



SilverMark's
Enhanced
JUnit

Version 3.7
Copyright 1996-2002 SilverMark, Inc.
www.javatesting.com
info@javatesting.com

User Reference

©1996-2002 SilverMark, Incorporated. All rights reserved

Document Version: **3.7.06.21.2002**

This edition applies to version 3.7 of SilverMark's Enhanced JUnit, 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 Unpublished rights reserved under the copyright laws of the United States.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:

<p>SilverMark Inc. 5511 Capital Center Drive Suite 510 Raleigh, NC 27606 Attention: Publications E-mail: publications@silvermark.com</p>

RELEASE NOTES	5
TRADEMARKS.....	5
<i>The following terms are trademarks of other companies:</i>	5
PREREQUISITES	5
<i>System requirements</i>	5
SUPPORT	5
INTRODUCTION.....	6
GETTING STARTED	6
WHAT IS JUNIT?.....	6
WHY ENHANCE JUNIT?	6
WHAT IS SILVERMARK’S ENHANCED JUNIT?	7
<i>User interface additions</i>	7
JUNIT/PROFILE	8
METHOD COVERAGE PROFILING	8
<i>Setting up method coverage profiling</i>	9
<i>Running with method coverage profiling</i>	10
OBJECT INTERACTION PROFILING	11
<i>Launching object interaction profiling</i>	12
<i>Object interaction recording wizard</i>	12
<i>Suggested usage</i>	16
JUNIT/LOAD.....	17
CONCEPTS	17
SETTING UP FOR LOAD TESTING.....	19
USING JUNIT/LOAD WITH JUNITPERF.....	22
<i>Using TimedTest with JUnit/load</i>	23
<i>Using LoadTest with JUnit/load</i>	24
JUNIT/GENERATE	25
GENERATING TESTS FOR CLASSES	25
GENERATING TESTS FROM RECORDED OBJECT INTERACTIONS.....	31
<i>Hands-on example of test generation from object interactions</i>	33
JUNIT/UTIL	42
DATA-DRIVEN TESTING	42
<i>Iterating over text records</i>	43
<i>Iterating over delimited records</i>	43
<i>Keyed delimited text record searches</i>	44
EXECUTION TIMING	45
<i>Timer lifecycle operations</i>	45
<i>Timer assertion operations</i>	45
<i>Special load test timing operation</i>	46
APPENDIX A – THE ENHANCED JUNIT ARCHITECTURE	48

APPENDIX B - INSTALLATION AND SETUP	49
SETTING UP SETVARS.BAT	50
SETTING UP SMEJUNIT.PROPERTIES.....	50
SETTING ENVIRONMENT SPACE FOR WINDOWS 95, 98 AND ME.....	51
APPENDIX C – JPDA SETUP	52
ADD JPDA TO THE JUNIT RUNNER CLASSPATH	52
ADD JPDA TO THE SYSTEM PATH	53
COMPATIBILITY WITH IDEs AND DEBUGGERS	54
APPENDIX D – STARTING ENHANCED JUNIT	54
EXAMPLES.....	55
<i>Start Swing UI runner</i>	55
<i>Start Swing UI runner, specifying a test to run on startup</i>	55
<i>Start AWT UI runner</i>	55
<i>Start AWT UI runner, specifying a test to run on startup</i>	55
<i>Start load test execution agent, using default port (3050)</i>	55
<i>Start load test execution agent, specifying required port to listen for requests on...</i>	56
APPENDIX E – ADDING CUSTOM LOAD TEST RAMP PROFILES.....	56

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:

Enhanced JUnit, Test Mentor

The following terms are trademarks of other companies:

Java

Sun Microsystems, Inc.

Java Platform Debugger Architecture

Sun Microsystems, Inc.

Windows

Microsoft Corporation

Prerequisites

- Enhanced JUnit must be executed with the Java™ 2 Standard Development Kit (SDK) Version 1.2.x or greater
- Enhanced JUnit's profiling and method coverage metrics requires access to the Java Platform Debugger architecture. This is included with Java SDK 1.3.0 and up, but has to be downloaded for earlier versions. See [Java Platform Debugger Architecture \(JPDA\) Interface Setup for details.](#)
- Enhanced JUnit requires Microsoft Windows/95/98/NT/2000.
 - SilverMark is certifying the product for other operating systems. If you are interested in using Enhanced JUnit on other platforms, please contact SilverMark for a release estimate.

System requirements

In order to operate most effectively, 128Mb or more of system memory, and 12Mb of storage is required.

Support

For questions, problems or general assistance regarding the SilverMark's Enhanced JUnit, send e-mail to support@silvermark.com.

Introduction

Getting started

Installation and setup instructions are located in [Appendix B - Installation and setup](#). There is some minor setup required so please read these instructions before starting the product.

Startup instructions are located in [Appendix D – Starting Enhanced JUnit](#).

What is JUnit?

JUnit is a simple, light-weight, open-source Java test execution framework available from www.junit.org (and is included with the Enhanced JUnit product distribution) written by Kent Beck and Erich Gamma. It provides a user interface for selecting and launching tests, and APIs for asserting conditions.

Why enhance JUnit?

As with any simple tool, there is always room for improvement – especially when time is so valuable for the tool's users. To that end, we see (among others) the following areas of opportunity for JUnit improvement:

- Make test creation easier through automation
 - Generate unit tests based on class structure. That is, generate reusable test assets for a class based on common test design patterns given a its methods, constructors and fields.
 - Generate unit tests based on interactions between objects
- Promote consistent test architecture based on common test design patterns
- Make it easy to perform load/stress testing through concurrent execution of multiple copies of a test
 - Distribute JUnit tests on multiple threads, VMs and CPUs
 - Show throughput response graphs
- Make it easier to validate the effectiveness of tests by reporting methods covered and more importantly, not covered by tests
- Make it easy to iterate over test data files to create data-driven tests

- Make it possible to add fine-grained execution time assertions

What is SilverMark's Enhanced JUnit?

SilverMark's Enhanced JUnit is a suite of tools that integrate with and expand upon the capabilities of JUnit:

[JUnit/profile](#) – method coverage and object interaction profiling

[JUnit/load](#) – load testing across any number of threads, JVMs and CPUs

[JUnit/generate](#) – automatic test generation

[JUnit/util](#) – a growing list of handy utilities and APIs

Note: Enhanced JUnit does not change JUnit in any way. Not one line of code in JUnit was changed to accommodate Enhanced JUnit. All Enhanced JUnit view classes are subclasses of corresponding JUnit classes.

User interface additions

The Enhanced JUnit AWT and Swing user interfaces appear identical to those of JUnit with the exception of the addition of a SilverMark menu:

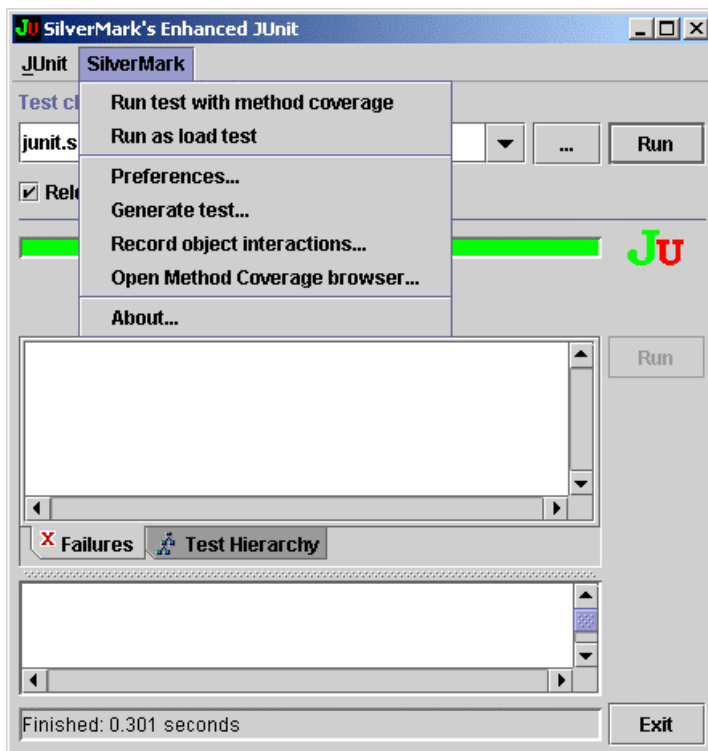


Figure 1 - JUnit Swing UI with SilverMark menu shown

You access all Enhanced JUnit functionality through this menu:

Menu	Purpose
Run test with method coverage...	Run specified test and display method coverage profile results view
Run as load test...	Distribute specified test to load test agents on same or remote machines, and display real-time execution time metrics over incremental load.
Open preferences...	Open preferences view to edit global settings.
Generate test	Generate tests according to global generation preferences.
Record object interactions...	Record interactions between objects in order to debug, or generate tests according to those interactions
Open method coverage browser...	Open an empty method coverage browser. You can then load saved coverage data
About...	Shows version and support information

JUnit/profile

Method coverage profiling

Problem: It is very easy to miss testing areas of code. This leads to false confidence in the effectiveness of your test cases.

Solution: Monitor execution of classes under test and test classes in order to determine if any methods are not being executed.

JUnit/*profile* JUnit provides the ability to monitor the execution of a class or classes, and presents a summary of which methods were executed, how many times they were executed, and which methods were not executed.

You may monitor method execution in the VM in which the test case is running or in a separate VM, such as a server running code that your test case is remotely requesting services from.

Setting up method coverage profiling

You must first have JPDA installed correctly on your system. See [Appendix C – JPDA setup](#) for details.

Open the Enhanced JUnit preferences by selecting the Preferences... item in the SilverMark menu, then navigate to the JUnit/profile tab:

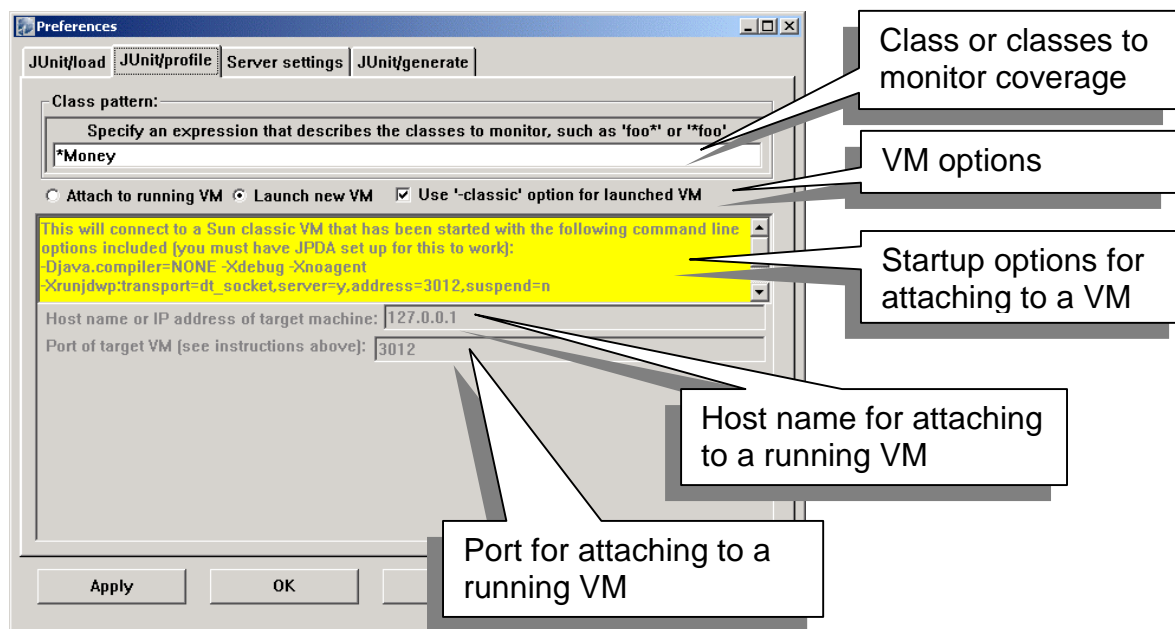


Figure 2 - Preferences settings for method coverage profiling

These settings specify which classes to watch when you choose to run with method coverage profiling enables, whether to attach to an existing VM (in the case that you are monitoring classes running on a server) or launch the test in an instrumented VM (the typical case).

Name	Description
Class pattern	A pattern that describes the classes to monitor. Matches are limited to exact matches of the given class pattern and matches of patterns that begin or end with *; for example, *.foo or java.*.
Launch new VM	Launch test in a new Java virtual machine. Use this for most cases where you are testing client or stand-alone code.

<p>Attach to running VM</p>	<p>Attach to an existing Java virtual machine to gather instrumentation information. Use this if you are testing classes that are deployed on a server running on a Java virtual machine that is separate from the test suite classes.</p> <p>Use this if your tests act as remote clients of the classes under test. Because the classes under test are running in a separate VM you would want to attach that VM to monitor method coverage and object interactions.</p> <p>In order to attach to a VM you need to start the target VM with the following command line options:</p> <pre>-Djava.compiler=NONE -Xdebug -Xnoagent -Xrunjdwp:transport=dt_socket,server=y,address=myPort,suspend=n</pre> <p>Note: This option is shown in yellow text. If you pop up over it, you will see a menu option to copy the options to the clipboard.</p>	
	<p>Host name or IP address of target machine</p>	<p>The name of the machine that is running the VM to attach to.</p>
	<p>Port of target VM</p>	<p>The port to connect to, as specified as a command line parameter for the VM when it was started.</p>
<p>Use <code>-classic</code> option</p>	<p>The <code>-classic</code> VM option may speed up execution of instrumented code. You should use this option if your Java VM supports <code>-classic</code>.</p>	

Running with method coverage profiling

To run with method coverage profiling, enter a test case name in the JUnit UI as you normally would and select the Run with method coverage... option in the SilverMark menu.

The test should run as it normally does, albeit a bit more slowly and with some informational messages printed to the console. Upon completion the Method coverage metrics browser opens.

