

The fifteen-minute overview to SilverMark's Test Mentor – *Java Edition*

Ok, you've heard about Test Mentor – *Java Edition* and you're wondering what it's about, but you don't have much time to spend reading about tools. Here's a brief introduction that should give you a pretty good feel for the who, what, why, and how of the tool. If you're still interested, then maybe you could spend an hour reading the user manual or even give the evaluation version of the product a test-drive. You can also contact SilverMark and talk to one of their engineers. They're very nice people and really like to talk about testing – just ask their families.

Problem Statement

Correctly operating (defect free) code is good. Even the best programmers create defects. It is less costly to catch defects earlier than later. You locate defects in code by testing it the way it is intended to be used. The earlier you begin testing, the earlier you remove defects. Writing tests for code without the right tools is hard, and most developers would rather be writing code than tests anyway.

These preceding statements should not be controversial, and you probably already agree or you wouldn't be reading this right now.

Now add this to the mix: Although developers like the idea of writing code and then pressing a button to find out whether that code will cause ships to sink and planes to fall from the sky, they don't often want to spend the time creating tests. At the same time, 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.

This is unfortunate because testers have the mindset and the mission to create automated tests, but they have to wait until late in the cycle to really flex their testing muscles.

The vision behind Test Mentor – *Java Edition* is to make it easier for your team to test code early. Test Mentor makes it possible for developers and testers to collaborate as a team to make it easy to test early and often. What follows is an overview of how the tool accomplishes this.

Test Mentor's approach

Test Mentor – Java Edition is a tool that automates testing of components that are written in the Java programming language. Furthermore, Test Mentor 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.

In our view of the world, a component is a single Java class, cluster of collaborating classes, or even an entire framework. In many cases, these are objects that implement business rules according to developers' interpretation of the product requirements. We do not include graphical user interfaces (GUIs) in this definition.

When we say testing, we mean using components in the same way that they are intended to be used. You can call this service-based or functional testing for components. A component that is tested in this way should offer very little in the way of surprises once it is integrated with the rest of an application.

In other words, if you do a good job testing the services that a component offers you should be pretty confident that you've flushed out any logic errors, and that its business rules are implemented correctly. If you test this way with a reasonably representative set of input data that includes boundary values, you can be pretty confident that there are no unhandled exceptions waiting to pounce on your customers with bared teeth once your components are deployed.

Automated test creation

Test Mentor automates test creation by generating complete or nearly complete tests from available information. If you have design models, such as those you might create with Rational Rose, it will generate tests from them. Design models can be very useful if they have sequence diagrams or state transition diagrams that describe the dynamic behavior of your components. Test Mentor will also generate tests by traversing static class structure and relationships. Test Mentor generates tests structured according to common test design patterns.

Test representation

The tests you create (or that Test Mentor creates for you) can be represented either as Java code, or as visual components that you define and edit within Test Mentor. You can pick and choose your representation, depending on your own preference, and you can mix the two representations together within Test Mentor.

- Besides looking pretty, the visual definition scheme enables you to create tests without knowing how to program in Java. This is key to enabling non-developers to create tests.

- Developers typically prefer to work directly in code. Test Mentor, with the change of a setting, generate tests as plain Java code instead of visually.

Terminology

Test Mentor introduces three terms:

- Test Suite
- Test Asset
- Test Step

Test Suite

Test suites organize items related to a particular area under test. You might use a suite to hold all the test items that you've created for testing a particular class, or a particular feature set of related services offered by a framework.

Test suites are physically represented by Java classes, although if you are not a Java programmer, much of the implementation is hidden for you.

Test Asset

A test asset is what we call a reusable test component. We would have used the term test component, but it seemed that the word "component" was already pretty overused, and the term, "asset", reinforced our view that the tests you create should be viewed as software assets that increase your organization's value.

Above, when we referred to "items" within a suite, what we really meant was test assets within a suite, because that is what test suites hold – test assets. You can reference a test asset from within any test, which means you can use them as building blocks to build large tests from small pieces. This is a very important feature that when used properly, leads to very well structured, maintainable and scalable tests.

Test assets are physically represented as methods within suite classes. Again, if you are not a Java programmer, much of the implementation of test assets is hidden for you.

Test Step

The test step is the most granular unit of execution within Test Mentor. Test assets are composed of test steps. Test Mentor offers a number of different types of test step, each one designed to provide a specific type of testing behavior or functionality.

Test Mentor provides a palette of test step types that, to name just a few, execute constructors and methods, read test data out of files, call EJBs, extract and validate state, and so on.

