

SilverMark's Test Mentor – *Java Edition*

JUnit integration feature

If you use the JUnit framework and are interested in either migrating to Test Mentor or simply enhancing your JUnit experience, this document is for you.

If you follow these directions, you should be up and running your existing JUnit tests within Test Mentor in about fifteen minutes. For details about how to really get the most out of Test Mentor, you should read the Test Mentor User Guide.

Problem Statement

JUnit is a good tool, and the price is certainly right, however it does have its limitations:

- JUnit does not automatically generate tests of any great substance.
- QA team members (*testers*), who are typically charged with the task of creating automated tests, are unable to participate in early testing because their lack of programming skills prevent them from writing tests for components that don't have a user interface with which to interact.
- It is difficult to isolate and run a single test, which you might want to do in order to debug either the test or your code under test.
- JUnit does not provide fine-grained execution metrics, so subtle performance problems can easily creep into your code without your knowledge.
- JUnit does not have detailed execution logging, which can often be useful for debugging, or as an audit trail.
- JUnit's user interface is very minimal, and does not provide a way to manage your test assets.
- JUnit does not provide any mechanisms for iterating over test data files.
- Etc...

Of course Test Mentor solves these problems and many more. One of the key points of Test Mentor is that it serves as a *bridge between testers and developers*, so developers and testers can both collaborate on component testing, using whichever test representation that they are most comfortable with. On the other hand, there is a certain attraction to JUnit's simplicity because developers do like simplicity.

So the question is, how does one marry the simplicity of JUnit to the power of Test Mentor?

Test Mentor's approach

Dwight Deugo said in the December 2000 issue of the *Java Report* that a great product should not interfere with the way you like to work. With this in mind we decided that Test Mentor should augment developer's activities only at such when they need the services Test Mentor provides. Otherwise, Test Mentor should sit quietly in their back pocket. What this means is that once a developer who is used to using JUnit adopts Test Mentor, they can continue with their existing development practices, only starting up Test Mentor as the need arises.

Bringing JUnit tests into Test Mentor

The easiest way to explain how to bring existing JUnit tests into Test Mentor is with an example. Since JUnit comes with a number of examples, we might as well use them.

We assume you've already installed Test Mentor. The discussion below assumes that you've set properly up Test Mentor's optional connection to the Java Platform Debugger Architecture (JPDA) for instrumentation. It would also be nice if you've read some of the introductory material in the Test Mentor User Guide to gain a rough familiarity with the bits and pieces of it.

Recompile your JUnit tests

Test Mentor uses a modified version of the JUnit framework in order to integrate with JUnit. For this example, you will have to recompile JUnit's example test classes with the Test Mentor version of **junit.jar**, which is shipped in **<installdir>\silvermark\junit**. This will bind the test classes to Test Mentor's version of the framework, specifically, `junit.framework.Assert`, which provides Test Mentor-enabled versions of the **assert(...)** methods.

Start Test Mentor



Figure 1 - Windows start menu items for Test Mentor

Use the Windows start menu shortcuts created for you by the installation program to start the server, and then the client. As you might recall from the

manual, the server contains anything having to do with Java execution and reflection. The client handles the user interface and test generation logic.

When you start the client, you should see the following window:

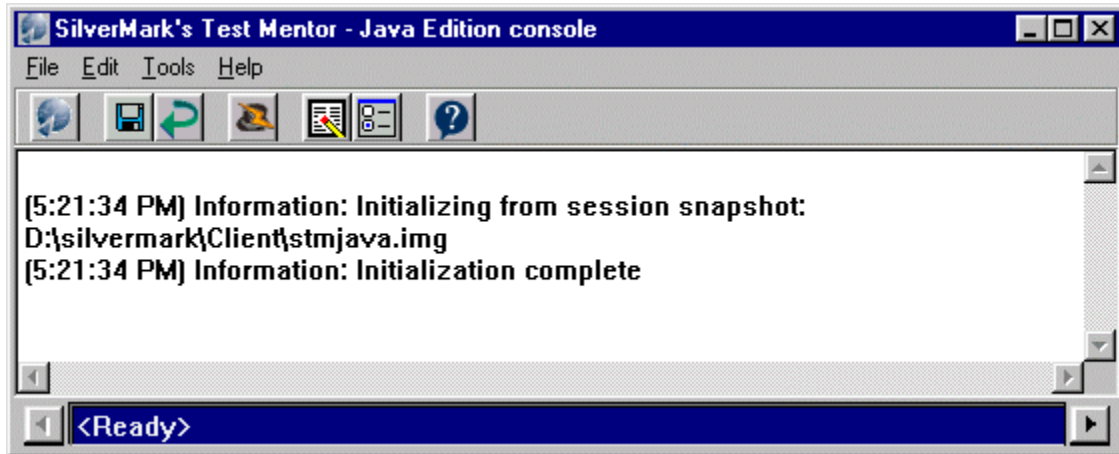



Figure 2 - Test Mentor console

Set the CLASSPATH

You're not really doing Java development unless you're tearing your hair out over CLASSPATH settings. Fortunately, Test Mentor makes setting your CLASSPATH easy.

