



End to End Process Testing & Validation:

A real-world example for managing the quality of
integrated business systems



Testing and validating today's heterogeneous and distributed systems is challenging. Such systems are complex, pinpointing the cause of their errors is difficult, and creating a test environment that enables end-to-end system testing can be tedious and costly.

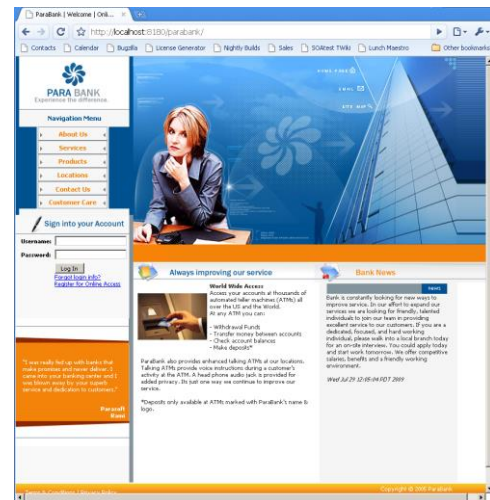
This paper discusses various test and validation techniques that can be used to identify and diagnose problems in a sample business system that comprises multiple components. The sample system architecture and test requirement are based on those that Parasoft has encountered while working with a broad customer base across various industries.

To show how easily the recommended techniques can be automated, we demonstrate how to apply them with Parasoft products. Specifically, we show how to configure repeatable tests that can execute and validate the system at different points of a transaction, as well as how to emulate (or "virtualize") unavailable/inaccessible system components to enable validation of the pieces that we are interested in. The introduced practices can significantly expedite error diagnosis as well as reduce the time and resources required to create reusable and repeatable regression tests.

Introducing the Scenario

Assume that our organization is responsible for the "Parabank" online banking system, which allows customers to open new accounts and transfer money between accounts. The system can be accessed from the web using a browser, as well as from other systems that connect to ATM terminals, mobile applications, or other sources.

We'll refer to this system as *OBA* (for Online Banking Application).

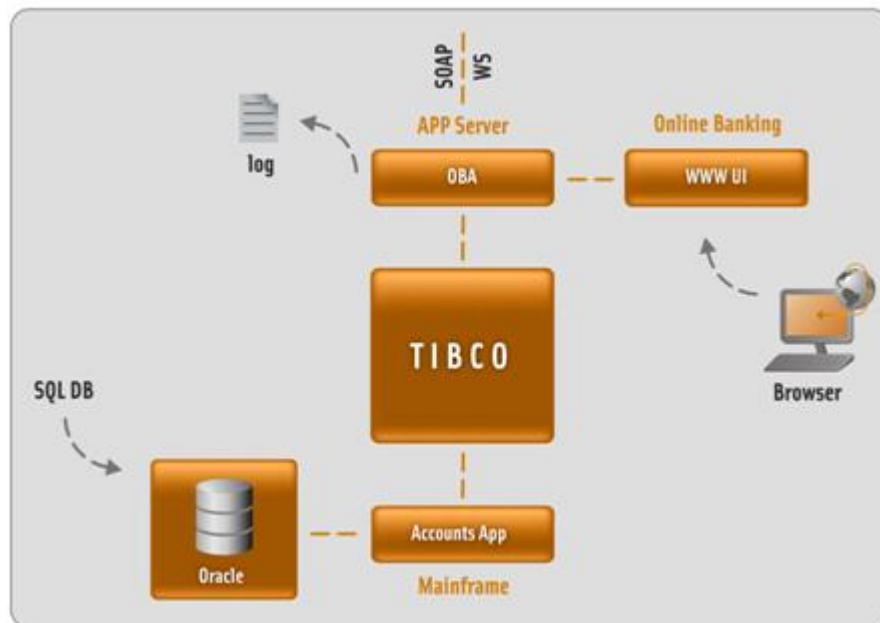


System Architecture

OBA is hosted on a JBoss application server and communicates with an accounts system over TIBCO EMS. The TIBCO bus brokers customer transactions between the online banking application and a mainframe backend where account data is managed. Transactions include transfer requests, deposits, withdrawals, etc. The broker middleware provides performance and reliability between the two tiers; it also facilitates ESB transactions.