Results of running with method coverage profiling

Method coverage metrics are shown with the following view:

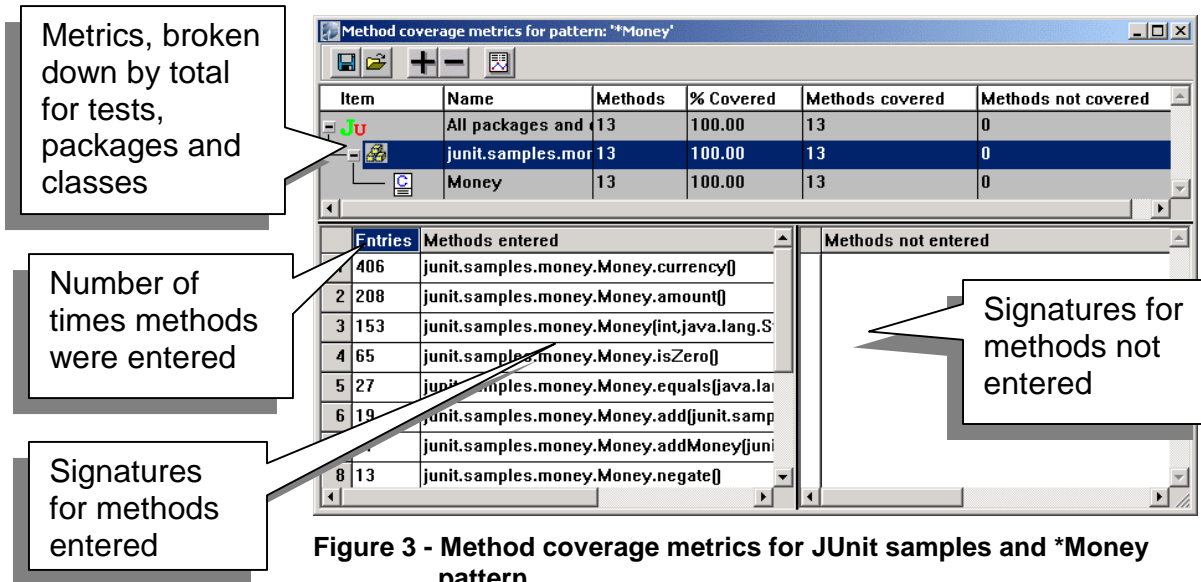




Figure 3 - Method coverage metrics for JUnit samples and *Money pattern

The top half shows a tree view broken down by the root (run) test case, packages and classes. It also shows the total number of methods, methods entered, methods not entered and percent coverage for each item.

The lower half lists methods entered and the number of times they were entered on the left, and methods not entered on the right.

You can save coverage results as a file by pressing , or open an existing file with . To open an empty method coverage browser view, simply select the Open Method Coverage Browser... item in the SilverMark menu.

Object interaction profiling

Problem 1: Extreme programming emphasizes test-first development, where the test cases come into existence as (slightly before) the code under test. As beneficial as this approach is, it is not always practical. Very few programmers have the luxury of always writing new code. Eventually you will end up inheriting somebody else's code that doesn't have test cases that you will have to fix, refactor or add to.

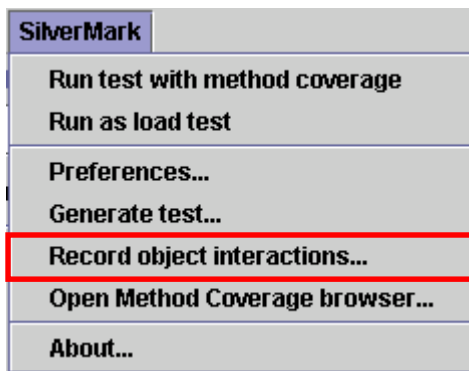
Problem 2: It is often difficult to visualize the relationships between objects just by looking at code or diagrams, and often tracing execution using a debugger provides too low-level a view.

Solution: Monitor interactions with objects under test in order to generate sequence diagrams that describe those interactions. Generate unit tests that mimic those interactions and discover the exact time-ordered interaction relationships between objects.

Enhanced JUnit provides a facility for specifying a class or classes to monitor, presents interactions with that class or classes as a time-ordered interaction diagram.

Launching object interaction profiling

To run object interaction recording, select the Record object interactions... item from the SilverMark menu.



This will open the Object Interaction Recording Wizard.

Object interaction recording wizard

The first panel shows recording options for specifying whether to launch or attach to a VM, and which class or classes to monitor:

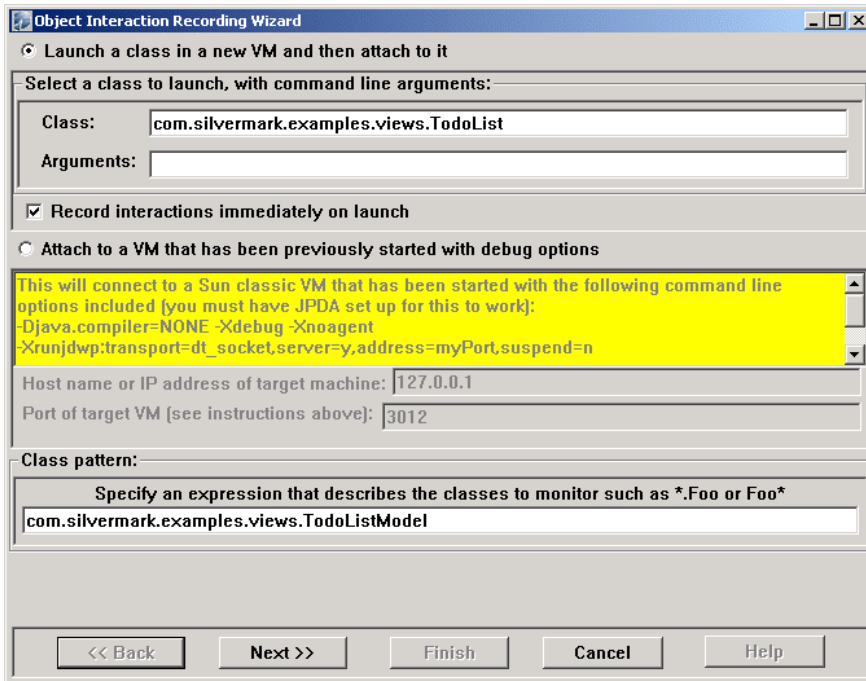


Figure 4 - Start panel for object interaction recording wizard

Name	Description	
Launch a class in a new VM and then attach to it	Select this if you want to launch a Java application in a new, instrumented. This is the most typical case.	
	Class to launch	Provide the name of a class to launch an instrumented VM on. You must have your JUnit classpath set properly so that it has visibility to this class.
	Arguments	Command line arguments to pass when launching the class
	Record interactions immediately on launch	Specifies whether or not to record interactions during application startup. Uncheck this for a speedier application startup if startup interactions are not interesting to you.

Attach to a VM that has been previously started with debug options	<p>Attach to an existing Java virtual machine to gather interaction information.</p> <p>Use this if you wish to observe interactions with classes within an already running application, such as a server</p> <p>In order to attach to a VM you need to start the target VM with the following command line options:</p> <pre>-Djava.compiler=NONE -Xdebug -Xnoagent -Xrunjdwp:transport=dt_socket,server=y,address=myPort,suspend=n</pre> <p>Note: This option is shown in yellow text. If you pop up over it, you will see a menu option to copy the options to the clipboard.</p>	
	Host name or IP address of target machine	The name of the machine that is running the VM to attach to.
	Port of target VM	The port to connect to, as specified as a command line parameter for the VM when it was started.
Class pattern	<p>A pattern that describes the classes to monitor. Matches are limited to exact matches of the given class pattern and matches of patterns that begin or end with *; for example, *.foo or java.*.</p>	

When you press Next >> you see an object interaction diagram.

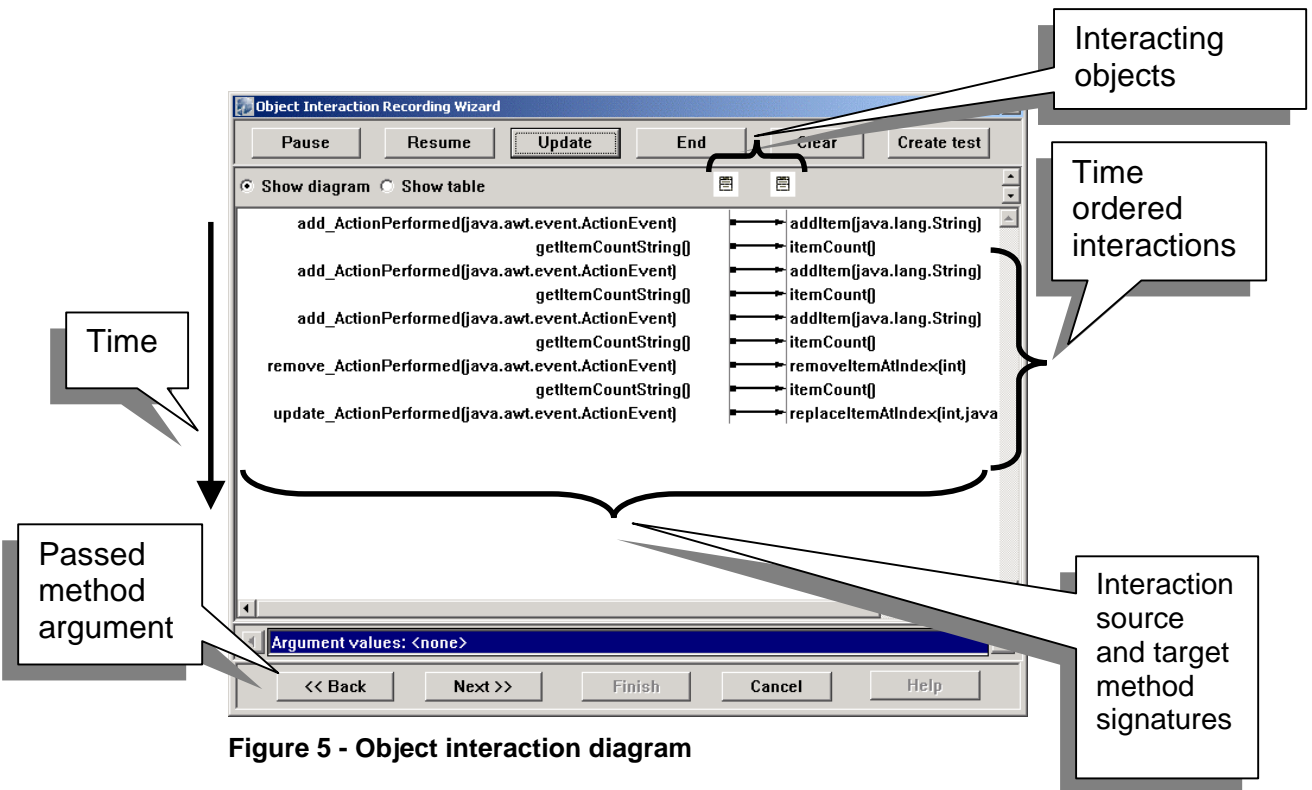


Figure 5 - Object interaction diagram

If you specified to record interactions on startup within the launch panel you may see items shown in the view. Otherwise it will be blank.

Pause button

The Pause button tells JUnit/*profile* to ignore any interactions that may be detected. It has no effect on your application.

Resume button

The Resume button tells JUnit/*profile* to resume recording if it has been paused. Interactions that were ignored while Pause was in effect will now be recorded.

Update button

When JUnit/*profile* records interactions, it holds them in a buffer until you request them to be displayed. When you press the Update button JUnit/*profile* moves the recorded interactions from the buffer to the interactions page.

End button

The End button ends the recording session. In the case where you used the [launching](#) option, the launched VM is shut down and your application closes. In the case where you used the [attaching](#) option, JUnit/*profile* detaches from the VM, but your application continues running.

Clear button

Use the [Clear](#) button to clear the contents of the interactions view. This does not clear any pending interactions. Any interactions that are already buffered will be shown when you press the [Update](#) button.

Create test button

At any time that the interactions view contains interactions, you may request to create a test that reflects those interactions.

Specifically, each recorded interaction represents a method call from one object to another. For interactions where the target object is your object under test you might want to create a test case that recreates those interactions with the object under test.

When you press [Create test](#), JUnit/*profile* opens a *Sequence Test Wizard* that guides you through specifying the information necessary for converting the sequence of recorded interactions into a test case based on:

- The class to generate the test into.
- A test asset name
- The object under test (target of interactions)
- An actor object, whose method calls to the object under test will be invoked by the test case

See [JUnit/generate](#) and [Hands on example of test generation](#) for information on generating tests from object interactions.

Suggested usage

The best way to monitor object interactions is to use a *use case scenario* based approach. A use case scenario is a particular path through or usage of your application under test. Use case scenarios correspond very closely to test cases. You should have in mind a particular set of paths through your application that will exercise your components in various ways. You can use this wizard to capture interactions with components for a particular use case scenario:

1. Bring your application under test into a particular start state
2. Begin recording, or if it has been recording, clear any existing recorded interactions (with the [Clear](#) button)

3. Perform whatever steps you need to perform in order to exercise your application for a particular use case. If your application has a visual interface, interact with that interface.
4. Pausing recording (with the [Pause](#) button)
5. View recorded interactions for that particular use case scenario
6. Optional: generate test based on recorded interactions (with the [Create test](#) button)

JUnit/load

Problem: It is often necessary to characterize the maximum number of clients that shared resources such as application or database servers can handle, while maintaining a reasonable response rate.

Solution: Create tests that access shared resources. Run multiple copies of those tests concurrently, measuring the execution time of the test, and gradually increasing the number of concurrent tests to see how many concurrent copies

Concepts

JUnit/load provides the ability to distribute and run the same test on multiple machines (CPUs), Java Virtual machines (JVMs) and threads. This provides solutions to both problems above because it gives the ability to run multiple copies of a test client, on as many machines as you need, in order to load a shared resource, and it also enables you to run many threads on the same machine, in order to flush out thread safety issues.

JUnit/load provides agents that you install and start on each machine that you wish to run tests on. You can run one or more agents on a given machine. Each agent will run one or more threads as necessary to balance the load.

From the perspective of a shared resource accessed by your test, the test threads act as virtual users of that resource.

When you run a test under JUnit/load you specify the number of concurrent copies of the test to start and end with, which agents to distribute them on and the number of copies of the test to increment by from run to run.