In the console above, press the Preferences  button. This opens the Preferences view. Select the Paths setting:

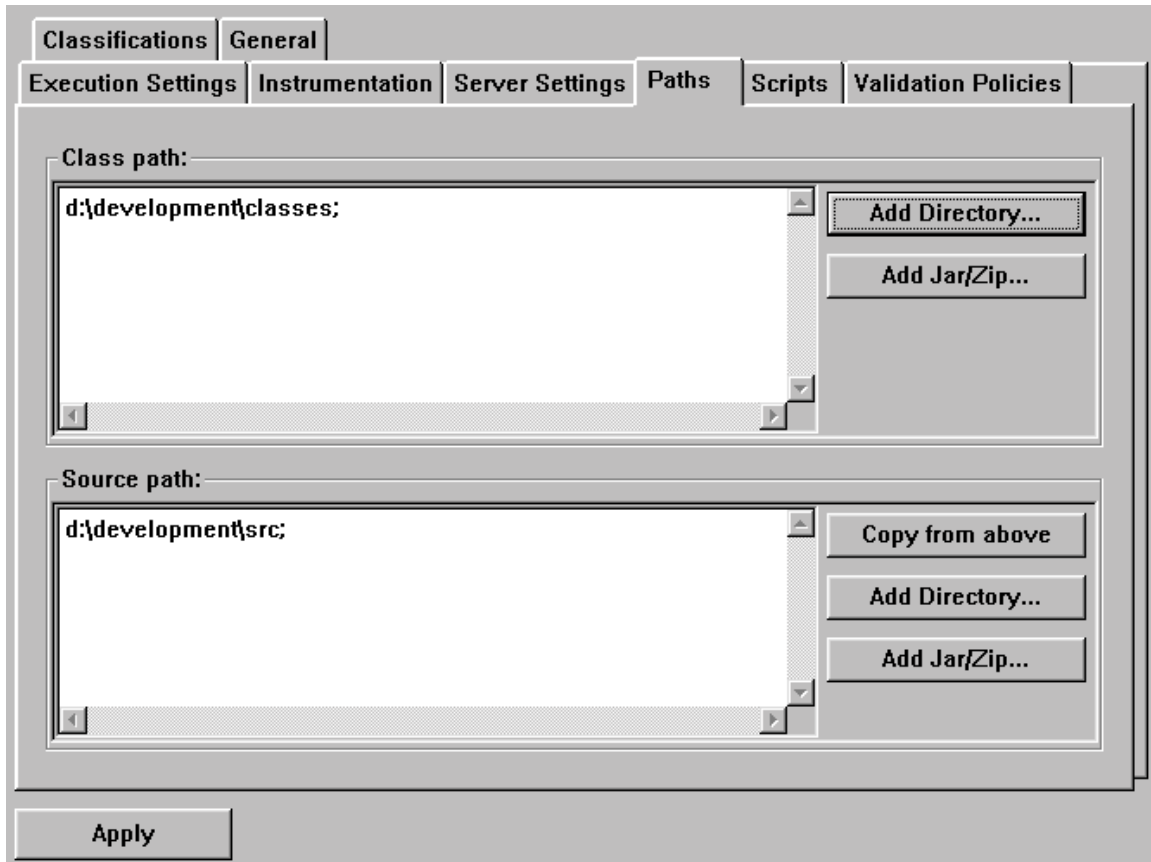


Figure 3 - Preferences 'paths' view

If you have not set anything in this before, it is probably empty. The area on the top lists CLASSPATH entries used for locating test classes and classes under test. For this example, you will set this to include your JUnit test classes. You will also need to include the JUnit framework itself here.

So, if you installed JUnit into **d:\junit**, and installed Test Mentor into **d:\silvermark**, you would set the Class path field to the following in order to give access to both the JUnit examples and SilverMark's version of the JUnit framework:

```
D:\junit;  
D:\silvermark\junit\junit.jar;
```


You must use SilverMark's version of JUnit for all this to work. You would then set the Source path field to:


```
D:\junit;
```

Since the source directory is the same as the class directory, you could have left the source field blank and it would have used the class directory by default.

Don't forget to press **Apply!**

Refresh your session

The Test Mentor client keeps a model of your test classes. This model is what is shown in the Test Workbench. In order to refresh this model with the JUnit tests, press the Refresh  button in the console. Doing so will cause Test Mentor to scan all the classes in the current CLASSPATH for any Test Mentor or JUnit test classes. You will be warned that scanning and building a model is a lengthy operation.

