playback messages may be sent to the Composition Editor instead of the appropriate target instance of the view. For example, if a user interface recording is to playback to an instance of the view *MyTestView* and a Composition Editor is open on *MyTestView* in addition to the test view, the playback messages may be sent to the shadow widgets in the Composition Editor. If playback does not seem to occur as expected, close any open Composition Editors that may interfere with proper widget referencing.

All drag events are automatically recorded as a move vote, generating a #moveDrag playback selector. If the intended operation should be to copy or link, override the playback selector using either the #copyDrag or #linkDrag APIs. Most common drag and drop recording and playback scenarios will automatically be handled, although some user-defined callData and clientData operations may not be properly reproduced. In these cases you may see that the drop operation does not produce the expected results. In order to avoid potential screen coordinate problems, most drag actions are based upon selection. You will get the best drag/drop record and playback performance if the items to be moved are first selected.

VisualAge OS/2 specific widgets - Are not recorded. The 'OS/2 Container' and 'OS/2 - Windows Notebook' can be sent playback messages using either the standard VisualAge part protocol or the Smalltalk Test Mentor widget protocol.

For security reasons, passwords entered during recording of #CwPasswordPrompterare are recorded as asterisks (*). You will need to edit these steps and enter the correct password value manually.

Index

A

abort on any test failure..... 71
abort on Fail..... 41
Abort on Fail..... 50, 78, 174
archive 81, 82, 132, 140, 141, 145, 158, 169

C

capture 33, 47, 50, 65, 66, 67, 69, 108, 109,
112, 113, 115, 116
Checked conditions for selected steps and 75, 129
class method step 42
Class Method Step 51
classification 41
Classification 160, 163, 164, 182
collection step 43
Collection Step..... 155
compare 82, 84, 85, 133, 134, 158, 159
Compress repetitive steps 55
Contract 59, 61, 73, 74, 80, 81, 86, 91, 92
Coverage analysis 88, 138

D

delete 55, 60, 64, 108
description..... 37, 41, 45
Description 41, 46, 49, 50, 69, 70, 72, 77, 78, 79,
85, 89, 90, 102, 103, 111, 114, 163,
164, 169, 170, 171, 172, 174, 182, 183
Details View ... 50, 51, 52, 53, 54, 56, 57, 58, 78,
85, 91, 106
differences..... 10
Differences 81, 84, 85, 86, 88, 91, 136, 137, 158,
159
drag 103, 104
drop-down list..... 51, 53, 64, 103, 106, 150

E

execution time 36, 38, 42, 43, 78, 79, 84, 85, 133,
136, 144, 158, 172
expand 35
Expand 59, 61, 73, 74, 80, 81, 86, 91, 103, 125,
130, 134, 139, 141
Exporting 80

F

File Iteration..... 43, 49, 52, 122, 171
find 36, 37
Find 59, 60, 73, 76, 80, 81, 86, 91, 137, 158,
160, 161

G

Generate workspace script 54

I

insertion point 103, 104, 105

Inspect Step..... 71, 129
instance Method Step..... 42
Instance Method Step..... 53
invariant..... 45, 46, 58, 150, 151, 163, 164, 170
Iterations 42, 50, 78, 107, 163, 164, 174, 183

L

list view 49
List View..... 70, 77, 84, 89

M

manual intervention step 54
manual intervention Step 43
match 84, 85, 136, 137
method coverage 10, 33, 46, 88, 138, 158, 163
Method Coverage Browser 10, 81, 88, 89, 91,
138, 140
Metrics 89, 158
Modifying a Step..... 64

N

name 31, 41
Name 41, 45, 46, 49, 50, 59, 60, 61, 64, 73,
74, 77, 79, 80, 81, 83, 85, 86, 87, 89,
90, 91, 92, 103, 109, 116, 143, 151,
160, 163, 164, 169, 170, 171, 172, 173,
174, 175, 178, 179, 182, 183
new scenario..... 29
New Scenario 59, 60, 63, 102
new step 30
New Step..... 59, 60, 63, 102
new suite 26, 28
New Suite..... 59, 60, 62, 101, 102
nontestable 91