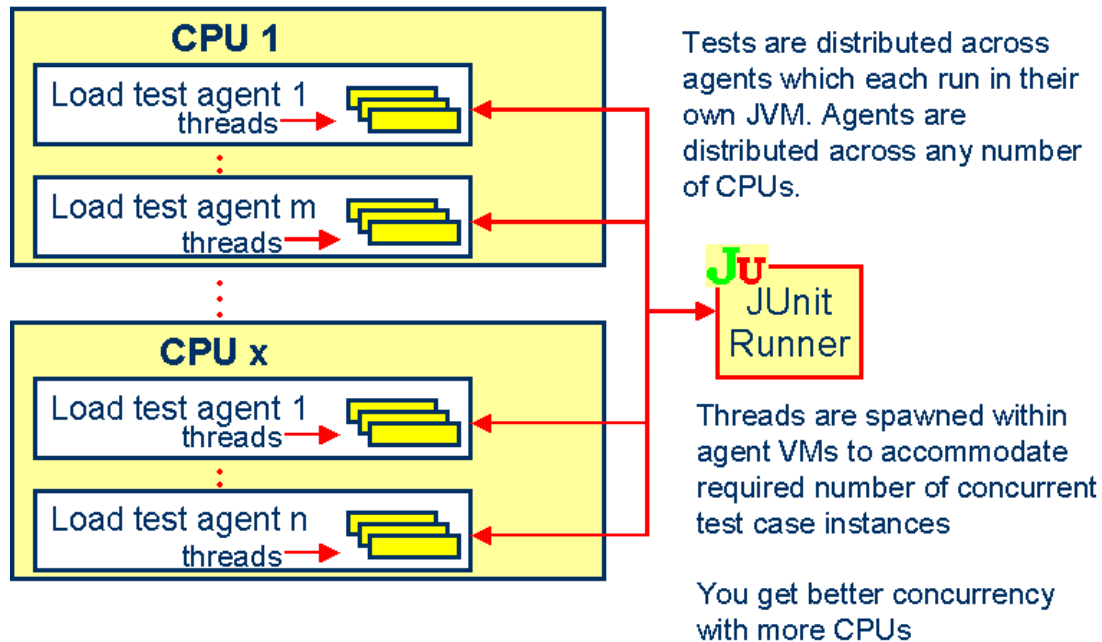


Figure 6 - JUnit/load topology

The above figure shows the way tests are distributed. They are ultimately run on threads within agents that themselves each run in a JVM on a machine. For a given test run, JUnit/load will evenly distribute copies to threads on agents in order to balance the test case load.

As an example, consider the following configuration:

Property	Setting
Agents	5
Start	10
End	1000
Increment	25

Under this configuration, the first run would run 10 copies of the test, distributed across 5 agents, so each agent would run 2 threads to handle the load (2 threads x 5 agents = 10 copies).

The next run would increment the number of copies of the test by 25 to 35 (10 + 25). In this case it would run 35 copies of the test, distributed across 5 agents, so each agent would run 7 threads to handle the load (7 threads x 5 agents = 35 copies).

To summarize:

Run	Agents	Copies	Threads
First	5	10	2
Second	5	35	7
Intermediate runs	5	60-960	12-192
Last	5	985	197

You should be careful to not try to run too many copies of your test at a time on a given machine. If you do, the overall load will be throttled by the limited ability of the client to execute concurrent tests and your results will be skewed.

When load testing, the idea is to get a much concurrency as possible for a given load. If your operating system's process model provides better concurrency than the JVM's thread model, you should run multiple agents on the same machine. If a machine cannot handle the load of running many copies of a test, you should add more machines for the agents to run on.

Setting up for load testing

To set up for load testing, simply start the load test execution agents, and use the Preferences to specify the *start load*, *end load*, *increment* and *locations of the agents*. You do this under the JUnit/load tab in the Preferences:

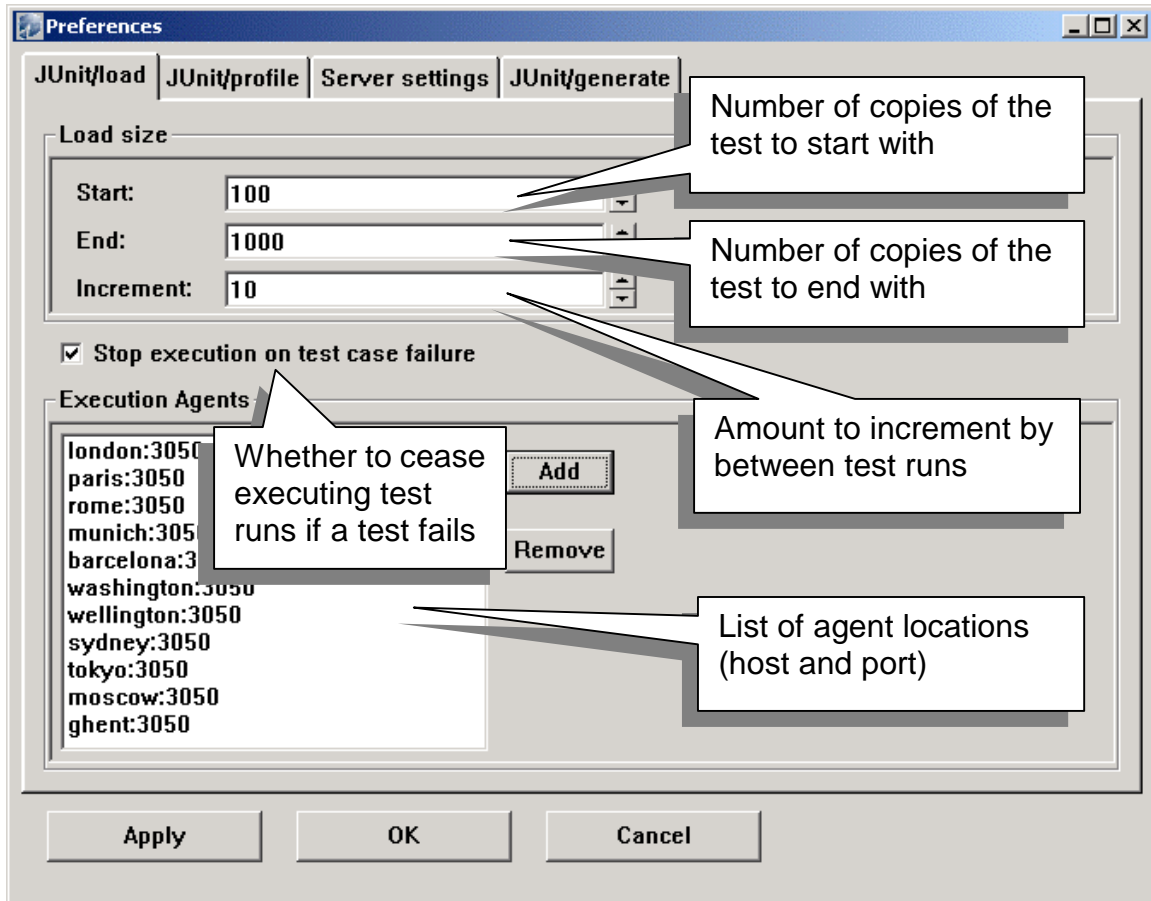


Figure 7 - Load testing preferences

Name		Description
Load size	Start	The number of concurrent test instances to start with (on the first test run)
	End	The number of concurrent test instances to end with (on the last test run)
	Increment	The number of test instances to add for each successive test run
Stop execution on test failure	If this is checked and a failure or error is detected during a test run, JUnit/load will not execute any subsequent test runs.	
Execution agents	A list of the addresses and ports of all execution agents. You must start load test agents on the specified machines, passing the specified ports before running the	

```
tests.
```

To start a load test agent, run the following in a command prompt with or without specifying a port:

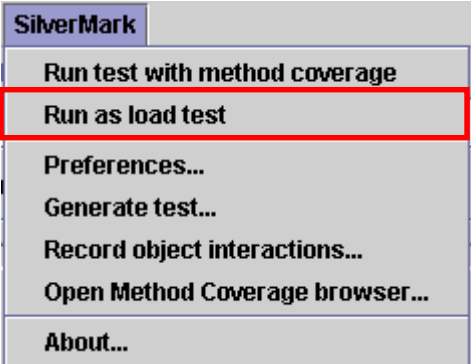
```
<install_dir>\smejunit\startloadtestagent.bat  
or  
<install_dir>\smejunit\startloadtestagent.bat [port]
```

If you don't specify a port to read from, a default of **3050** is used.

For example:

```
c:\silvermark\smejunit\startloadtestagent 4025
```

With the load test execution agents all waiting to do their job the next step is to enter the name of your test in the JUnit GUI as you normally would and select Run as load test... in the SilverMark menu.



Once the results for the first test run appear a graph will open and begin to accumulate execution times for each test run.

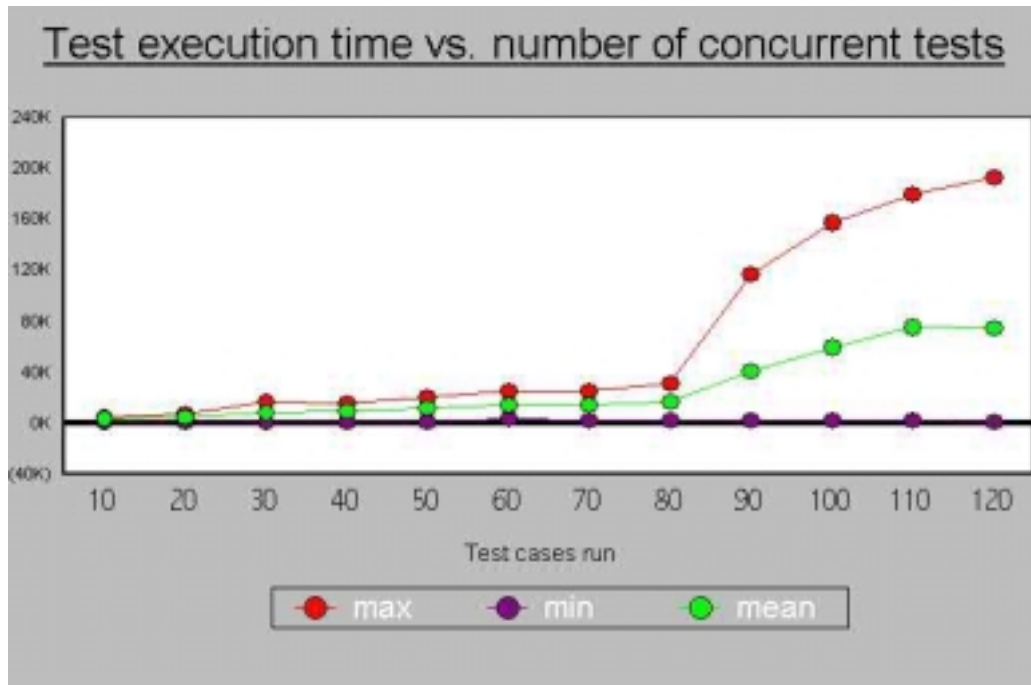


Figure 8 - Execution times graph

Upon completion, failures and errors for all test runs will be shown in the usual place in the JUnit UI.

To stop the load test agents, use Ctrl-c in the command prompt or simply close it.

Using JUnit/load with JUnitPerf

JUnitPerf is an open-source collection of test decorators, available from [Clarkware Consulting, Inc.](http://www.clarkware.com) JUnitPerf enables you to decorate any JUnit test with the following JUnit test decorators: (copied from the JUnitPerf documentation):

TimedTest

A `TimedTest` is a test decorator that runs a test and measures the elapsed time of the test.

A `TimedTest` is constructed with a specified maximum elapsed time. By default, a `TimedTest` will wait for the completion of its decorated test and then fail if the maximum elapsed time was exceeded. Alternatively, a `TimedTest` can be constructed to immediately signal a failure when the maximum elapsed time of its decorated test is exceeded.

LoadTest

A `LoadTest` is a test decorator that runs a test with a simulated number of concurrent users and iterations.

The load can be ramped by registering a pluggable `Timer` instance that prescribes a delay between the addition of each concurrent user. A `ConstantTimer` has a constant delay, with a zero value indicating that all users will be started simultaneously. A `RandomTimer` has a random delay with a uniformly distributed variation.

ThreadedTest

A test decorator that runs a test in a separate thread.

A copy of JUnitPerf is provided in the Enhanced JUnit distribution. Check the [readme.pdf](#) for the location of the JUnitPerf files. You may also want to visit the [JUnitPerf web site](#) to check for updates.

The only real area of overlap between JUnitPerf and JUnit/load is the functionality provided by the LoadTest decorator. Both frameworks provide the ability to run tests on a separate thread, although JUnitPerf is limited to distributing tests on threads on a single virtual machine, whereas JUnit/load can distribute them across multiple VMs and physical machines.

Using TimedTest with JUnit/load

When you decorate a test with the TimedTest test decorator, you add the ability to not only fail the test if the test takes too long to execute, but end the test as well. You can use this in conjunction with JUnit/load to detect when increased load has caused a test to exceed a maximum time limit.

When a TimedTest detects that a test has exceeded the specified maximum execution time it throws a `junit.framework.AssertionFailedError` with information, **“Maximum time exceeded! Expected was ___ms but was ___ms”**.

If you check the [Stop execution on test case failure box](#) in the JUnit/load preferences, no subsequent load test runs will be spawned after detection of the error. Recall that each incremental advance in the number of concurrently executing tests between the start and end limits is called a test run. For example, if you specify a load test with the following parameters

- start load = 10
- end load = 100
- increment = 10

and at the point where 50 tests are being run, one of the tests' execution time exceeds the limit specified for TimedTest, JUnit/load will not run with the next amount, which would be 60 concurrent test instances.

If you do not check the [Stop execution on test case failure box](#) in the JUnit/load preferences, all test runs will be performed and each individual failure will be logged in the JUnit failures list.

Using LoadTest with JUnit/load

Like the TestRunner decorator, JUnit/load provides a mechanism for distributing multiple, concurrent instances of a test across threads. In addition, JUnit/load also provides the ability to distribute concurrent test instances across multiple VMs and CPUs.

What is interesting about the LoadTest decorator is that it gives you the ability to incrementally ramp the number of test instances (one per thread) over time, where the addition of each instance is separated by a value that defines the delay between the addition of each concurrent instance.