On completion, press the Test Workbench  button in the console. If you set up the above correctly to give Test Mentor access to both the JUnit examples, and Test Mentor's version of JUnit, you should see the following:

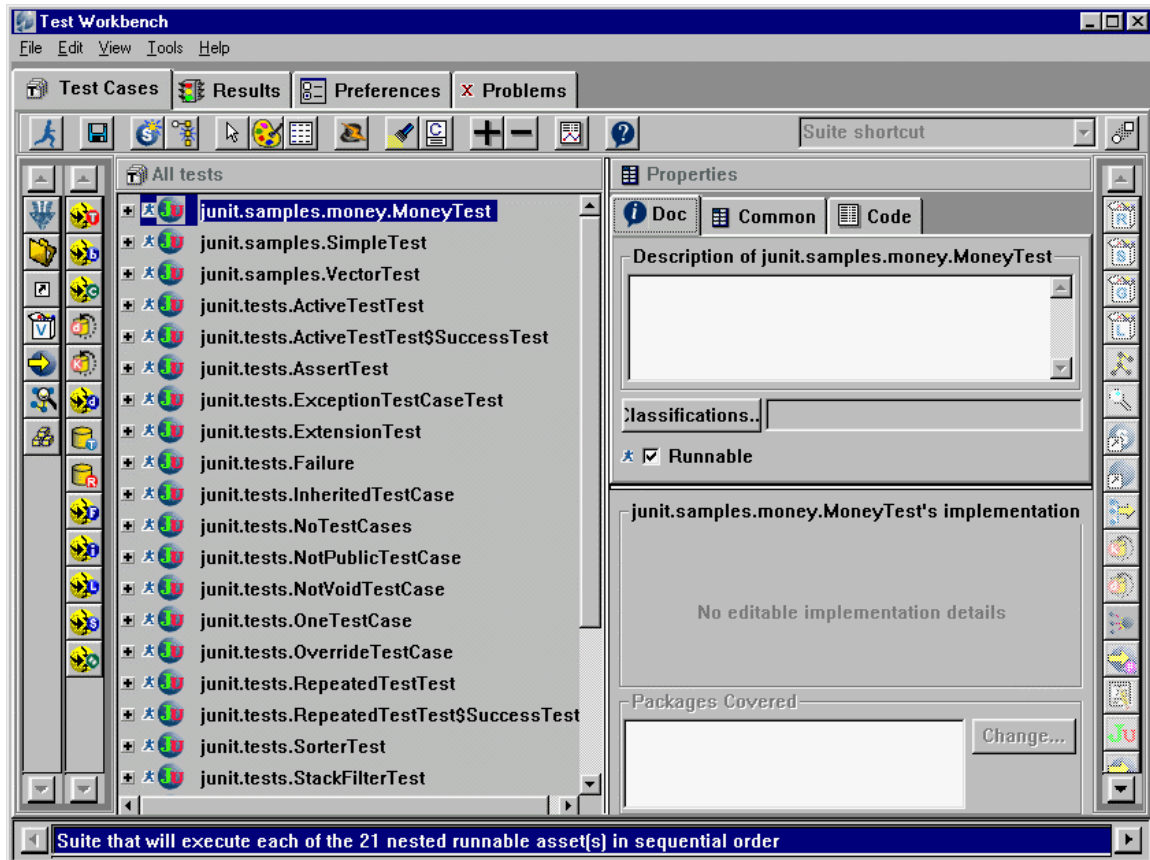


Figure 4 - Test Workbench after loading JUnit examples

Each item in the list shows a JUnit class. If you expand one of the items, for example, `junit.samples.money.MoneyTest`, you will see the individual tests contained within the class:

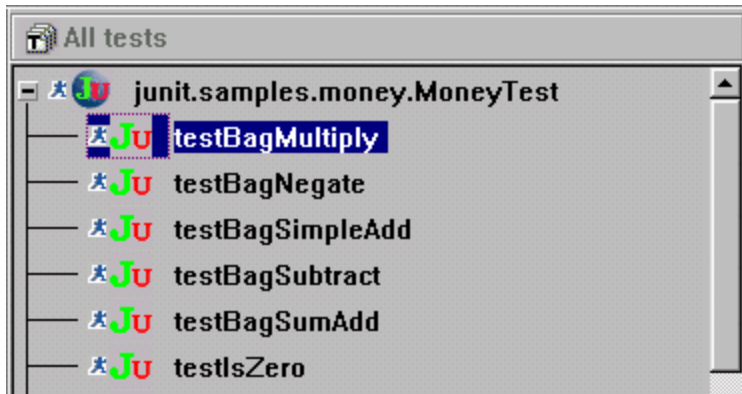


Figure 5 - List of tests within junit.samples.money.MoneyTest

Optional – enable instrumentation

The following introduces you to Test Mentor's optional instrumentation feature, which relies on the JPDA. If you run with instrumentation turned on, you'll be able to view metrics on method coverage, and object interactions. The disadvantage to running with instrumentation turned on is that tests run much more slowly than they would normally.