Although this example uses TIBCO, the same concepts and solutions we explore in this paper apply to WebSphere, Sonic, WebMethods or any other Message-Oriented Middleware or ESB platform.

The mainframe backend connects with an Oracle database that maintains customer account data. OBA also includes a web services SOAP interface for consumption by other applications that are external to this system (for example, the bank's ATM network, an iPhone app, branch terminals, and so on). Although such a web service interface may not be incorporated into the same OBA application in most real banks, we will make this assumption here in order to keep our example at a reasonable level of complexity.



Such multi-tier integrated systems are very common in enterprises today, and are typically far more complex than this example. Nevertheless, this example should help illustrate the challenges of creating a test environment that enables the various components to be tested and validated in the context of realistic use case scenarios.

Use Case Scenario

Now, let's explore a sample use case scenario for OBA. Assume that we have the following use case:

- Deposit \$500 over the OBA web services SOAP interface (assuming this was initiated by an ATM or some other remote application).
- Log in from the web browser and verify that the web page shows the updated balance.
- Log out.

From a testing perspective, such a simple use case scenario often translates to more elaborate test requirements with many steps; for example:

- Run a reset script to set the database to the initial state.
- Verify account content using a database console tool.
- Use a browser HTTP POST tool to:
 - Paste the transfer SOAP XML template.
 - Edit it to reflect the desired amount (\$500 dollars).
 - Transmit it.

- Log in from the web browser and verify the new account balance.
- Log out.
- (If the scenario failed) Verify the application server log file as follows:
 - Open an FTP window to the OBA server.
 - Fetch the activity log file.
 - Verify the new transaction entry.
- (If the scenario failed) Verify the deposit message on the bus as follows:
 - Log in to the ESB server.
 - Find the message in the message browser.

Challenges

Such steps can place a significant burden on QA engineers because:

- A diverse set of tools and methods would be required to validate system functionality at the various points.
- It is quite time-consuming to do it all repeatedly, so a compromise would need to be reached between the frequency of performing such regression tests and the time and resources required to execute them.
- It is difficult to scale: only a few such scenarios can be tested in that fashion. This can pose a quality risk with consequences on security, reliability, and compliance.

Orchestrating Test and Validation

One way to orchestrate these various test and validation activities over multiple interfaces and protocols is to leverage Parasoft SOAtest, a full-lifecycle quality platform for ensuring secure, reliable, and compliant business processes. This paper will use Parasoft SOAtest to demonstrate how to automate key strategies. For a more general discussion of SOAtest, see <http://www.parasoft.com/soatest>.

Database Initialization

Using SOAtest's DB tool, we define a set-up test that will automatically execute a series of queries and put the database into a state that is suitable for starting the transaction. This includes restoring the account balance to a certain value.

Name:

▼ Tool Settings

☐ File

Input file:

☒ Persist as relative path

☒ Local

Driver:

URL:

Username:

Password:

☒ Close connection

▼ SQL Query

☒ Fail on SQL exception

☐ Encode XML characters in result

Deposit Transaction

Next, we use the SOAP Client tool to deposit \$500. Parasoft SOAtest creates tests automatically from a WSDL, traffic logs, SOA registries, or other sources. It then visualizes the request parameters in an XML tree that allows for visual configuration.

To configure this deposit, we just need to specify the appropriate account number and dollar amount as shown below:

Name:

Views: Operation:

☒ SOAP Body
 ☐ SOAP Header

☒ deposit

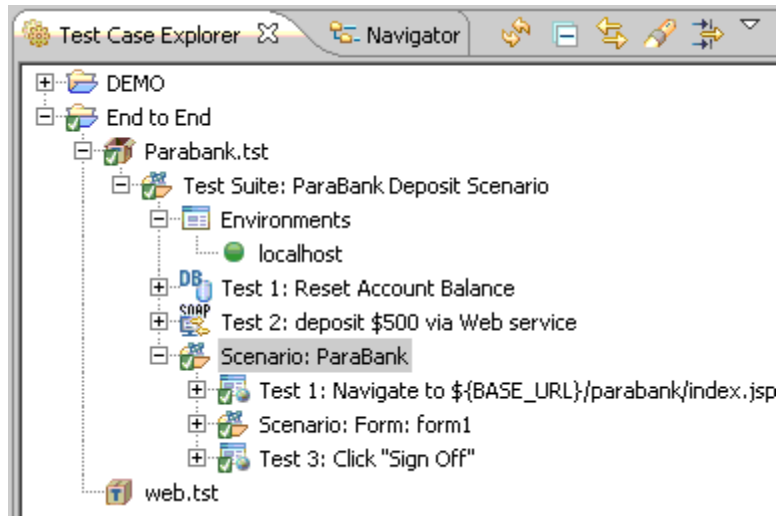
☒ acctnum
 ☒ amount

▼ acctnum

▼ amount

Account Balance Verification

Next, we want to verify that the deposit was posted to the account properly. We could use a web browser to manually log in and check on the balance. However, we want to make this a repeatable, automated step. Thus, we decide to have SOAtest record the browser actions, then replay them as needed during the end-to-end deposit test suite scenario. Once the recording is completed, we copy the log in scenario to the test suite scenario so it can be incorporated into the full transaction.



Diagnosing the Problem

Assume that the account balance did not get updated as expected and it did not reflect the new balance after the deposit. How can we determine what caused this problem?

Application Log Validation

We know that when OBA processes a deposit request, it logs a "Message sent" entry to indicate that a JMS message was produced by OBA and sent to the accounts backend. We could log in to the server machine and check the log. However, we prefer to have a process that we can easily repeat in the future (as with the account balance verification discussed previously). As a result, we decide to automate it.

We automate this process using Parasoft SOAtest's FTP tool, which can be used to fetch files or execute commands over SFTP/FTP. Alternatively, we could use SOAtest's Extension tool with customized code to bring the log file contents from the server machine or other sources. After the FTP tool is added, we attach a Search tool to its output and configure it to search the log contents for the "Message sent" string. The presence of this string indicates that the application has posted a deposit message to the backend over JMS.

▼ Search Terms

New Search Term

[STDOUT] Message sent.

▼ Search Terms Options

Custom Output Message (Non-Regex: Use %0 for matching terms, %1 for pat

Search string not found: %0

☐ Apply current message to all terms

☐ Treat term as word

▼ Search Options

☐ Use Regular Expressions ☐ Ignore Case ☐ Only Report First Occurrence

Display Message If Search Term ☐ Found ☒ Not Found

After we rerun the scenario, we notice that the “Message sent” entry is logged as expected—even though no account balance update is visible in the OBA web UI. In this case, our next logical step is to verify whether OBA really posted a JMS message on the TIBCO bus for consumption by the accounts mainframe backend.

Monitoring Transaction Messages on the Bus

One way to verify if OBA is properly posting a deposit transaction on the bus is to use SOAtest’s Event Monitor to trace the messages that pass through TIBCO EMS. To do this, we add an Event Monitor tool to the test suite, select the TIBCO option, and add the needed messaging provider jars to the SOAtest system properties classpath.

When we run the scenario with the Event Monitor monitoring the TIBCO queue, we notice that no JMS message is being posted on the queue. The SOAtest Event Monitor clearly shows the deposit test executing and completing execution, but no JMS message event occurred.

Test: Reset Account Balance



1063ms



Test: Reset Account Balance

Test: deposit \$500 via Web service



1406ms



Test: deposit \$500 via Web service

Monitoring the Application at the Code Level

Now that we know no deposit message occurred, what do we do next?

There is obviously a problem in the application: it is not posting the messages, and the logs are not indicating any problems. To gain more visibility into the problem, we can monitor the application at the code level and check if the code for sending the JMS messages is being invoked properly at runtime.

The first step in configuring such monitoring is to modify the application's startup script to include the Parasoft Java monitoring jars. There is no need to change code or rebuild, but an extra command line argument needs to be provided. The Parasoft Java runtime monitoring system then connects with the JVM instrumentation API to trace various code execution events and return them from the server to the SOAtest instance running on the tester's desktop. Essentially, the goal is that whenever the test scenario executes, we will also be able to see what application methods are being invoked—both remotely and directly from within SOAtest.

This runtime event monitoring with Java execution traces can serve as a valuable tool for identifying problems and their causes. Although developers can sometimes debug applications from their desktops (where code sources are available using an IDE such as Eclipse), this is often difficult when applications are deployed in a different environment and running under a configuration that is different from the development environment. Moreover, QA engineers commonly lack such access altogether.