This adds an additional dimension to load testing. For example, suppose you have a JUnit test that accesses a shared resource such as a database or web site. You can use JUnit/load to test the resiliency of that resource by distributing those test cases to multiple machines, increasing the number of instances for each test run. You can then add the LoadTest decorator to the equation in order to spawn additional instances of greater numbers of tests over a given time for each test run.

Consider the following example:

- start load = 10
- end load = 100
- increment = 10

Run	Agents	Concurrent instances	Concurrent instances per agent
1	5	10	2
2	5	20	4
3	5	30	6
4	5	40	8
5	5	50	10

If you add a LoadTest decorator, that ramps from 1 to 10 concurrent threads, adding one thread per second you would have the following:

Run	Agents	Concurrent instances	Concurrent instances per agent
1	5	10-100	2-20
2	5	20-200	4-40
3	5	30-300	6-60

4	5	40-400	8-80
5	5	50-500	10-100

In this case, each test run would start with an instantaneous load and then incrementally add additional test instances.

JUnit/generate

Problem 1: Creating JUnit tests requires the odd combination of both tedious repetition as well as careful planning and design. On the one hand, there is a lot of test creation that is common from class to class and method to method. On the other hand, it is important to use sound design and architectural principals to guide your test creation.

Problem 2: Often in the case of existing systems, developers are required to maintain code that does not have tests for it.

Solution for problem 1: Use JUnit/*generate* to automatically generate tests for you based on the structure of your classes and common test design patterns.

Solution for problem 2: Use JUnit/*generate* to automatically generate tests directly from object interactions recorded by JUnit/*profile*.

Generating tests for classes

To generate tests for classes you must first select your preferred generation options in the Test Generation settings of the Enhanced JUnit preferences. These preferences settings are divided into two pages:

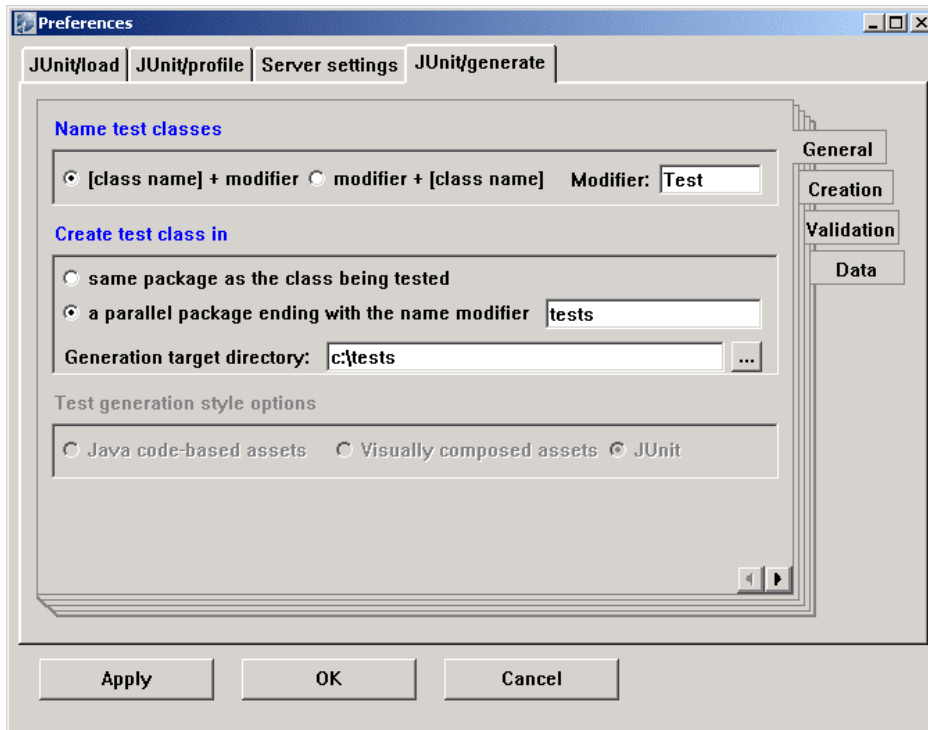


Figure 9 - Test Generation preferences (general page)

Name	Description
<p>Name test classes:</p> <ul style="list-style-type: none"> • [class name] + modifier • modifier + [class name] 	<p>JUnit/<i>generate</i> generates classes to represent test suites. Use the radio button set to specify whether to insert a name modifier in front of or after the name of the class.</p> <p>For example if the name of a class under test is MyClass and the name modifier is Test, JUnit/<i>generate</i> will generate a suite class named MyClassTest or TestMyClass for it, based on this setting.</p>
<p>Create test class in: same package as the class being tested</p>	<p>Selecting this option causes test classes to be generated in the same package as the class under test.</p>

<p>Create test class in: a parallel package ending with the name modifier _____</p>	<p>Selecting this option causes test classes to be generated in a package based on that of the class under test, with the given name modifier appended.</p> <p>For example, if a class under test is located in com.mybank and the name modifier is tests, then the test package used will be com.mybank.tests.</p>
<p>Generation target directory</p>	<p>Use this to specify the root directory to place generated tests in. For example, if you specify c:\tests and generate for myproject.tests.MyClass, you will generate the file c:\tests\myproject\tests\MyClass.java. If you do not specify a generation target directory, a JUnit/generate will use the classpath directory that most closely matches the directory structure of the generated class.</p>
<p>Test generation style options</p>	<p>JUnit style only. Other styles are only supported by SilverMark's Test Mentor.</p>

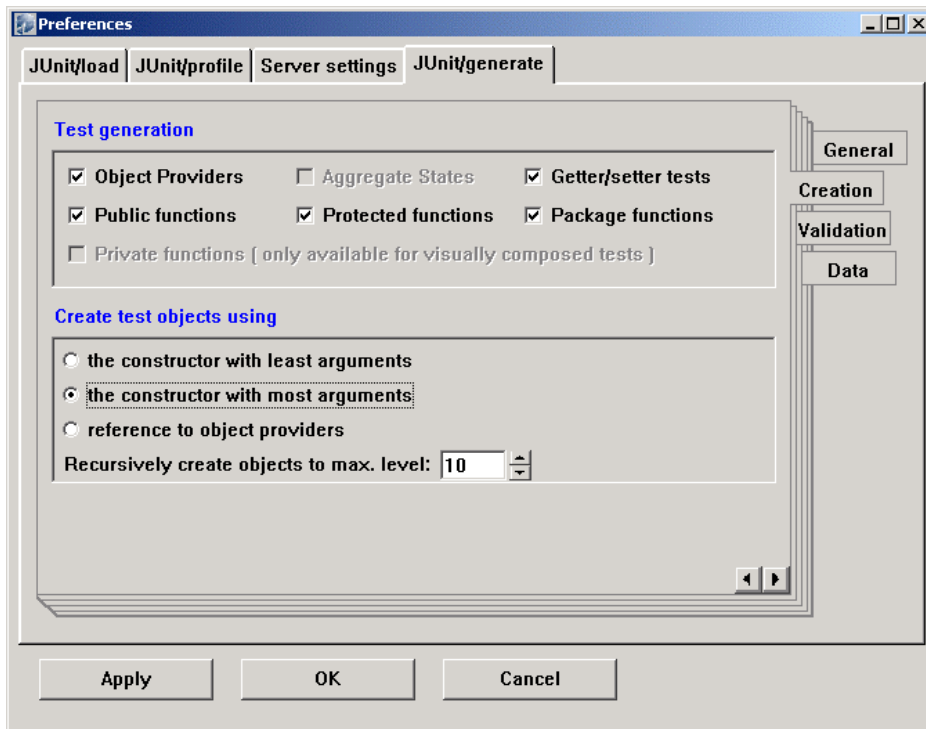
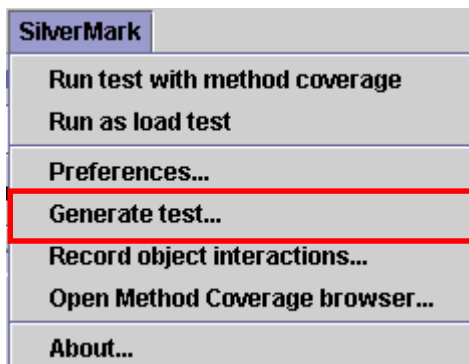


Figure 10 - Test Generation preferences (creation page)

Name	Description
Test Generation: Object providers	Generate test assets that implement the object provider pattern. If you select this, JUnit/ <i>generate</i> will generate assets that return instances of your classes under test. You can then copy or edit the generated assets to return objects configured in certain ways, as needed.
Test Generation: Aggregate states	Generate aggregate state-based test assets that implement the state extraction pattern. If you select this, it will generate assets that represent flattened collections of primitive values within your object under test, as well as gold standard values. You can then copy or edit the generated assets to extract different values, as needed. Note: Only supported by SilverMark's Test Mentor
Test Generation: Getter/setter tests	Generate tests for all methods that match the getter/setter pattern (getxxx, setxxx for variable xxx)
Test Generation: Public functions	Generate tests for methods with the public modifier.
Test Generation: Protected functions	Generate tests for methods with the protected modifier.
Test Generation: Package functions	Generate tests for methods with the package modifier.
Test Generation: Private functions	Generate tests for methods with the private modifier. This feature is only provided by SilverMark's Test Mentor

<p>Create test objects using:</p> <ul style="list-style-type: none"> • the constructor with least arguments • the constructor with most arguments 	<p>When JUnit/<i>generate</i> generates a reference to a method or constructor that takes some object as a parameter, it uses this preference to determine whether to instantiate parameters with the constructor for the parameter type with the most or least arguments. Remember that this is done for you as a convenience and that you can always edit the generated test to pass whichever objects you wish.</p>
<p>Create test objects using: reference to object providers</p>	<p>If you requested to create object providers above, you have the additional option of generating tests that pass parameter values using the configured object values returned by those object providers, instead of instantiating values directly using constructors.</p>
<p>Recursively create objects to max. level: ____</p>	<p>When JUnit/<i>generate</i> instantiates objects using constructors, often those constructors take objects as parameters. This setting specifies the number of levels of constructors to generate for parameters that require parameters that require parameters... and on and on...</p>

Once you have set your preferences, simply select the Generate test... item of the SilverMark menu.



You will then be prompted for the name of the class to generate for:

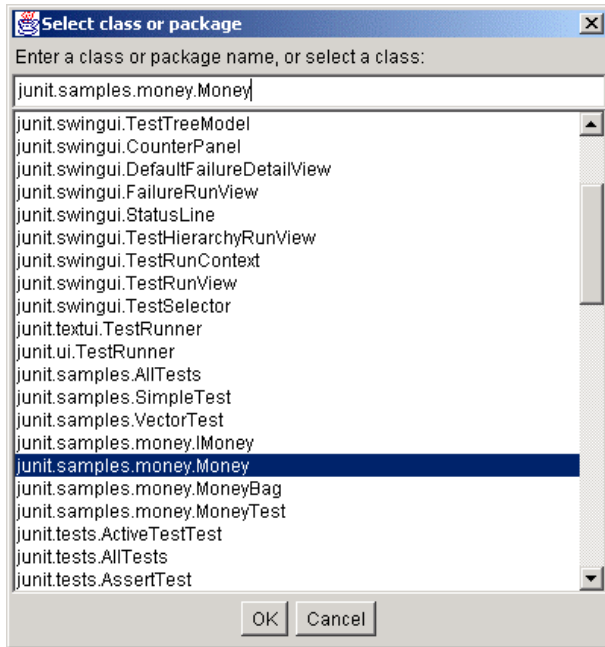


Figure 11 - Prompt for class or package to generate for

Enter a class or package name, or select from the list of classes that are visible in the JUnit startup classpath. JUnit/*generate* will then generate a test class according to the preferences you set into a test class.

Note: JUnit/*generate* generates tests into directory specified by the Generation target directory preference or if none, it searches the classpaths for a directory structure that most closely matches the package structure of the generated test class, or places it in the class in the first classpath entry.

The generated code includes `setUp()`, `tearDown()`, as well as `test___` methods for each public method that many JUnit test generation programs provide. The aspect that JUnit/*generate* adds is that it generates starter tests for each method it generates for. These tests:

1. Instantiate the object under test using the instantiation setting from the preferences of
 - Constructor with least arguments
 - Constructor with most arguments
 - Reference to Object provider method. Object provider methods are methods that represent test assets that return object instances configured in a certain way. JUnit/*generate* gives you the option of generating reusable test assets that provide instances of objects under test. JUnit/*generate* will also generate references to those assets to provide instances, rather than generating object instantiation in-line

within your tests. Breaking out object instantiation into a separate asset makes it very easy to plug in other instantiation schemes and input and reference object variations.

2. Apply a stimulus (call method)
3. Apply postcondition validation
 - For void methods, uses assertEquals for the toString() form of the object under test
 - For methods that return values, uses assertEquals for the returned object.

As described above, JUnit/*generate* also generates object provider methods if so requested in the Preferences. You can then take the generated methods and further configure them or use them as a template for creating more interesting and useful ones.

Generating tests from recorded object interactions

JUnit/*generate* adds the ability to generate tests from object interactions recorded by JUnit/*profile*.

To engage this feature, simply record a set of interactions with an object and press the Create test button in the Object interaction recording wizard. This will launch the Sequence test wizard which will lead you through the process of transforming the sequence of recorded interactions into a test.

There are two panels to the wizard. In the first panel you specify information about the class to generate the test into.

The idea behind generating tests from recorded interactions is that there is an object that is the target of the interactions and an object that is the source. In generating a test from the interactions, the test case will take the place of the source, which is known as the *actor*. In the second panel you specify which object within the recorded interactions acts as the test subject and which object acts as the actor.