To enable instrumentation, navigate to the [Instrumentation](#) page in the Preferences view, which is accessible from the [Preferences](#) tab of the Test Workbench:

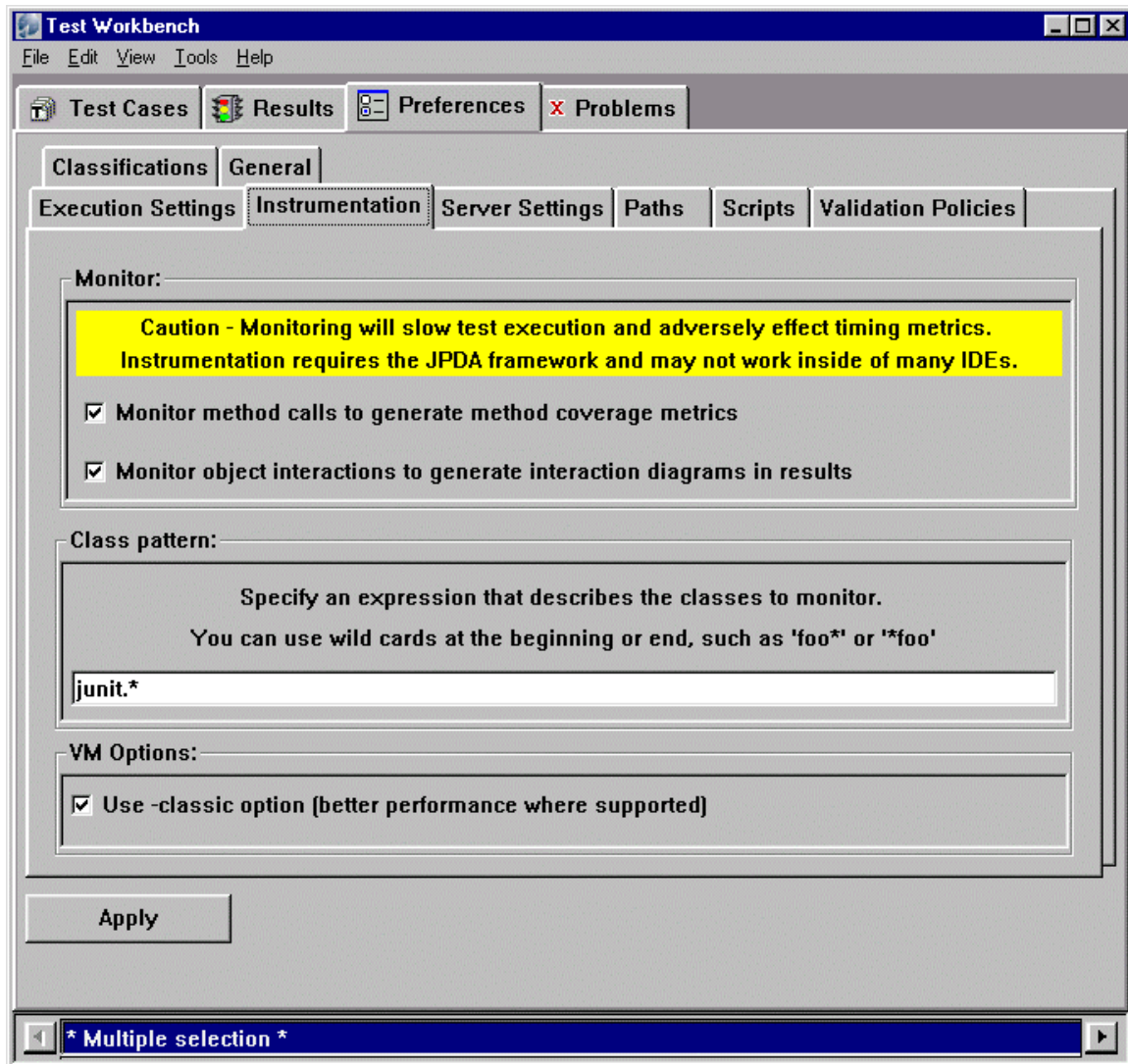


Figure 6 - Preferences set for JUnit class instrumentation


In the above, you can see that instrumentation for method calls and object interactions has been enabled.

In addition, the expression, **junit.*** has been specified. This expression is used to determine which objects to instrument. In this case, we'll get all the JUnit examples, plus the JUnit framework itself.