Test steps are configured in terms of properties that they need to know in order to act on your components, as well as common properties such as preconditions, postconditions, expected exceptions, iterations, etc. Most of the properties of steps are optional, so you don't need to learn them all at once in order to become productive right away.

Test steps are physically represented as statements within asset methods. As always, if you are not a Java programmer, much of the implementation of test steps is hidden for you.

Terminology summary

To summarize, suites contain assets, which contain steps. You can reuse assets by referencing them from within tests, and you can pick from a palette of step types that you may or may not further configure with all sorts of interesting properties.

A very fast product tour

This section takes you on a very fast tour of the product so you'll have a rough idea of what Test Mentor offers.

Test Workbench

The test workbench is where all the action is:

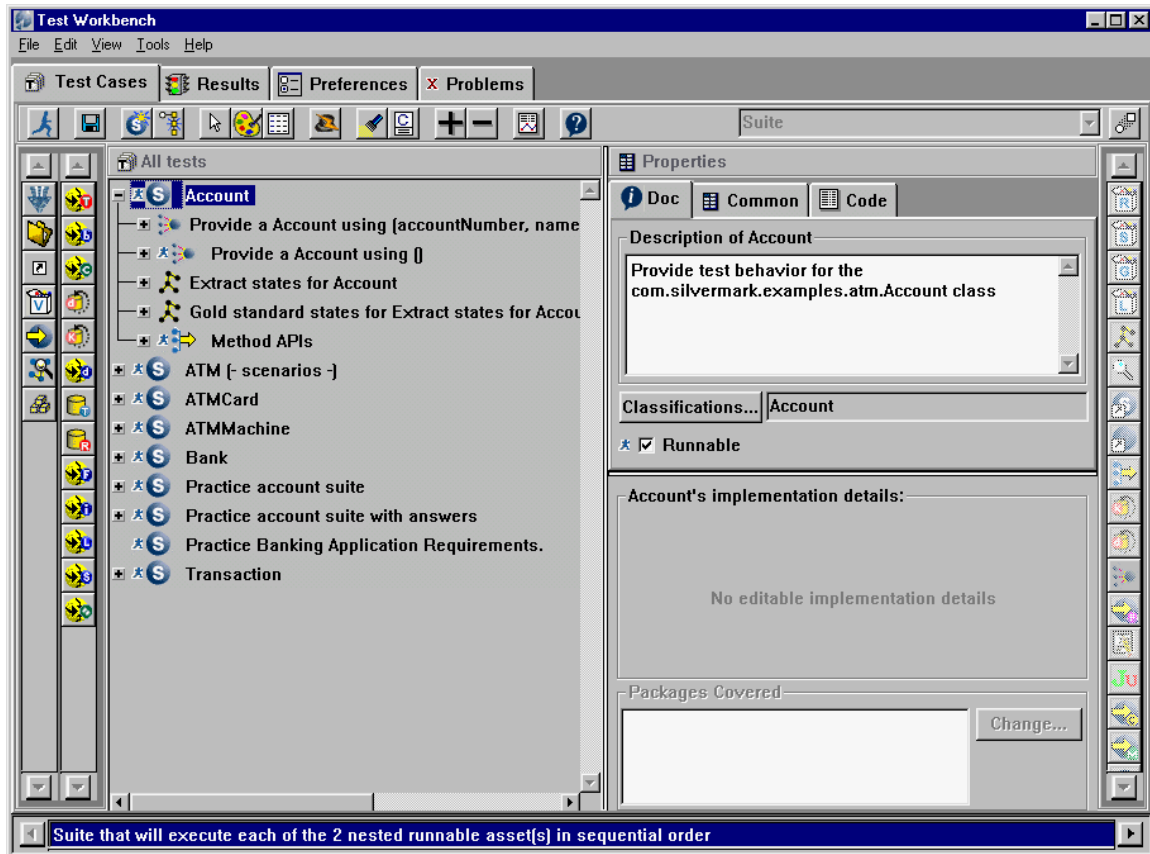


Figure 1 - Test Workbench


The Test Workbench provides all the editing facilities for visual test construction, and provides a launching point for the automated test generation wizards, test results analysis, and configuration tools.

The Test Workbench is divided into four views, accessible via notebook tabs:

- **Test cases** – used to manage test suites, assets and steps
- **Results** – used to show the results of test execution
- **Preferences** – used to configure the tool
- **Problems** – used to show you which steps need further configuration

There's a lot in the Test Workbench; much more that will fit in a fifteen-minute overview, so the following sections will only hit some of the high points.