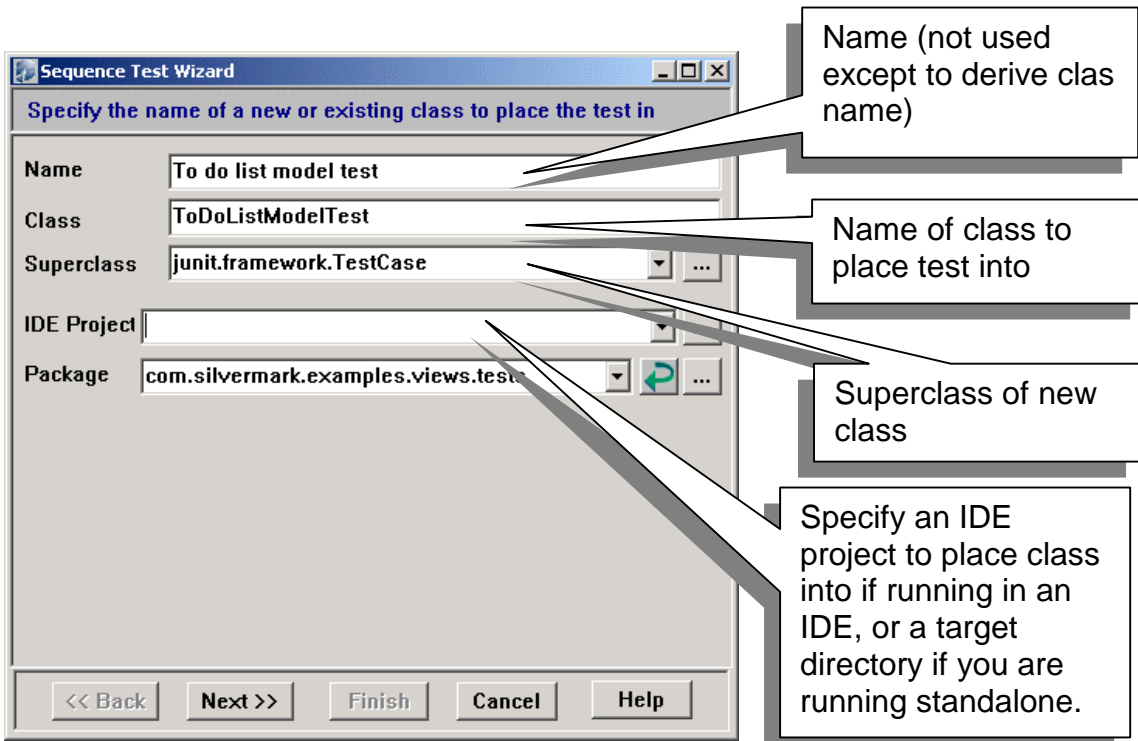


Figure 12 - Sequence test wizard target class specification panel

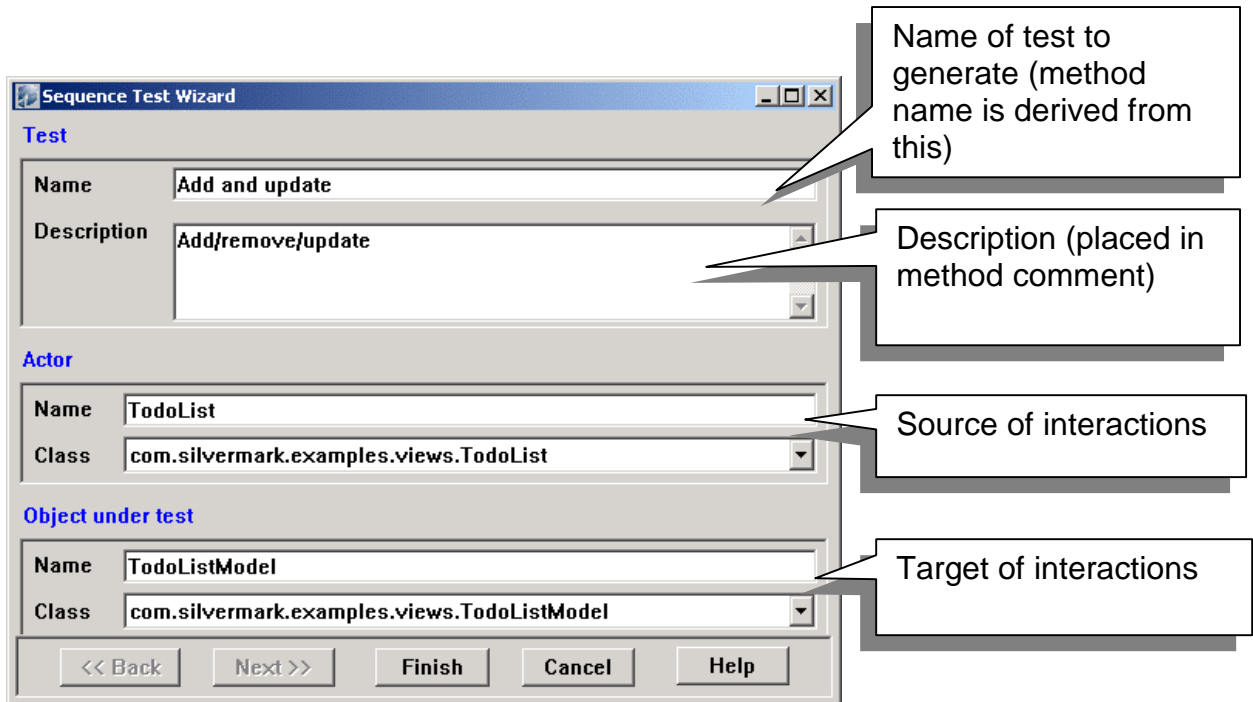
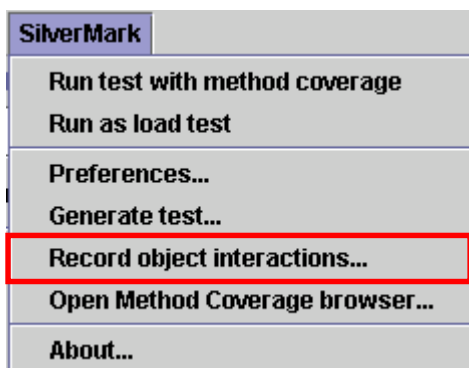


Figure 13 - Sequence test wizard actor and test subject specification panel

Hands-on example of test generation from object interactions

Follow the steps in this example to create a test based on recorded object interactions. This example shows how to create a unit test for a domain object class within a Java application by exercising it from its user interface.

From the JUnit UI, launch object interaction recording by selecting the Record Object Interactions... menu:



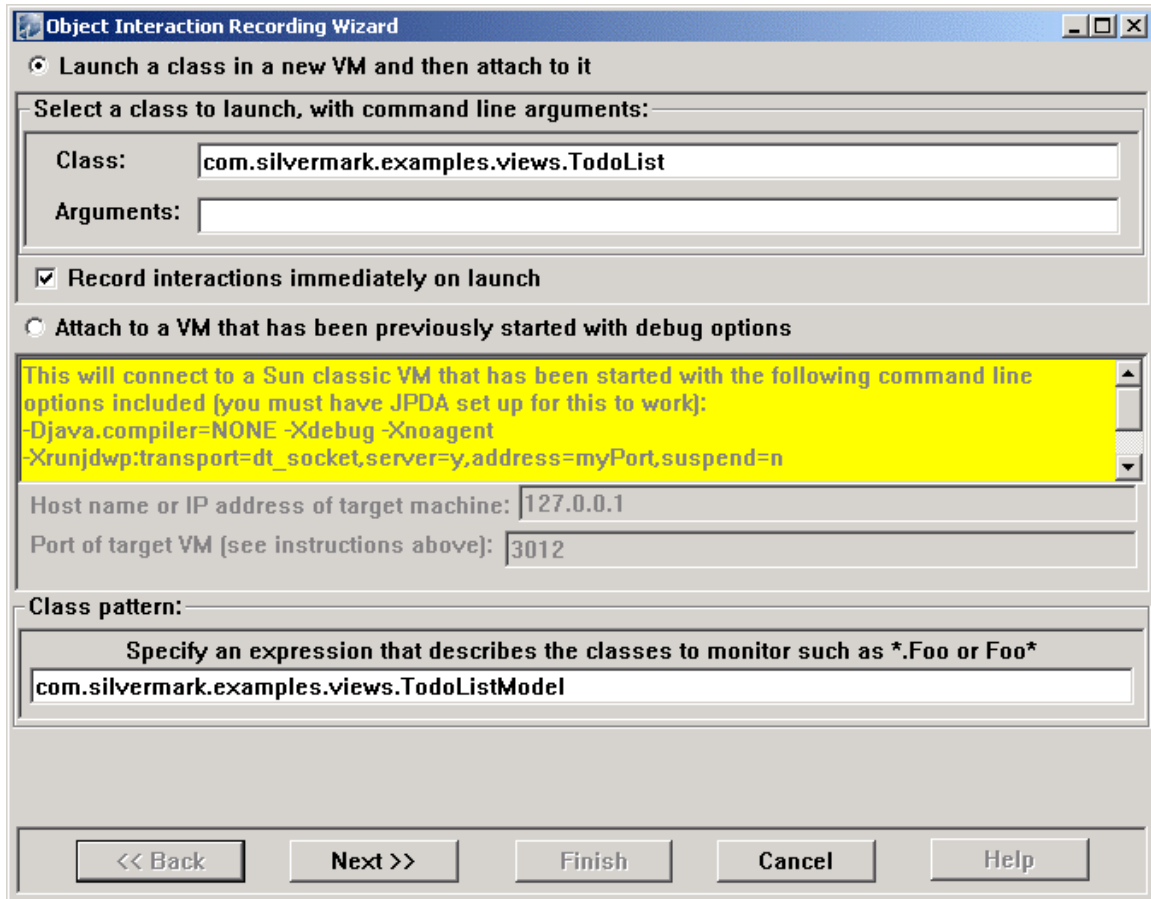


Figure 14 - Launch panel for object interaction recording

1. Fill in the fields as you see them above to specify the class to launch (`com.silvermark.examples.views.TODOList`) and the class to watch interactions with (`com.silvermark.examples.views.TODOListModel`) and press Next>>.

You should see the following:

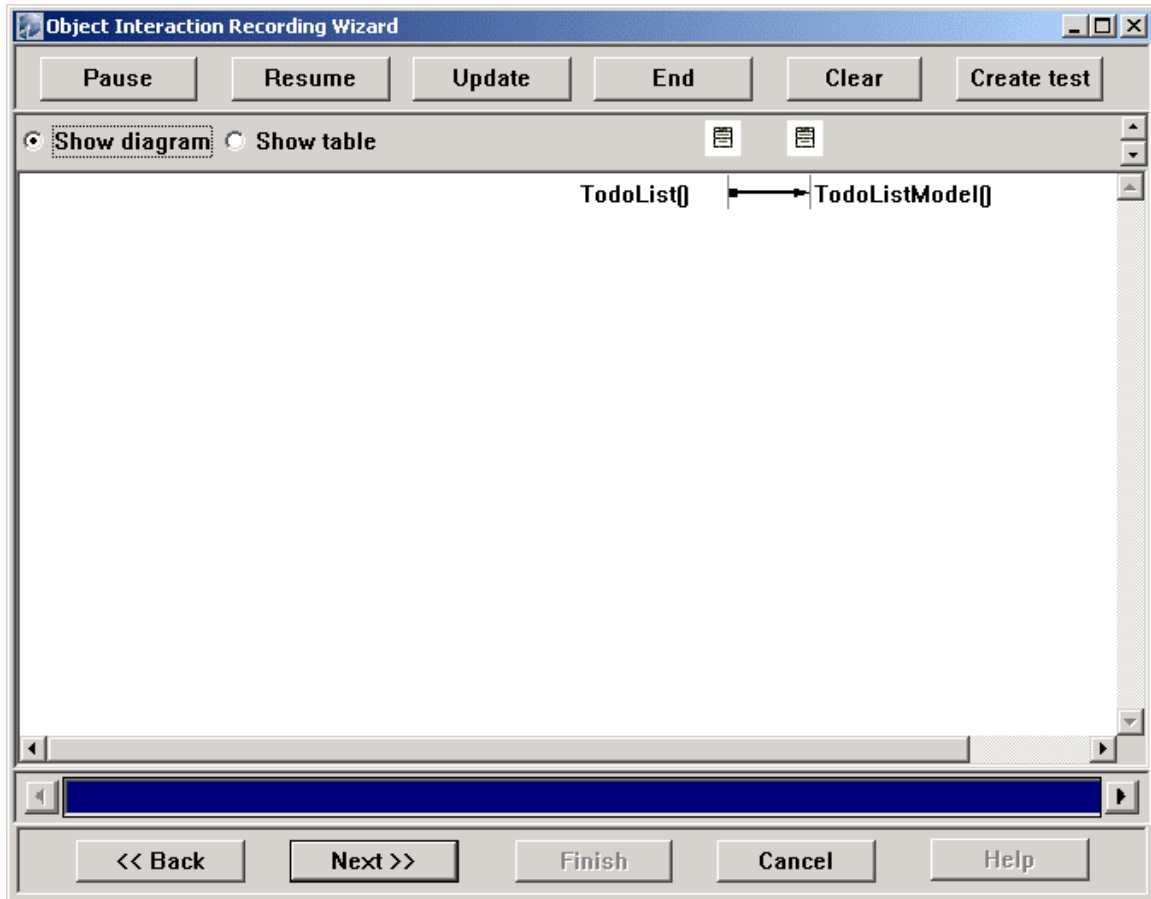



Figure 15 - Interactions panel showing interactions recorded during startup

2. This window shows the interactions with `ToDoListModel` during application startup as a list. The arrow represents an interaction between two objects.

If you hold the cursor over the  icons, you will see the names of the classes corresponding to the interactions. The arrow points to the target, which is the `com.silvermark.examples.views.ToDoListModel` that is being observed. What we see here is that `ToDoList` created an instance of `ToDoListModel` using the `ToDoListModel()` constructor during initialization.