Finally, the using the `-classic` VM option often speeds things up when instrumentation is turned on. Not all VMs support it, but if yours does, you should definitely use it.

Running JUnit tests

You can select any one or more items in the list to run them. For example, if you are having problems with

`junit.samples.money.MoneyTest.testBagSubtract()` you can simply select and run it. To run selected items, press the Run  button in the upper left-hand corner.

If you run all the JUnit tests at once, by multi-selecting them and pressing the Run button, you should see the following results in the Results tab of the Test Workbench:

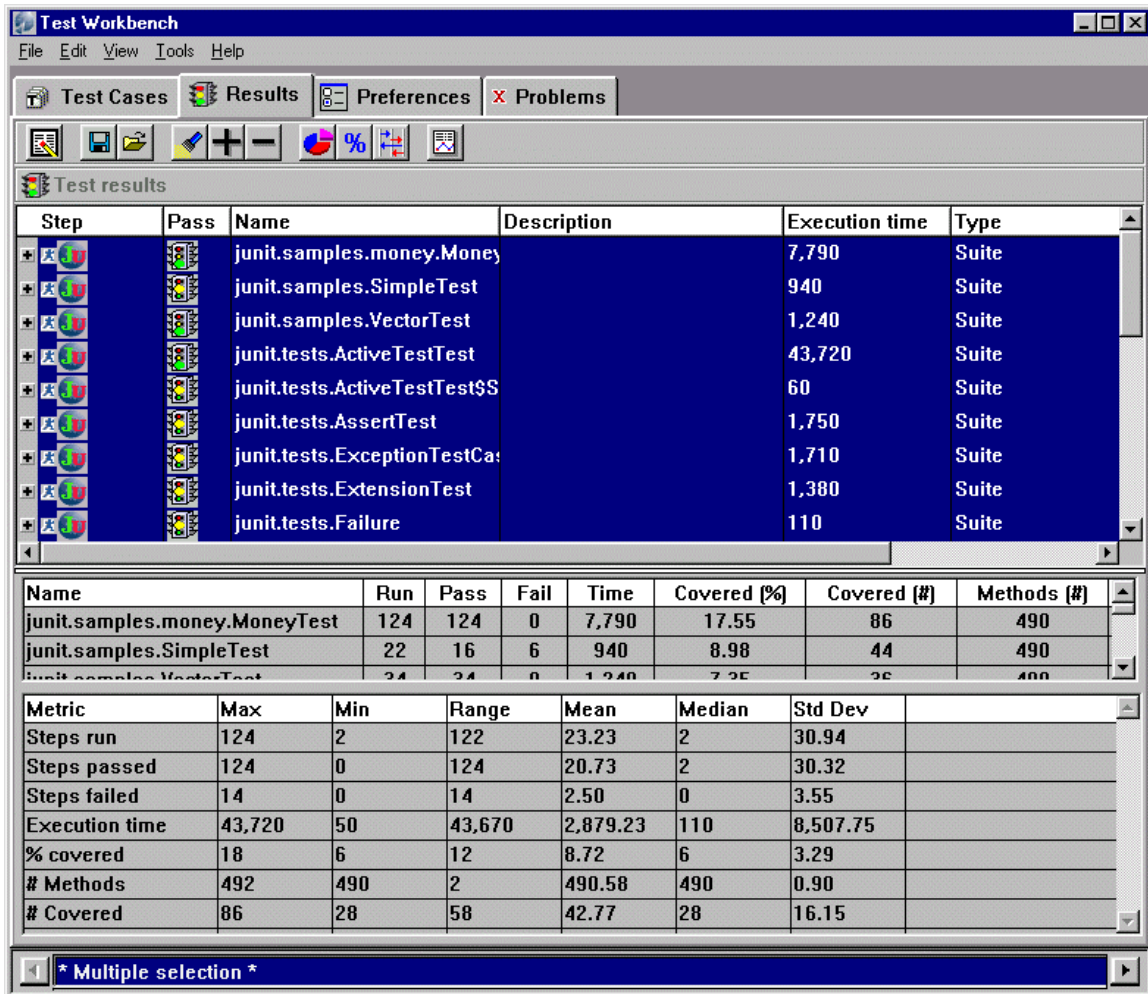





Figure 7 - Results (with all items selected) after running all JUnit example tests

As you can see, the structure of the test results closely follows the structure of the tests themselves. The green traffic lights  indicate tests that have passed. Yellow traffic lights  indicate either assert failures, or a test class that contains a test that had a failure. The red traffic lights  indicate unhandled exceptions, such as:

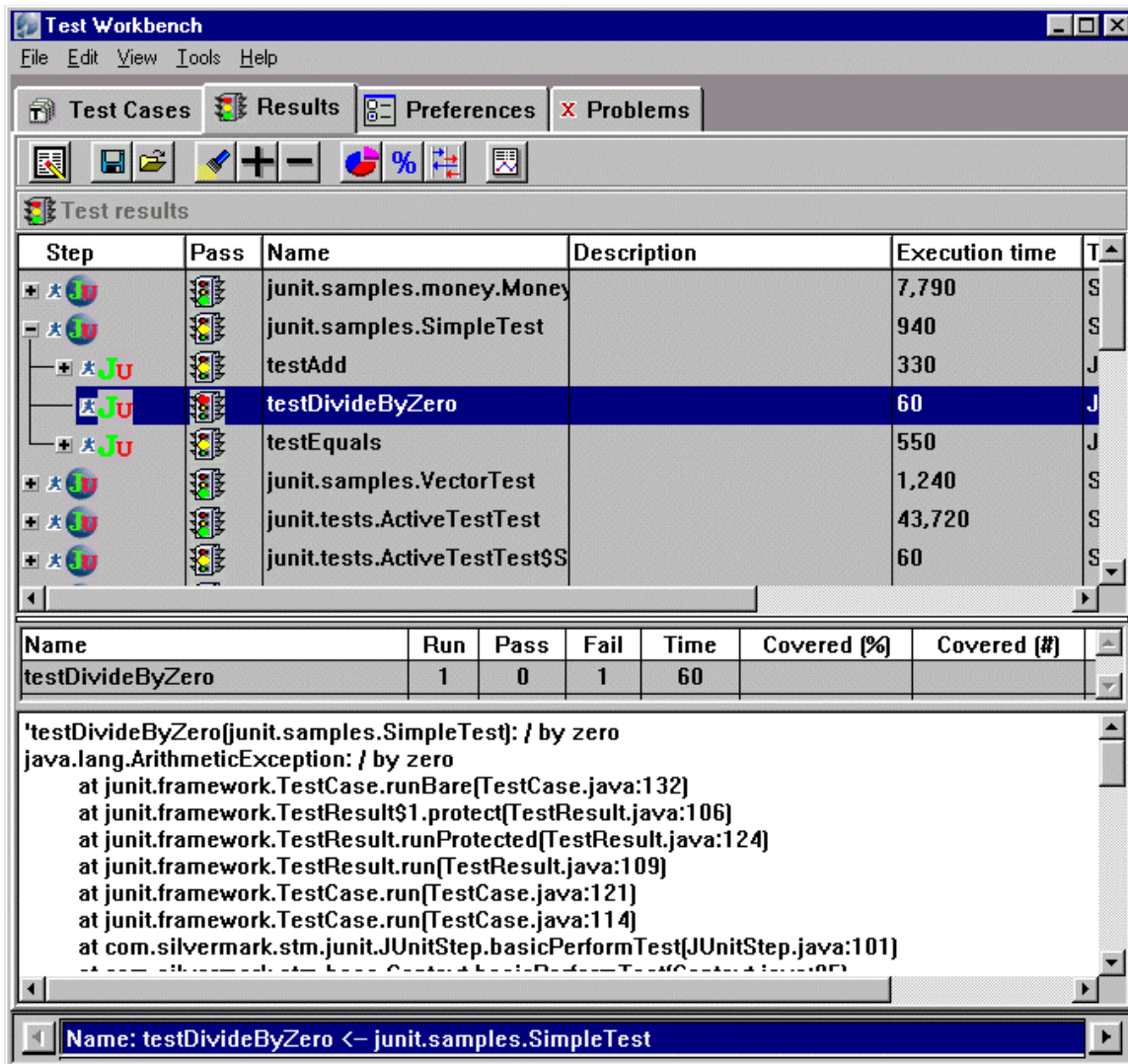



Figure 8 - Unhandled exception shown in results

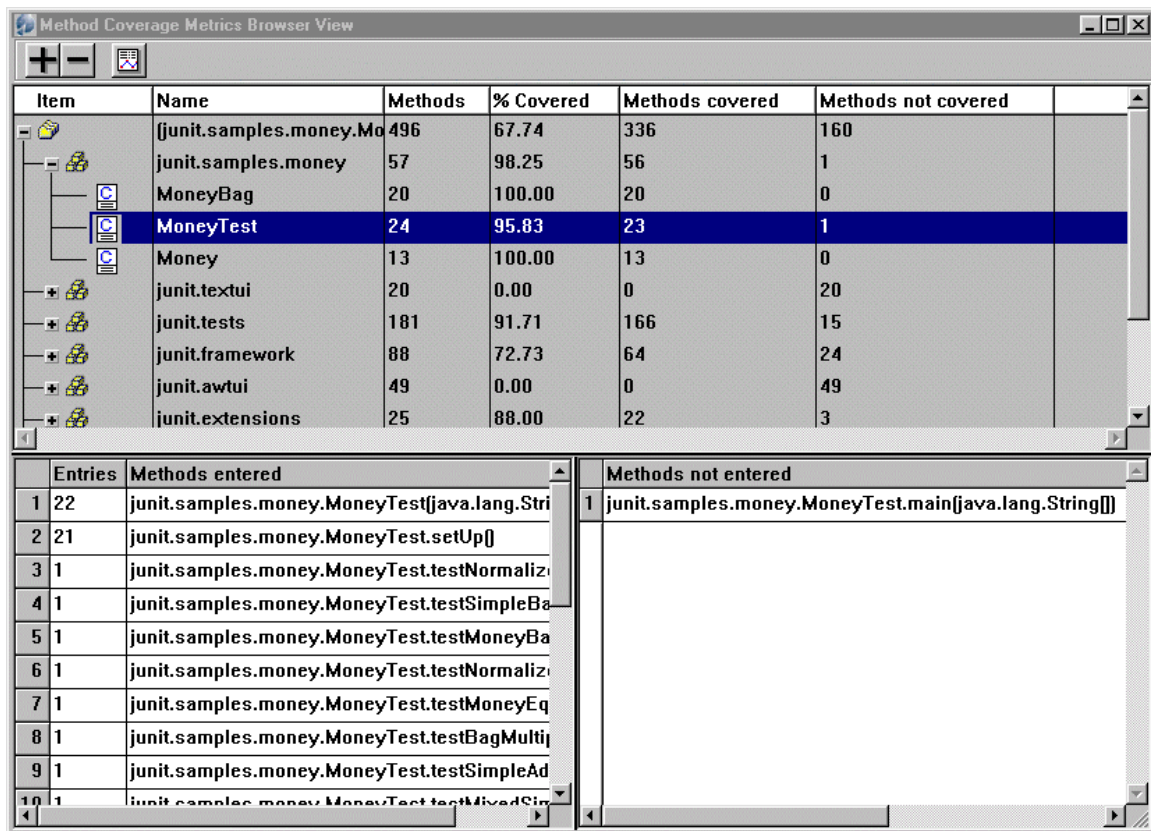
Note: You will see some failures show up for some example tests, that normally do not fail when run under the general version of JUnit. This is generally in tests for internal JUnit framework APIs provided in the examples that do not apply to the Test Mentor version. You can safely ignore these.

If you see some failures that indicate that is unable to locate one of the `assert (...)` methods, you probably didn't recompile the example tests with the Test Mentor version of `junit.jar`.

Method coverage metrics

Notice in the **Covered (%)**, **Covered (#)** and **Methods (#)** metrics in the results. These indicate which methods were covered by the given test. If you set up the JPDA connection properly and enabled instrumentation, you should see numbers there for each of the JUnit test classes.

If you select all the items and press the  button in the Results view, you should see the following:




Item	Name	Methods	% Covered	Methods covered	Methods not covered
	(junit.samples.money.Mo	496	67.74	336	160