Test cases view

The above screen capture shows the test cases view. Each  icon represents a suite. The top suite (for testing the Account class) is selected. You can see that

this suite contains five test assets. The assets are created from steps, whose types are indicated by specific icons. The Account suite shown is an example of visually constructed test assets. As it happens, this particular test suite as shown was entirely generated by one of Test Mentor's automatic test generation wizards.

On the left side of the view is a palette of buttons for icons for types of steps. You can select and drag steps from this palette on to the list, or you can use one of the wizards to automatically generate tests or guide you through test creation. You'll typically use the step palette for fine tuning your visually created test assets.

Results view

The results view shows the results of executing a test:

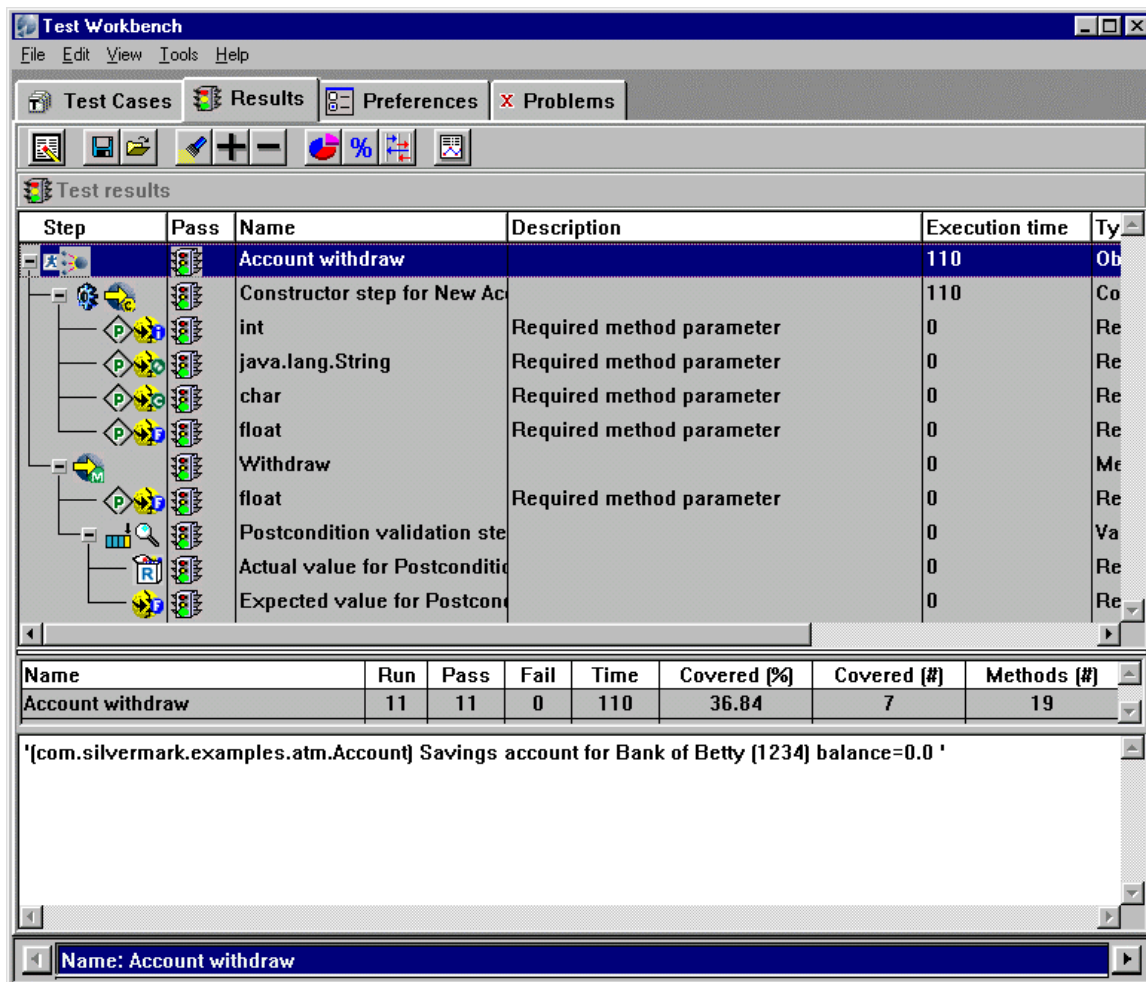
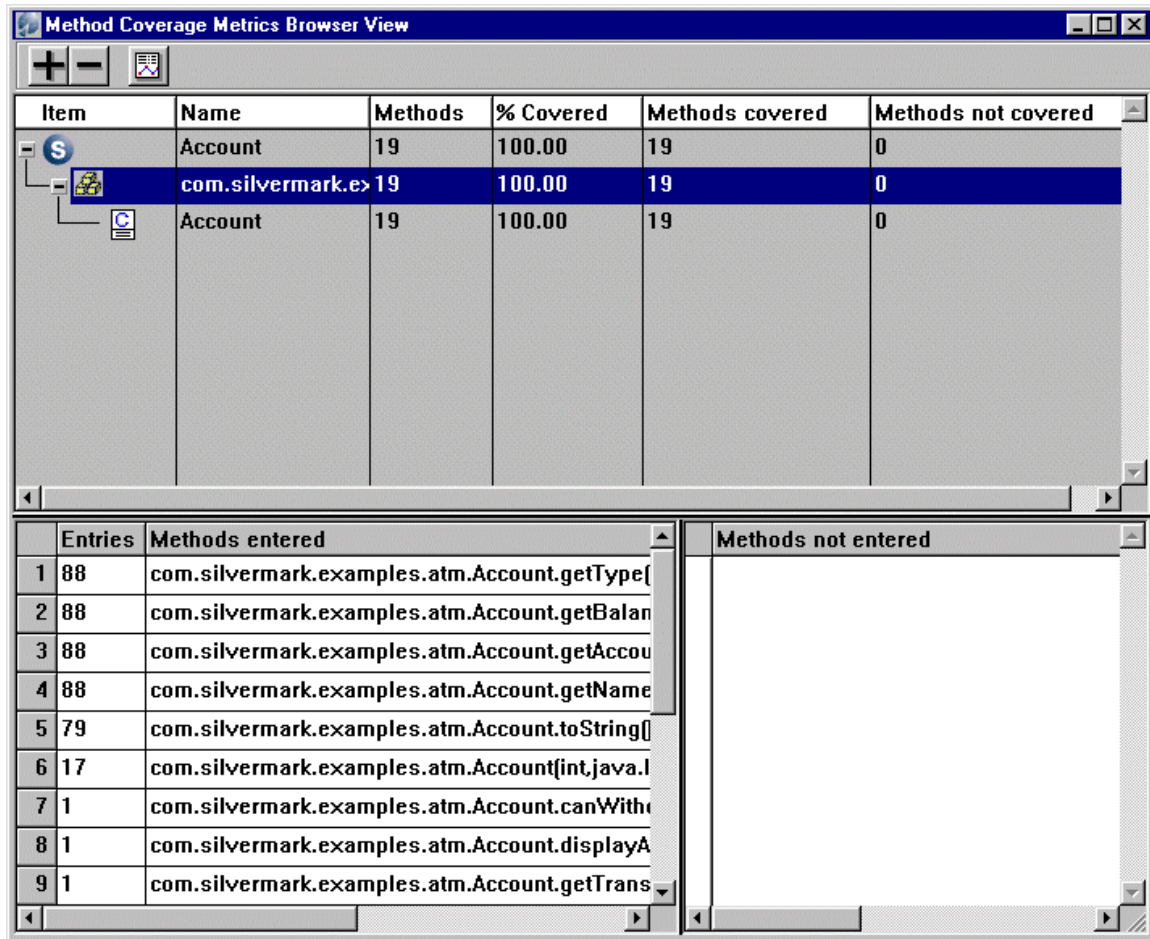