3. Press the Clear button to remove this interaction.
4. Now locate and select the window titled **To-do list**. It might be hidden behind another window:

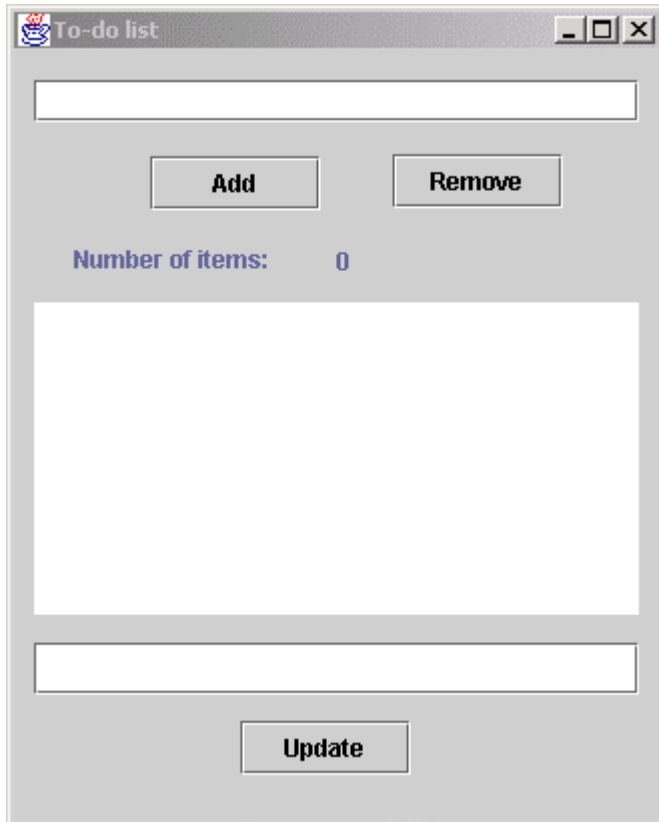


Figure 16 - Example To-do list view

5. Interact with it in the following way:
 - a. Type "Wake up" in the upper text area
 - b. Press Add
 - c. Type "Eat breakfast" in the upper text area
 - d. Press Add
 - e. Type "Go to work" in the upper text area
 - f. Press Add
 - g. Select "Go to work"
 - h. Press Remove
 - i. Select "Eat breakfast"
 - j. Type "Go back to sleep" in the lower text area
 - k. Press Update in the To-do list view
6. Press Update in the Object Interaction Recording Wizard. This will update the view with any interactions recorded since the last update. You should see the following:

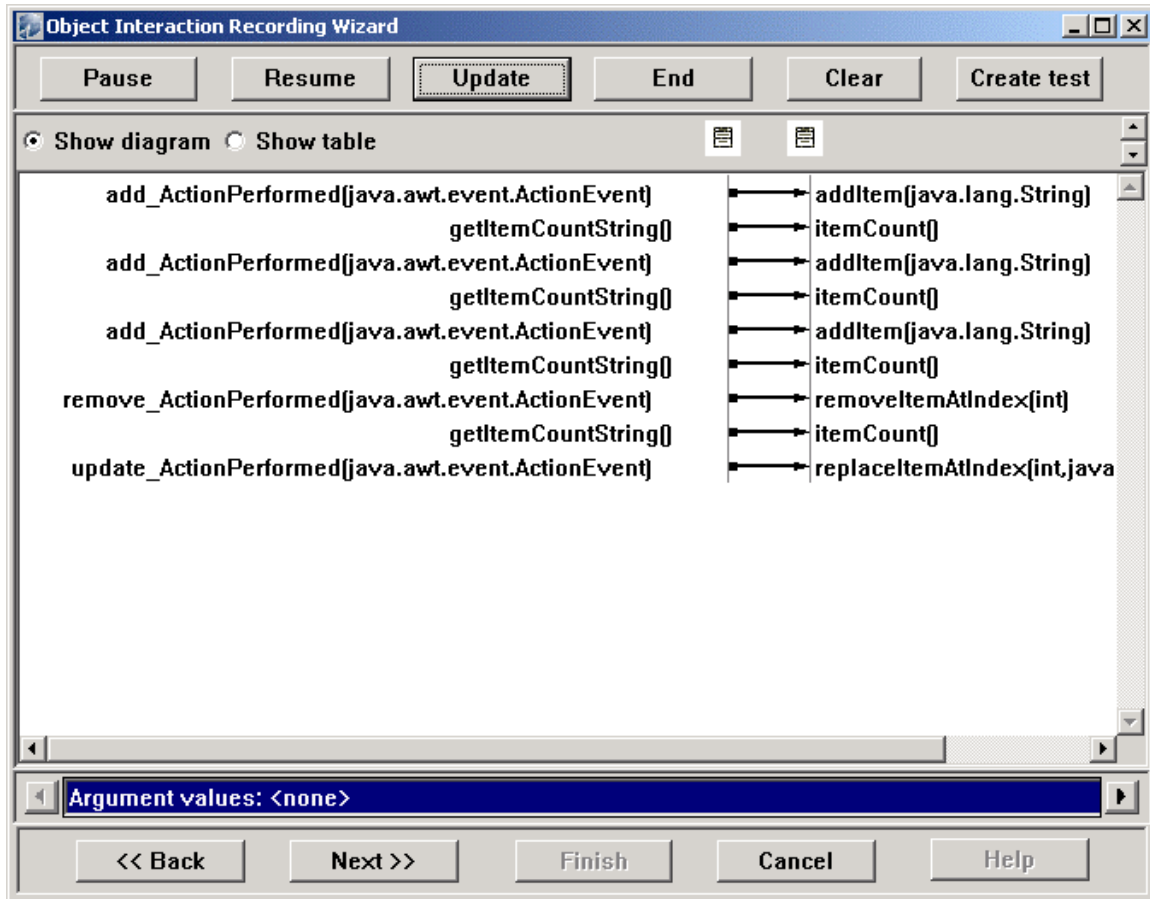


Figure 17 - Recorded interactions

Notice that the object interactions recorded roughly correspond to the user interface interactions you performed. For example **add_ActionPerformed(...)** → **addItem(..)** correspond with pressing the Add button.

These are the actual interactions that occurred between objects as a result of your interactions with the user interface.

7. Select the last item and look in the status view on the bottom. You should see the argument for the `replaceItemAtIndex(int itemIndex, String newItem)` method:



You see the values that correspond to the parameters for the method call.

8. Press Create Test. The following dialog opens:

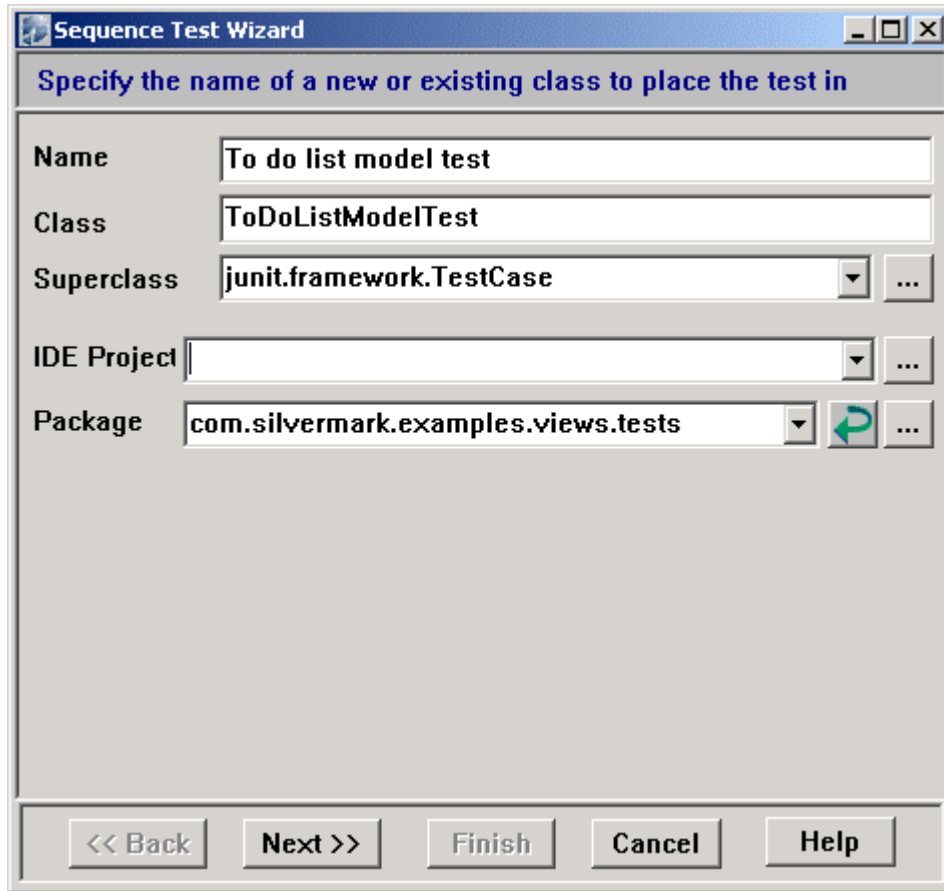


Figure 18 - Sequence test wizard generation class selection panel

Specify a class to place the test that will be generated, as shown above.

9. Press Next >>. You will see the following:

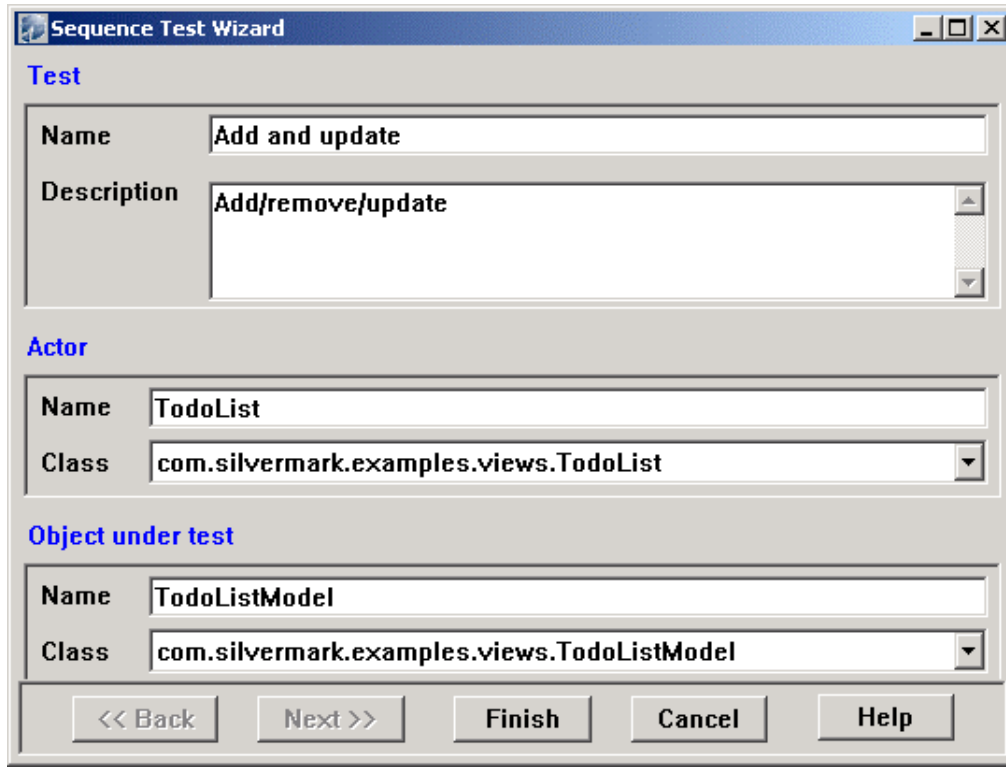


Figure 19 - Sequence test wizard collaborators selection panel

This gives you an opportunity to specify a name and description for the test. In addition, you also must select the actor and object under test from the diagram, from which to generate tests.

The object under test is the test object that will be created and acted upon by the test. It is the target of interactions.

The actor represents the object that acts upon the object under test. JUnit/*generate* takes a best guess based on the first interaction. The generated test will take the place of the actor by performing actions upon the object under test.

For this example you can leave it all as it is and press OK.

This is a test asset based on the recorded interactions. Navigate through the items and you will see that the **Create test subject** step uses a constructor to create an instance of `ToDoListModel`. After that, the test calls methods according to the recorded method calls above, passing recorded parameters.

JUnit/*generate* searches the classpaths for a directory structure that most closely matches the package structure of the generated test class, or places it in the class in the first classpath entry.

The generated test looks like this (with annotation):

```

package com.silvermark.examples.views.tests;

import junit.framework.*;

public class ToDoListModelTest extends TestCase{

    /**
     * Test Case constructor
     *
     * @param name java.lang.String
     */

    public ToDoListModelTest(String name) {
        super(name);
    }

    /**
     * ONE-TIME AUTOMATICALLY GENERATED CODE (WILL NOT BE REGENERATED)
     *
     * @param args java.lang.String[]
     */
    public static void main(String[] args) {
        junit.textui.TestRunner.run(com.silvermark.examples.views.tests.ToDoListModelTest.class);
    }

    /**
     * Sets up the fixture, for example, open a network connection.
     * This method is called before a test is executed.
     *
     * @throws Exception
     */
    protected void setUp() throws Exception {
    }

    /**
     * Tears down the fixture, for example, close a network connection.
     * This method is called after a test is executed.
     *
     * @throws Exception
     */
    protected void tearDown() throws Exception {
    }

    /**
     * Test the Test Add and update
     * Add/remove/update
     *
     *
     * This asset has (9) step(s):
     * <ul>
     * <li>addItem(String)</li>
     * <li>itemCount()</li>
     * <li>addItem(String)</li>
     * <li>itemCount()</li>
     * <li>addItem(String)</li>
     * <li>itemCount()</li>
     */
}

```

```

* <li>removeItemAtIndex(int)</li>
* <li>itemCount()</li>
* <li>replaceItemAtIndex(int,String)</li>
* </ul>
*
* Creation date: (4/13/2002 9:41:50 PM)
* @author Generated by SilverMark's Test Mentor Version 5.4
*/
public void testAddAndUpdate() throws java.lang.Throwable {
    java.lang.Object subject = null;

    // Test Add and update:
    // Test the Test Add and update
    // Add/remove/update

    // Create test subject:
    // Create a test instance of TodoListModel as a
    // the receiver for this test
    subject com.silvermark.examples.views.TODOListModel = new
com.silvermark.examples.views.TODOListModel();

    // addItem(String):
    // Apply the method addItem as a stimulus to the
    // com.silvermark.examples.views.TODOListModel test instance
    subject.addItem("Wake up");

    // itemCount():
    // Apply the method itemCount as a stimulus to the
    // com.silvermark.examples.views.TODOListModel test instance
    subject.itemCount();

    // addItem(String):
    // Apply the method addItem as a stimulus to the
    // com.silvermark.examples.views.TODOListModel test instance
    subject.addItem("Eat breakfast");

    // itemCount():
    // Apply the method itemCount as a stimulus to the
    // com.silvermark.examples.views.TODOListModel test instance
    subject.itemCount();

    // addItem(String):
    // Apply the method addItem as a stimulus to the
    // com.silvermark.examples.views.TODOListModel test instance
    subject.addItem("Go to work");

    // itemCount():
    // Apply the method itemCount as a stimulus to the
    // com.silvermark.examples.views.TODOListModel test instance
    subject.itemCount();

    // removeItemAtIndex(int):
    // Apply the method removeItemAtIndex as a stimulus to the
    // com.silvermark.examples.views.TODOListModel test instance
    subject.removeItemAtIndex(2);

    // itemCount():

```

You may have to set the appropriate constructor here

Recorded interaction

Good place to add an assert(...)

```
// Apply the method itemCount as a stimulus to the
// com.silvermark.examples.views.TODOListModel test instance
subject.itemCount();

// replaceItemAtIndex(int,String):
// Apply the method replaceItemAtIndex as a stimulus to the
// com.silvermark.examples.views.TODOListModel test instance
subject.replaceItemAtIndex(1,
"Go back to sleep");
}
}
```

As follow-on activity to generating tests from recordings, you might:

- Adjust the names of the generated suite and asset to suite your own conventions
- Add validation to ensure the target object and returned values are in an expected state. These might be added after each step, certain steps or at the end of the test.
- Make the steps data-driven, to automatically iterate over a range of input values

10. To finish up, press End to stop recording and close down the launched VM.

JUnit/util

Data-driven testing

Problem: JUnit does not provide a *simple* way to make tests data-driven. A data-driven test is one that executes one or more times over a range of different input and output values, usually acquired from a test data file or database, possibly stored on a server. While reading data from a file is relatively straightforward in Java, there are matters of parsing and type conversion to be considered.