	junit.samples.money	57	98.25	56	1
	MoneyBag	20	100.00	20	0
	MoneyTest	24	95.83	23	1
	Money	13	100.00	13	0
	junit.textui	20	0.00	0	20
	junit.tests	181	91.71	166	15
	junit.framework	88	72.73	64	24
	junit.awtui	49	0.00	0	49
	junit.extensions	25	88.00	22	3

Entries	Methods entered	Methods not entered
1 22	junit.samples.money.MoneyTest(java.lang.Stri	1 junit.samples.money.MoneyTest.main(java.lang.String[])
2 21	junit.samples.money.MoneyTest.setUp()	
3 1	junit.samples.money.MoneyTest.testNormaliz	
4 1	junit.samples.money.MoneyTest.testSimpleBa	
5 1	junit.samples.money.MoneyTest.testMoneyBa	
6 1	junit.samples.money.MoneyTest.testNormaliz	
7 1	junit.samples.money.MoneyTest.testMoneyEq	
8 1	junit.samples.money.MoneyTest.testBagMulti	
9 1	junit.samples.money.MoneyTest.testSimpleAd	
10 1	junit.samples.money.MoneyTest.testMixedSim	

Figure 9 - Method coverage view for JUnit example tests

This view shows coverage metrics organized by test, package and class. You should be able to use this to see which methods still need test cases. For example, the above shows that `junit.samples.money.MoneyTest.main(String[])` is not executed by these sample tests.