Figure 2 - Test results view

The test results view above shows the results of individually executed steps (if you ask it to), including values passed as parameters, execution time and other execution metrics, such as method coverage summary.

Method coverage

You can launch a method coverage metrics browser to show details of which methods were covered by a test:



The screenshot shows a window titled "Method Coverage Metrics Browser View". It contains a tree view on the left and a table of entries on the right. The tree view shows a hierarchy of items, with "com.silvermark.e" selected. The table below shows the details of the methods covered and not covered.

Item	Name	Methods	% Covered	Methods covered	Methods not covered
S	Account	19	100.00	19	0
com.silvermark.e	Account	19	100.00	19	0

Entries	Methods entered	Methods not entered
1	88	com.silvermark.examples.atm.Account.getType[
2	88	com.silvermark.examples.atm.Account.getBalanc
3	88	com.silvermark.examples.atm.Account.getAccou
4	88	com.silvermark.examples.atm.Account.getName
5	79	com.silvermark.examples.atm.Account.toString[
6	17	com.silvermark.examples.atm.Account(int,java.l
7	1	com.silvermark.examples.atm.Account.canWithd
8	1	com.silvermark.examples.atm.Account.displayA
9	1	com.silvermark.examples.atm.Account.getTrans

This gives you a clear idea, organized by step, package and class, of which methods were covered by the test and which were not. We recommend you use this to guide you in determining where to focus your efforts in creating tests.