O

Open Debugger 71, 75, 76, 127

P

passed 36, 69, 72, 85, 135, 137, 138, 143, 159,
163, 172, 180, 183, 184
pause 67, 70, 71, 75, 76, 112, 123, 124, 127,
128, 129, 130
Pausing the Execution of a Test 75
precondition 41
Precondition 50, 78, 155, 162, 163, 164, 165,
166, 167, 168, 174, 183

Q

Quick Run. 33, 34, 35, 59, 61, 65, 66, 67, 68, 69,
73, 74, 108, 109, 116, 123, 124, 130,
138, 151
Quick Runner 9, 33, 34, 35, 59, 61, 65, 66, 67,
68, 69, 73, 74, 108, 123, 124, 130

R

record 33, 35, 45
Record 45, 54, 65, 66, 67, 108, 109, 111, 112,
113, 114, 115, 116, 165, 166, 167, 168,
169, 171, 172, 176
resume 35
Resume 67, 71, 76, 112, 113, 120, 129, 130
Running a Simple Test..... 74

S

Saving Changes..... 64
scenario 9, 29, 40, 42, 45
Scenario 42, 49, 50, 53, 57, 59, 60, 62, 63, 75,
102, 150, 155, 156, 162, 163, 164, 166,
168, 170, 174, 175, 178
script step 44
Script Step..... 56
Smalltalk Test Mentor - *runner* 9, 10, 68
StmCalculatorView. 32, 109, 113, 116, 120, 139,
174, 175, 176
suite 26, 27, 40, 42, 46
Suite 42, 49, 50, 53, 58, 59, 60, 62, 63, 101,
102, 108, 125, 148, 149, 150, 151, 162,
169, 170, 173, 174, 175, 178

T

Terminate 67, 71, 129, 130
Test Browser. 9, 68, 70, 123, 125, 129, 130, 132,
138, 160, 178
test case structure 40
test Editor 47
Test Editor .. 9, 25, 30, 31, 34, 44, 45, 46, 47, 59,
60, 62, 63, 64, 65, 66, 67, 68, 69, 73,
80, 86, 91, 100, 101, 102, 103, 106,
108, 109, 112, 113, 114, 115, 116, 120,
123, 124, 125, 129, 130, 132, 134, 135,
138, 141, 142, 162, 166
Test Results Browser .. 35, 36, 74, 76, 77, 82, 84,
88, 113, 124, 125, 132, 133, 135, 136,
138, 140, 141, 142, 143, 145, 158, 173

Test Runner 9, 68, 70, 71, 73, 74, 75, 76, 123,
124, 125, 126, 127, 129, 130, 135, 142,
155
Test Step 40, 41, 64, 76
testable 79, 88, 90, 91, 184
time stamp 78, 85, 137
traffic light 36, 77, 85
type 31
Type 32, 50, 64, 78, 102, 103, 106, 160, 178,
185

U

UI recording 32, 105, 108, 109, 111, 112, 113,
114, 115, 116, 174
unspecified 31, 45
Unspecified 32, 49, 50, 57, 64, 103
Untestable 90
use case 9, 40, 146, 148
user interface interactions ... 9, 32, 33, 65, 66, 69,
108, 111, 113, 114, 116, 130
user interface recording mode 34, 65, 66, 67
User interface recording mode 65
user interface recording step ... 33, 34, 45, 54, 55,
65, 66, 67, 69, 168
user interface step 45
User Interface Step 57
user interface verification Step 45
User Interface Verification Step 57

V

verify 45, 115, 116, 120, 156, 163, 164, 165,
167, 168, 174, 180
VisualWorks See . See . **See** . **See** . See . See .
See . **See** . See . **See** . **See** . **See** . **See** .
See . **See** . See . See . See . See . See .
See . See . See . See . See . See . See .
See . See . See . **See** . **See** . See . See .
See . See . See . See . See . See . See .
See . See . See . See . See . See

W

Walkbacks 69, 72
workspace step 43