Solution: Use the JUnit/*util* supplied classes for iterating over test data. All of these classes implement the `java.util.Enumeration` interface for traversing data files that are specified in terms of a URL. The classes are provided in the `com.silvermark.stm.base` package.

Note: You can also use JXUnit (<http://jxunit.sourceforge.net/>) for data-driven testing, which appears to be powerful, but not very simple to use.

Iterating over text records

This is the simplest interface as all it does is provide undigested access to records within a text file.

Example:

```
FileReader reader = new FileReader("file:///c:/testdata.dat");
while (reader.hasMoreElements() ) {
    String nextRecord = (String) reader.nextElement();
    System.out.println(nextRecord)
}
```

In this example, you can see that a URL String representation is passed to the constructor. You can pass an actual `java.net.URL` to the constructor if you already have one.

Simply use the enumeration interface (`hasMoreElements()` and `nextElement()`) to traverse the records.

Iterating over delimited records

We use the notion of a delimited record as a text record that contains values delimited by a separator. `JUnit/util` adds an API for iterating over delimited records and extracting values. The best way to illustrate this is through an example. Consider the following file of bank account data:

```
accountNumber,name,type,balance
int,String,char,float
1111,Thurston Howell III,c,99999999.0
5555,Professor Roy Hinkley,c,35000.0
0000,Willie Gilligan,c,0.56
```

Notice that the elements of each record are separated by a comma (,). Also notice that the first two records do not contain data, but rather descriptive information. The first record contains name information and the second contains type information. If we look at the information as it would appear in a spreadsheet, it becomes clearer.

accountNumber	Name	type	balance
Int	String	char	float
1111	Thurston Howell III	c	99999999.0
5555	Professor Roy Hinkley	c	35000.0
0000	Willie Gilligan	c	0.56

The idea is that each column corresponds to some data value whose name is given in the first record and whose type is given in the second. JUnit/*util* provides an enumeration interface for iterating over records. Instead of returning the String form of each record, it returns a `java.util.Hashtable` whose keys are the given column names and whose values are the corresponding values. Consider the following example:

```
DelimitedFileReader reader =
    new DelimitedFileReader(
        "file:///c:/testdata.dat",    // URL String for data file
        ",",                          // Delimiter
        true,                          // 1st record has variable names
        true);                          // 2nd record has variable types

// Read all records from delimited text file.
// Records are parsed and provided as Hashtable
while (reader.hasMoreElements()) {
    Hashtable data = (Hashtable) reader.nextElement();
    System.out.println(data);
}
```

Providing variable names and types in the data file is optional. If you don't provide names, the default is `field1`, `field2`, ... etc. If you don't provide types, String is assumed.

You might ask yourself, "why not provide the data as XML?" The answer to this is that we've found that record oriented, delimiter-separated data files are much easier to create and maintain for use in test cases – especially if you have a spreadsheet available. Often it is the customer or domain expert who provides the test data files, and creating XML is slightly out of their reach. In other words, this is the simplest thing that could possibly work.

Keyed delimited text record searches

Some times all you need for a particular test is a subset of the data that is available in a test data file. For this case, you can use `KeyedDelimitedFileReader` to only provide records that match a particular key/value pair. Consider this example:

```
KeyedDelimitedFileReader reader =
    new KeyedDelimitedFileReader(
        "file:///c:/testdata.dat",    // URL String for data file
        ",",                          // Delimiter
        true,                          // 1st record has variable names
        true,                          // 2nd record has variable types
        "name",                        // Key (column name)
        "Willie Gilligan");           // Value to match
```

```

while (reader.hasMoreElements()) {
    Hashtable data = (Hashtable) r.nextElement();
    System.out.println(data);
}

```

Usage is nearly identical to `DelimitedFileReader` with the addition of a key and value pair. The enumeration interface will only return items whose given field contains the given value. Values must be String values as they appear in the data file.

Execution timing

JUnit/*util* provides the ability to perform fine grained time-value assertions with the `com.silvermark.loadtest.ExecutionTimer` class.

Timer lifecycle operations

Operation	Description
<code>void start()</code>	Start the timer.
<code>long checkpoint()</code>	Return the current accumulated time in milliseconds. This does not effect the current accumulated time.
<code>long pause()</code>	Return the current accumulated time in milliseconds and cease accumulating time.
<code>void resume()</code>	Resume accumulating time
<code>long end()</code>	Return the current accumulated time in milliseconds and cease accumulating time. This simply calls <code>pause()</code> , however it is useful if your intention is to communicate to the reader of the code that you are done timing.
<code>void reset()</code>	Reset accumulated time to zero and cease accumulating additional time

Timer assertion operations

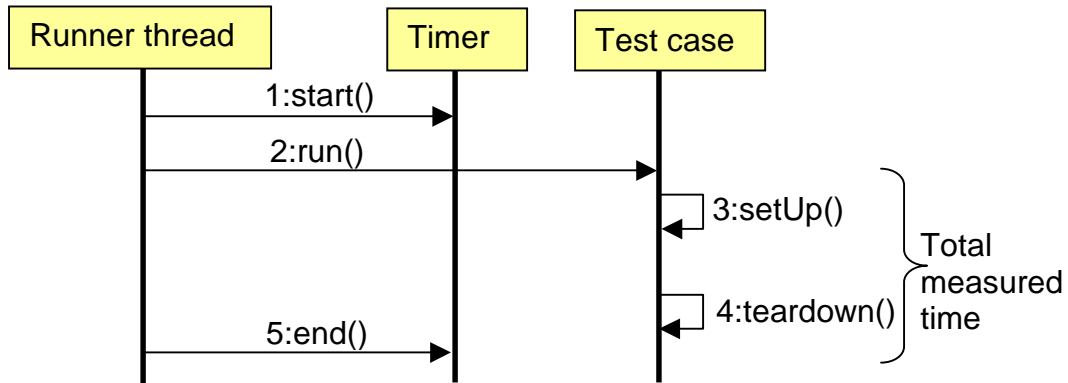
Operation	Description
<code>void assertTimeOutsideOfBoundary(String message,</code>	Assert that the current execution time is outside of (greater than) a specified

long boundary)	boundary time
void assertTimeWithinBoundary(String message, long boundary)	Assert that the current execution time is within (less than or equal to) a specified boundary time
void assertTimeWithinBounds(String message, long startBoundary, long endBoundary)	Assert that the current execution time is within specified start and end boundary times: startBoundary <= time <= endBoundary

Use these operations to add fine-grained execution timing within test cases.

Special load test timing operation

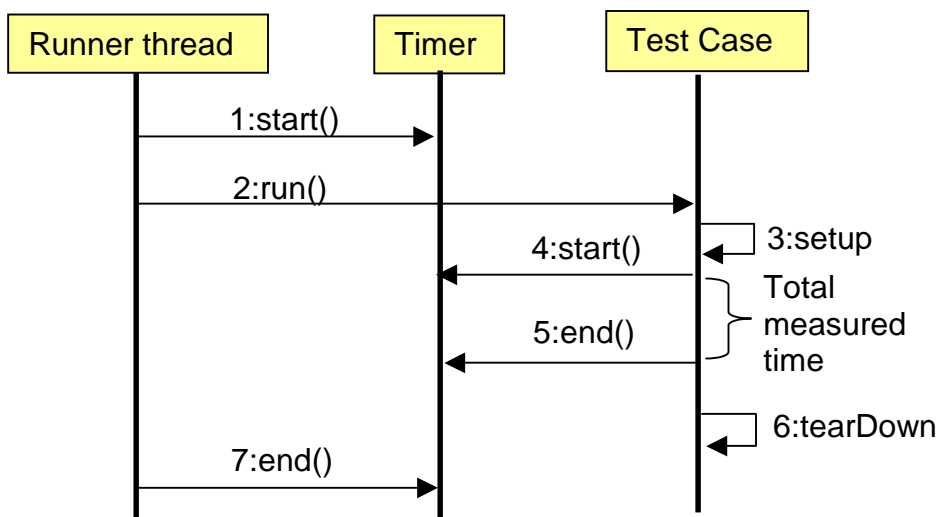
An instance of ExecutionTimer is used on each load test agent thread to measure a test case's execution time. Timing is started immediately before executing the test case. The accumulated time at the end of the test case is used for the test case's execution time. This timing may include setup and teardown, which is not really relevant to the load test timing. For example:



Notice that the total time for running the test on this the load test runner thread includes `setUp()` and `tearDown()`.

Fortunately `JUnit/util` provides easy access to the timer instance so you can adjust the time to include just the parts of your test that are important in the timing. You can simply override the timer `start()`, `pause()`, `resume()`, and `end()` right from within your test.

To access the timer instance for the current load test agent runner thread, simply use `com.silvermark.loadtest.ExecutionTimer.current()`. Note that this will return null if it is not being run from a load test agent runner thread.



In the above example, you can see that `start()` was overridden by calling it again from within the test. Calling `end()` within the test ceased accumulating time. The call to `end()` by the runner thread then has no effect.

Appendix A – The Enhanced JUnit Architecture

Enhanced JUnit builds on to JUnit without changing it. Like JUnit there are Swing and AWT test runners. The AWT and Swing test runners are `com.silvermark.junit.awtui.TestRunner` and `com.silvermark.junit.swingui.TestRunner`. They inherit from `junit.awtui.TestRunner` and `junit.swingui.TestRunner`, respectively.

The Enhanced JUnit test runners request services of the Enhanced JUnit Workbench, which is started separately.

Note: Enhanced JUnit is actually two separately running applications (the Enhanced JUnit Runner and the Enhanced JUnit Workbench) that communicate via a socket. Actually, this is not entirely correct. The load test agents add more separately running applications but they are not discussed in this section.

The Enhanced JUnit Workbench itself is derived from SilverMark's Test Mentor Test Workbench. The workbench performs the test generation, displays load test and profiling results, as well as preferences. You know when the workbench is running because it shows a small console window in the lower right-hand corner of the screen. In general you can minimize and ignore the console.

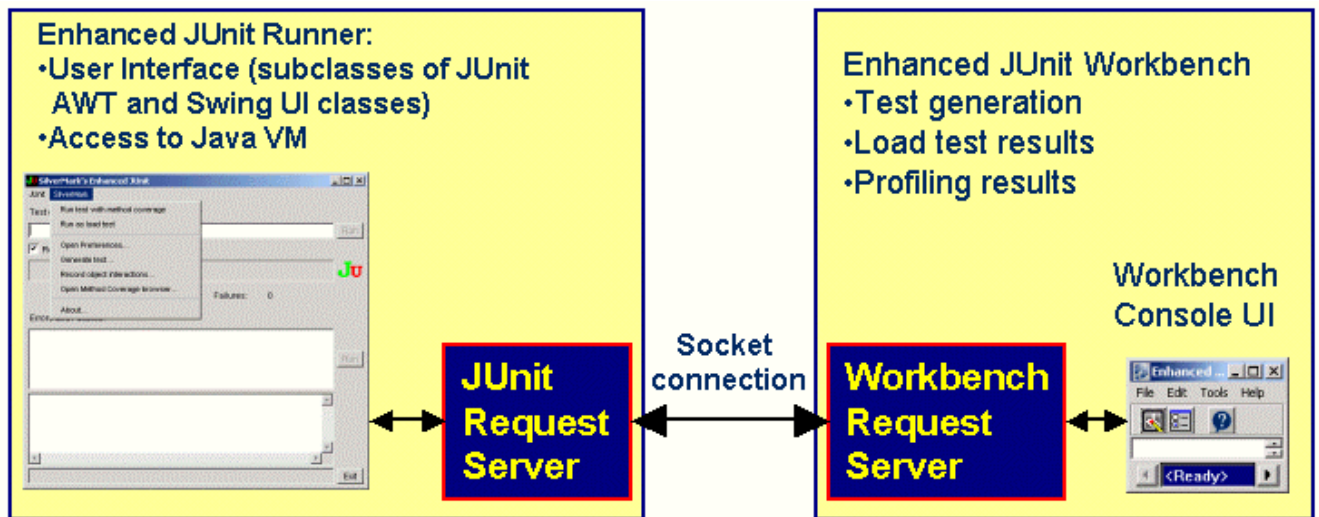


Figure 20 – Enhanced JUnit runner and Workbench relationship

Like the JUnit test runner views, the Enhanced JUnit Runner comes in AWT and Swing versions. In fact, the Enhanced JUnit Runner classes are extensions of the JUnit AWT and Swing test runner classes that have been enhanced to provide a menu of utilities (under the SilverMark menu), as well as some other services for the benefit of the Enhanced JUnit Workbench.