Graphical results views

You can export Test Mentor's results (they are in XML format) for your own purposes or use some of Test Mentor's pre-digested charts to visualize the health of your code. Lastly, Test Mentor will capture interactions between objects during the execution of a step and render them for you as an object interaction

diagram. This can be useful if you are trying to validate that objects are interacting as you would expect, and can even be used as a debugging tool.

Test generation wizards

The test generation wizards take input from either design models or static class information, via reflection, and generate tests according to some set test implementation patterns. As we have seen above, the test generation wizards will generate tests as visual components, or as Java code shown below:

```
/**
 * Tests the <validWithdraw> UML Sequence Diagram.
 * Test an ATM cash withdrawal transaction
 */
public java.lang.Object testValidAccountWithdrawal() {
    com.silvermark.examples.atm.ATMMachine subject = null;

    // Create test subject from asset
    subject = (com.silvermark.examples.atm.ATMMachine) executeAsset(
        "com.silvermark.examples.atm.tests.ATMMachine"
        "provideAStandardATMMachine");

    // Insert an ATM card
    subject.insertCardNumber(123);

    // Set password
    subject.insertPassword("please");

    // Set account type. The user has two types of accounts,
    // a checking account and a saving account.
    // Here the user selects the account type
    // by passing 'c' for checking and 's' for saving
    account.subject.insertAccountType('c');

    // Withdraw cash
    subject.withdraw(100.0f);

    assert("Verify results", subject.getCashDispensed(), 100.0f );

    return subject;
}
```

The above code was generated directly by Test Mentor from a design model.

Test Mentor in the development process

Before wrapping up, let's discuss where Test Mentor fits within the development process.

On a typical development project, developers write code, and they may or may not create and execute unit tests. If the project has a QA team, testers may get involved at some point later on when there is a user interface with which to test.

If the developers do a good job with their unit tests, the testers can concentrate on isolating those last bugs, integration problems and errors of omission that were missed by the unit tests.

If the developers do not do a good job unit testing, the testers may find themselves finding simple problems that should have been caught earlier. Now the testers are frustrated because they keep getting 'broken' builds, which keep them from really digging in and testing the product as a customer might use it. In addition, the developers end up fixing bugs later in the cycle which destabilizes the product at the worst possible time.

The problem with the above is that the testers are wholly reliant on the developers doing a good job unit testing at the same time they are writing code, because they are unable to really get involved until there is a user interface, because testers typically don't have the skills to write Java unit test code.

When you add Test Mentor to the mix, developers and testers can collaborate early in the process. Enabling testers to test Java components via Test Mentor's visual representation relieves developers from the burden of developing unit tests all by themselves.

With Test Mentor, developers sit down with testers to plan how best to test their components. If a developer uses design models, he or she may include testing artifacts in the design that testers can use as cues for their test creation.

Test Mentor's test suites support a mix of code and visual representations. This means that developers and testers can both create, combine, and share test assets.

As a summary, here's a dialog between developer Pat and tester, Chris where they plan out how they are going to collaborate on a new class:

Pat: Chris, I'm about to start implementing the new PolicyRater class and could use some tests.

Chris: Sounds good. What have you got for me?

Pat: We have some design models that outline the public interface of the class, and some sequence diagrams that show typical interactions.

Chris: Cool! I'll just run Test Mentor on those to start off.

Pat: Ok, but I think there's some more detailed interactions with the class that aren't documented, but should really be turned into tests.

Chris: Ok, just let me know what they are.

< a few minutes at the white board later >

Chris: I think I've got it. I'll go create and document the suite, and you can review it.

Pat: Sounds good. You had some good ideas for structuring the tests. By the time you're done with the implementation for the suite, I should be ready to run it on my class. Also, I'll add that verification we talked about, because it's a little bit complicated and it would be best to do that in code.

Chris: That would be great. Just let me know when and I'll make sure it gets added to our big body of tests. That way, they'll get run automatically with every build.

Pat: Yeah, but I'll make sure I run them on my own machine before I release any code. I don't need all those dirty looks like I got the last time I forgot to run them, and broke the build.

Where is Test Mentor going?

Our focus, as we carry Test Mentor forward, is on more automated test creation, integration with additional modeling tools, development environments, application servers, and support for additional Java implementation and deployment architectures.

If you have a favorite development tool or are using a specific Java deployment architecture or application server please let us know. We are always open to input.

Conclusion

We hope this was helpful. Perhaps it raised more questions than answers. That's ok. 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.