To configure OBA for this monitoring, we add the following lines to our server startup script:

```
set MONITOR= [Parasoft Java monitoring jar location]

set PARASOFT_AGENT_OPTS=-
javaagent:"%MONITOR%\MONITOR.jar=soatest,port=5091",instrument=com.parabank.transaction:com.parabank.customer,trace=com.parabank.transaction.TransactionBean:com.parabank.transaction.AccountMessageQueue:com.parabank.customer
set PARASOFT_BOOTCLASSPATH_OPTS=-Xbootclasspath/a:"%MONITOR%\MONITOR.jar"

REM and then the Java startup command to use these variables:

"%JAVA%" %PARASOFT_AGENT_OPTS% %PARASOFT_BOOTCLASSPATH_OPTS% %JAVA_OPTS% "-Djava.endorsed.dirs=%JBOSS_ENDORSED_DIRS%" -classpath "%JBOSS_CLASSPATH%" org.jboss.Main %*
```

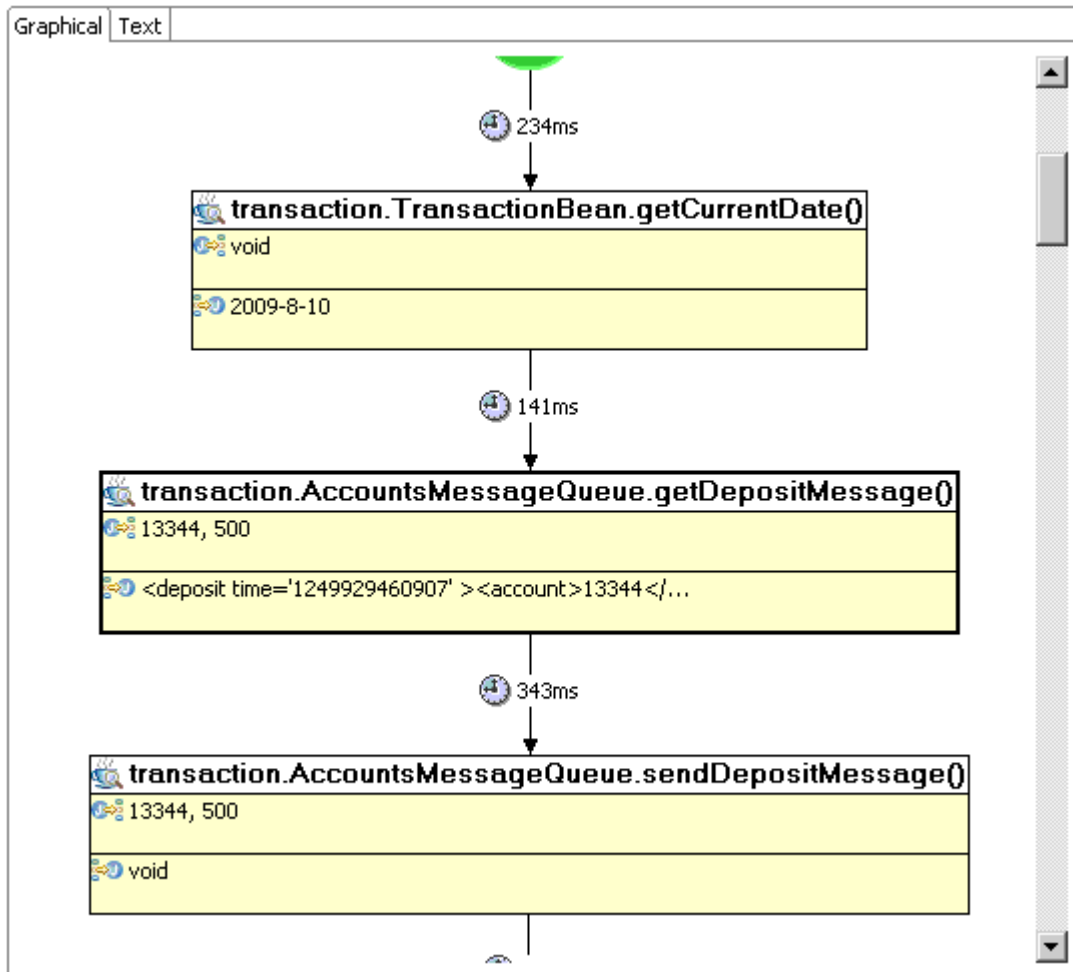
Note that although this example uses a JBoss server, the same general configuration strategy applies to WebLogic, WebSphere, and other popular application servers.