Object interactions

Go back to the Test Workbench and press the Interaction Analysis  button. You should see a view like the following:

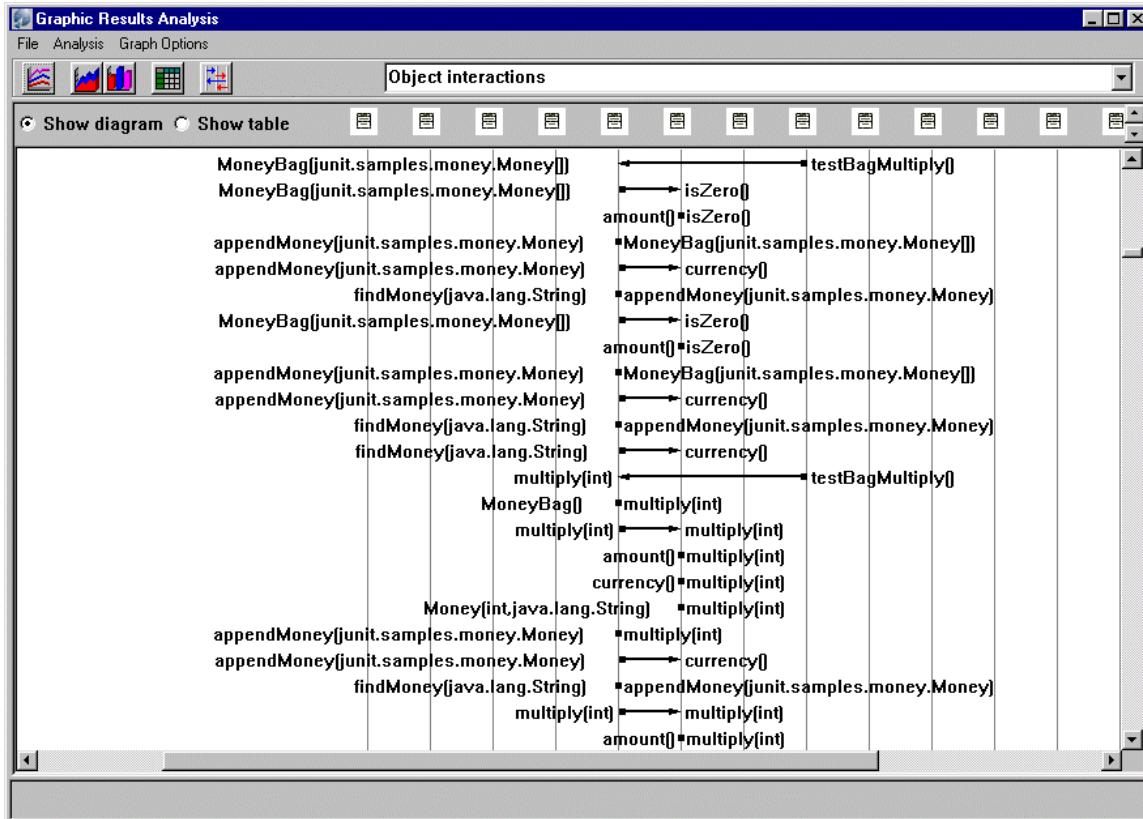


Figure 10 - Object interaction diagram for JUnit examples

This view gives you an overview of the objects that interacted as a result of executing the tests. This can be very useful for determining the relationships between objects, especially with code that is more complex than the JUnit samples.

There are other graphical views and features of Test Mentor that you can read about in the Test Mentor User Guide.

Combining Test Mentor tests with JUnit tests

Test Mentor provides facilities for automatically generating and editing test assets. You can read more about those in Test Mentor User Guide.

Test Mentor tests can be installed in and peacefully coexist with JUnit test classes.

You can use Test Mentor to add Test Mentor test assets to any JUnit class (subclass of `junit.framework.TestCase`). It will create methods for those assets without disturbing your JUnit test methods unless you create a test asset with the same name as one of your test methods.

Sharing tests with QA team members

Because Test Mentor represents tests visually and provides automatic test generation facilities, QA team members (testers) can write unit tests without knowing how to program in Java. This means that testers can pitch in and help with early test creation, and the tests they create can coexist with your JUnit-style tests. Likewise, testers can use Test Mentor to run your JUnit-style tests with Test Mentor's user interface.

This means that testers can get involved with testing early in the cycle, instead of waiting until a user interface is available for them to test with. Developers and testers can collaborate on creating and executing unit tests right from the beginning.

Conclusion

We hope this was helpful. You should now have a good idea how to use Test Mentor to run your JUnit tests and understand how you can create Test Mentor test assets within JUnit test classes. You should also have a feel for how developers and testers can collaborate on testing right from the beginning.

If it sounds like Test Mentor might make your team's job of delivering high-quality code with minimal personal stress a reality, feel free to download an evaluation version of Test Mentor from the SilverMark web site at www.silvermark.com, or contact info@silvermark.com or U.S. 888-588-0668 to speak with an engineer.