The Enhanced JUnit Runner and Workbench use light-weight request servers to communicate with each other via a socket connection.

Note 1: You don't really need to understand anything about the JUnit and Workbench servers. Your only responsibility in their regard is to make sure the ports that they listen for requests on are not used by any other applications.

Note 2: By default the Enhanced JUnit Request Server listens for requests on port 3000 and the Workbench request server listens for requests on port 3002.

Tip: When the Enhanced JUnit runner UI starts, it searches for a running Workbench. If it does not find one, it launches it. Otherwise, it uses the one that is already running. This means that you can start and stop the Enhanced JUnit runner UI without any need to stop the Workbench.

Appendix B - Installation and setup

Enhanced JUnit is fairly simple to install. In fact, if you install to c:\silvermark and just want to run the JUnit and SilverMark examples, there is no setup required at all.

Otherwise, after you unzip the **smejunit.zip** file into a directory (which we will refer to as **<install_dir>**), you will need to edit two files to set a few items such as classpaths for the test classes and classes under test:

File name	Purpose
<install_dir>\smejunit\setvars.bat	This file sets two variables: JAVA_HOME – This is usually set for you by your Java SDK installation, but if not, you can set it here to the location of your Java SDK DIRS_AND_JARS - Set this to your test classes and classes under test. It is set in the

	'factory' to point to the JUnit and SilverMark examples provided
<install_dir>\smejunit\lib\smejunit.properties	This file configures the Enhanced JUnit runner

Setting up setvars.bat

Ensure JAVA_HOME is set. You can check if this is already set by executing the following in a command prompt: ECHO %JAVA_HOME%. If it returns %JAVA_HOME%, then you know the value has not been set and you need to edit the following line in the file:

```
rem *****
rem * Set JAVA_HOME to Java SDK directory if not already set *
rem * For example: JAVA_HOME=c:\jdk1.3.1 *
rem * If JAVA_HOME is not already set, uncomment and set it here here *
rem *****
rem set JAVA_HOME=
```

Remove comment

Specify Java directory here

For example:

```
set JAVA_HOME=c:\jdk1.3.1
```

Set the classpath for the classes under test and test classes. It is set in the 'factory' to point to the JUnit and SilverMark examples provided:

```
rem *****
rem * Set libraries required for the test classes and classes under test *
rem * For example, to add the JUnit and SilverMark examples and tests: *
rem * This is already set to the delivered example classes and tests *
rem *****
set DIRS_AND_JARS=..\junit3.7;..\examples
```

For example, you might set is like this if your classes under test are in c:\development\myapplication.jar and your test classes are in cL\tests:

```
set DIRS_AND_JARS=c:\development\myapplication.jar;c:\tests
```

Setting up smejunit.properties

This file contains settings for the Enhanced JUnit test runner. The values are explained with comments within the file.

The only value that may need to be changed is that of the workbench path. This is the path to the directory where the Test Workbench is installed.

Note: If you installed the product in the c:\silvermark directory, you do not need to change this file

The line to change is shown here:

```
# Set the startup path for the Enhanced JUnit workbench so it may be
# automatically started by the the JUnit runner.
# This should be specified
# as the <install_dir>\bin\ directory (with directory separator on
# the end, as shown).
# Note: Escape characters must be double-escaped.
#   For example, c:\silvermark\smejunit\bin\ should read,
#       c:\\silvermark\\smejunit\\bin\\
workbenchPath=c:\\silvermark\\smejunit\\bin\\
```

Note: The only values in this file that are used by the load test execution agent are the logging values (*logDebugMessages*, *logMessages*, *logErrorMessage*s, *logWarningMessages*, *logStateMessages*).

Setting environment space for Windows 95, 98 and ME

Versions of Windows that are DOS based may fail during startup with the message, “**Out of environment space**”. There are two causes for this:

- A. **JAVA_HOME** is not set. If this is the case, set the value in setvars.bat.
- B. The default MS DOS environment space setting is too low. To remedy this, simply edit the properties for the **.bat** file you are trying to start (select file/pop up/Properties)

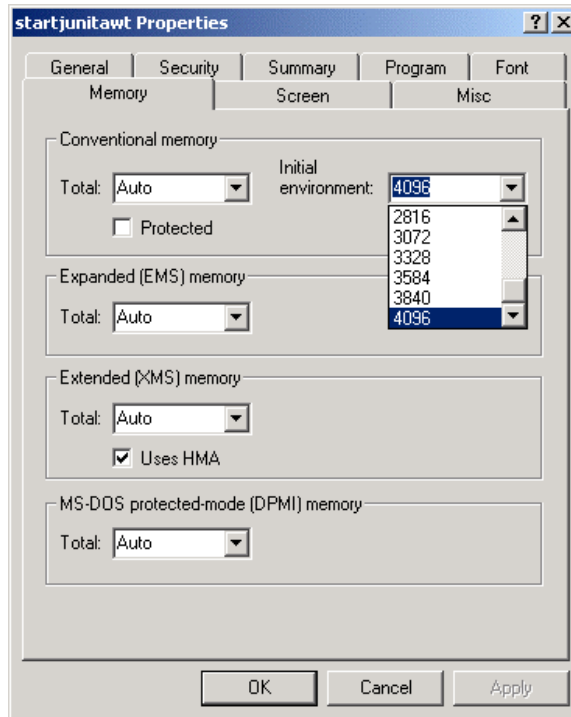


Figure 21 - Memory setting for MS DOS .bat file

Then set the Initial environment the largest value available. Remember to press OK or Apply.

Appendix C – JPDA setup

In order to use the code coverage and object interaction instrumentation feature of Enhanced JUnit, you will need to set up Sun's Java™ Platform Debugger Architecture (JPDA) add-in. JPDA is a feature of the Java™ 2 Standard Development Kit (SDK) Version 1.2.x or greater that provides debugging and instrumentation abilities to external tools.

You will need to separately download and install JPDA unless it is already installed on your system. **JPDA is included with Java 2 Platform Standard Edition (J2SE) SDK 1.3 and higher for Windows, so there is no need to separately download it.** To determine which Java version you are using, open a console window and type `<java_install_dir>\bin\java -version`. To download JPDA, please visit <http://java.sun.com/products/jpda/> and carefully follow the installation instructions found at this website.

Add JPDA to the JUnit Runner CLASSPATH

The `startjunitawt.bat` and `startunitswing.bat` files set and use the **JPDA** environment variable to point to a Java archive file containing the JPDA runtime support, with the following:

```

rem *****
rem * Java Platform debugger architecture (JPDA) setup for code coverage and *
rem * object interaction recording: *
rem * *
rem * For Sun J2SE SDK 1.3: *
rem * If you are using Sun J2SE SDK 1.3, the JPDA libraries come bundled *
rem * with the SDK in JAVA_HOME\lib\tools.jar *
rem * *
rem * For Sun J2SE SDK 1.2.x: *
rem * SDK 1.2.x users will need to separately download and install the JPDA. *
rem * To download, go to: *
rem * *
rem * http://java.sun.com/products/jpda/ *
rem * *
rem * After downloading and installing the JPDA, modify the environment *
rem * variable below to point to jpda.jar instead of tools.jar. *
rem * *
rem * The setting below assumes you are using J2SE SDK 1.3 *
rem *****
set JPDA=%JAVA_LIB%\tools.jar

```

JPDA environment variable setup section of startagent.bat

If the **JPDA** environment variable is set up correctly according to the instructions included above, it will be automatically added to the **CLASSPATH** for you. The **JAVA_LIB** environment variable is based on the **JAVA_HOME** environment variable, which is set earlier in the file.

Add JPDA to the system PATH

JPDA is implemented with several platform specific DLLs. You will need to make these accessible to your system. Review the following section of the `startjunitawt.bat` and `startjunitswing.bat` files to ensure the **PATH** environment variable correctly:

Note: The difference between **PATH** and **CLASSPATH** is that the **PATH** defines DLLs and other code accessible to the Windows system. The **CLASSPATH** defines Java classes that are available to the Java platform.

```

rem *****
rem * Append to your environment path so that the native JPDA libraries can be *
rem * dynamically loaded. The exact locations of these required .dll *
rem * libraries depends on which version of the Java SDK you are using. *
rem * *
rem * DLLs: jdwp.dll dt_socket.dll dt_shmem.dll *
rem * *
rem * Sun J2SE SDK 1.3: *
rem * The .dll libraries are found in the bin\ subdirectory of your *
rem * SDK 1.3 installation home (eg. d:\jdk1.3\bin). *
rem * *
rem * Sun J2SE SDK 1.2.x: *
rem * The .dll libraries are found in the bin\ subdirectory of your *
rem * JPDA installation home which you previously downloaded and installed *

```

```

rem *      separately (eg. d:\jpdal.0\bin)      *
rem *
rem * The setting below assumes you are using J2SE SDK 1.3      *
rem *****
set PATH=%PATH%;%JAVA_BIN%

```

JPDA path environment variable setup section of startagent.bat

Compatibility with IDEs and debuggers

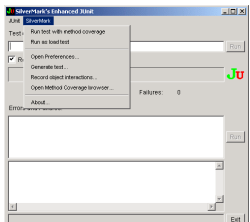
If you are using a development environment or debugging tool that uses the JPDA, you cannot use Enhanced JUnit's profiling feature at the same time you are running that debugger. This is because the JPDA framework has the restriction that it can only support one client at a time.

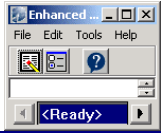


Because VisualAge for Java has its own instrumented virtual machine, it **does not support JPDA**. Therefore, if you are using Enhanced JUnit within the VisualAge for Java development environment, you will need to export your classes in order to gather code coverage and object interaction metrics.

Appendix D – Starting Enhanced JUnit

Enhanced JUnit consists of three separately running applications, all communicating with each other through socket connections.

Application	Purpose	To start
Test Runner	This is equivalent to the JUnit AWT or Swing UI that you are used to, with additional menu for SilverMark's enhancements. 	Run <install_dir>\smejunit\startjunitawt.bat or <install_dir>\smejunit\startjunitswing.bat

<p>Test Workbench</p>	<p>This provides test generation, and other services, as well as the user interface for object interaction recording, method coverage metrics, load test results and setting preferences.</p> 	<p>This is started automatically for you when you start the Test Runner if you have set the workbenchPath property in <install_dir>\smejunit\lib\smejunit.properties.</p> <p>You may also start it manually by running <install_dir>\smejunit\startworkbench.bat</p>
<p>Load test execution agent(s)</p>	<p>Each load test agent runs in its own Java Virtual Machine and runs one or more test threads as virtual users. You may run many of these on many machines</p>	<p>Run <install_dir>\smejunit\startloadtestagent.bat on the machine that it is to be run on, passing the socket to listen for requests.</p>

Examples

Start Swing UI runner

```
startjunitswing
```

Start Swing UI runner, specifying a test to run on startup

Start the Swing-based runner so that it immediately runs the sample Money test that comes with JUnit.

```
startjunitswing junit.samples.money.MoneyTest
```

Start AWT UI runner

```
startjunitawt
```

Start AWT UI runner, specifying a test to run on startup

Start the AWT-based runner so that it immediately runs the sample Money test that comes with JUnit.

```
startjunitawt junit.samples.money.MoneyTest
```

Start load test execution agent, using default port (3050)

```
startloadtestagent
```

Start load test execution agent, specifying required port to listen for requests on

Start the load test execution agent, listening for requests on port 4025.

```
startloadtestagent 4025
```

Appendix E – Adding custom load test ramp profiles

When you run a load test with JUnit/load, you specify the number of concurrent instances of the test case to start with the number to end with and an increment. The increment determines how many steps to take in ramping the number of concurrent tests between the *start* and *end* values.

By default, JUnit/load uses a simple algorithm to ramp the number of concurrent tests between the *start* and *end* values, in increments of the specified *increment* value. This profile for this is specified in

`com.silvermark.loadtest.SimpleLoadTestIncrementor`. You can easily install your own profile by following the following steps:

1. Create a class that implements the `com.silvermark.loadtest.LoadTestIncrementor` interface.
2. Specify that class in **smejunit.properties**

Here is the line in `smejunit.properties` that sets the load test ramp class, with the default value:

```
# Properties required for load testing  
  
loadTestRampClass=com.silvermark.loadtest.SimpleLoadTestIncrementor
```

The default class is:

```
package com.silvermark.loadtest;  
  
/**  
 * Simple strategy for ramping the number of concurrently  
 * executing tests from  
 * 'start' to 'end' by some 'increment' value.  
 * @author: SilverMark, Inc.  
 */  
public class SimpleLoadTestIncrementor implements LoadTestIncrementor {  
    private int count = 0;  
/**  
 * @copyright Copyright 1999-2001 SilverMark, Inc.  
 * @return int  
 */  
public int getCount() {
```

```

        return count;
    }
    /**
     * Given passed load test properties, calculate and return the next
     * number of tests to concurrently execute.
     * Return 0 if at the end.
     *
     * @copyright Copyright 1999-2001 SilverMark, Inc.
     * @return int The next number of tests to run. Return 0 if at end.
     * @param properties com.silvermark.loadtest.LoadTestProperties
     */
    public int increment(LoadTestProperties properties) {
        if (count == 0)    count = properties.getStart();
        else count += properties.getIncrement();

        if (count > properties.getEnd()) return 0;
        else return count;
    }
    /**
     * @copyright Copyright 1999-2001 SilverMark, Inc.
     * @param newCount int
     */
    public void setCount(int newCount) {
        count = newCount;
    }
    /**
     * Return the total number of test runs to perform.
     * That is, the total number of
     * times that some number of tests are to be executed on
     * load test agents.
     * @copyright Copyright 1999-2001 SilverMark, Inc.
     * @return int
     * @param properties Load test properties that specify start,
     * end and increment values.
     */
    public int testRuns(LoadTestProperties properties) {
        return 1 +
            (properties.getEnd() - properties.getStart()) /
            properties.getIncrement();
    }
}

```

Notice that it evenly increases the number of tests to execute between *start* and *end* by the *increment* value.

-END-