The server is then started as usual, but it is ready to be monitored during the scenario execution. To configure monitoring for the scenario, we add another Event Monitor tool to the test suite; this time, we select the “Instrumented Java Application” option (rather than the TIBCO option that we used for the previous Event Monitor tool).

Name:

Event Source	Options	Event Viewer
Platform: <input type="text" value="Instrumented Java Application"/>		
▼ Java Event Monitor		
Host:	<input type="text" value="localhost"/>	
Parasoft Agent Port:	<input type="text" value="5091"/>	

We run the scenario and notice the various Java method invocation events that occur within the designated classes/packages during scenario execution.



These execution trace details include the parameter values that are passed to the various methods as well as the return values. Such details are helpful for identifying the cause (or at least the location) of the problem in the code. This way, even if a QA engineer does not have direct knowledge about the code and how it is designed, the relevant classes and method calls can still be identified. These classes can then be traced with Parasoft Jtest Tracer to produce JUnit tests that help the development team rapidly reproduce the problem within the development environment.

Generating Tests that Help Development Reproduce the Issue

The first step in using Jtest Tracer's JUnit test generation technology is to configure the server to run with the Tracer library. Again, there is no need to change code or rebuild, but additional arguments need to be added to the server JVM startup. Here, we use the following:

```
-agentlib:pmt=monitor=transaction.*,nostart,port=1234
```


[What the QA Engineer Does](#)

Next, the QA Engineer adds the “Jtest Tracer Client” to the SOAtest Parabank scenario as follows:

Name:

▼ **Tool Settings**

▼ **Jtest Tracer Controller**

Host

Port

▼ **Tracer Options**

☒ Start trace ☐ Stop trace

▼ **Result File**

☒ Save to file

☒ Local file ☐ Remote file

File path

▼ **Monitored Classes/Packages**

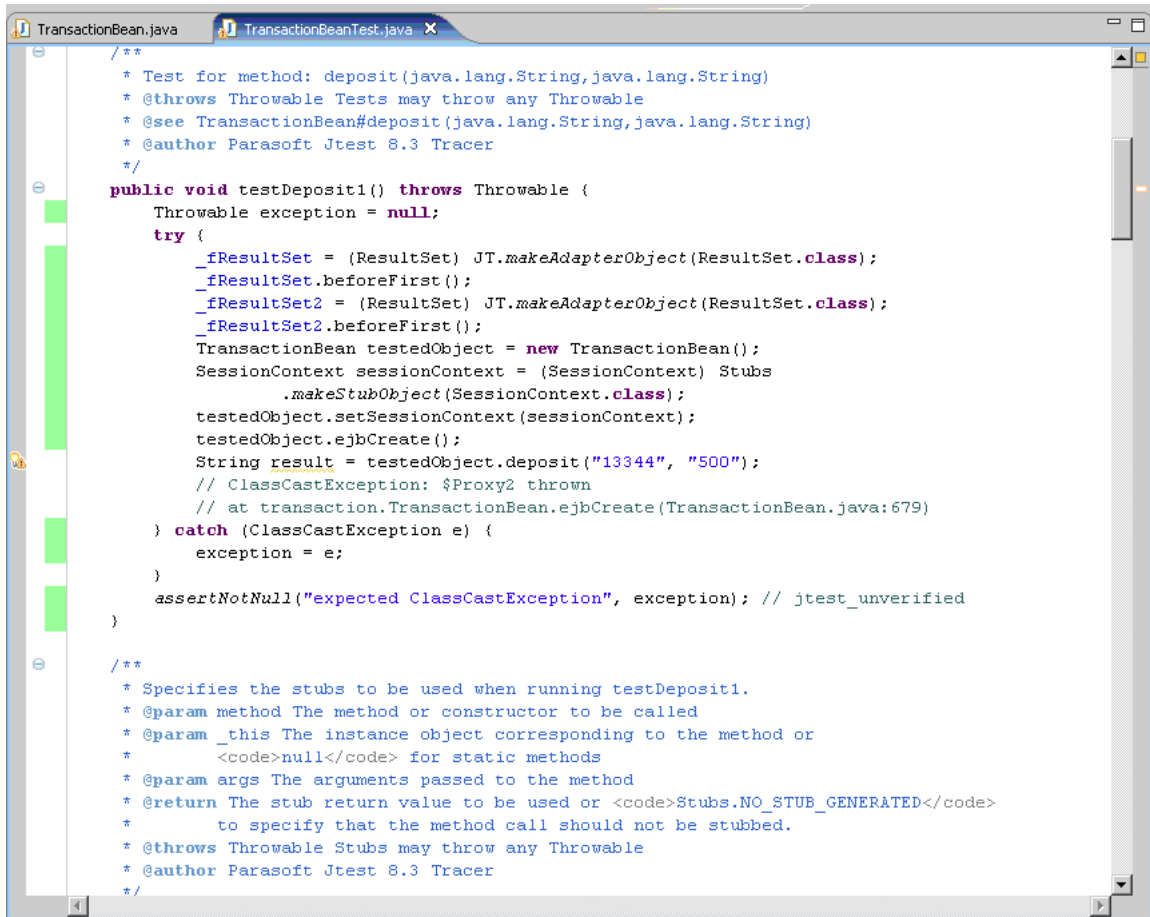
Classes to monitor

Monitored classes/packages
transaction.TransactionBean

Then, the scenario is run as usual. This will produce a “trace.out” file at the specified location. This file is then given to developers.

[What the Developer Does](#)

The developer can then take that trace.out file and use it to generate JUnit tests within the IDE. This will produce one or more JUnit tests that can replay the events at the Java code level—with external code dependencies (such as JDBC calls, JMS connection calls, and so on) stubbed out. This enables the developer to automatically emulate the behavior of the real assets within the QA environment where the problem is occurring.



```

TransactionBeanTest.java
/**
 * Test for method: deposit(java.lang.String,java.lang.String)
 * @throws Throwable Tests may throw any Throwable
 * @see TransactionBean#deposit(java.lang.String,java.lang.String)
 * @author Parasoft Jtest 8.3 Tracer
 */
public void testDeposit1() throws Throwable {
    Throwable exception = null;
    try {
        _fResultSet = (ResultSet) JT.makeAdapterObject(ResultSet.class);
        _fResultSet.beforeFirst();
        _fResultSet2 = (ResultSet) JT.makeAdapterObject(ResultSet.class);
        _fResultSet2.beforeFirst();
        TransactionBean testedObject = new TransactionBean();
        SessionContext sessionContext = (SessionContext) Stubs
            .makeStubObject(SessionContext.class);
        testedObject.setSessionContext(sessionContext);
        testedObject.ejbCreate();
        String result = testedObject.deposit("13344", "500");
        // ClassCastException: $Proxy2 thrown
        // at transaction.TransactionBean.ejbCreate(TransactionBean.java:679)
    } catch (ClassCastException e) {
        exception = e;
    }
    assertNotNull("expected ClassCastException", exception); // jtest_unverified
}

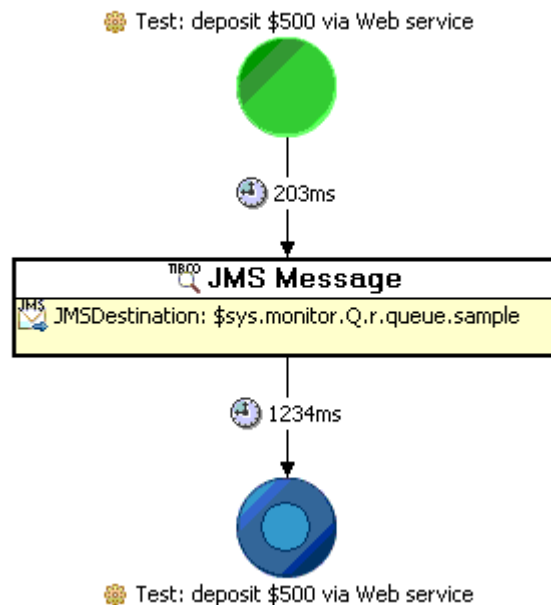
/**
 * Specifies the stubs to be used when running testDeposit1.
 * @param method The method or constructor to be called
 * @param _this The instance object corresponding to the method or
 * <code>null</code> for static methods
 * @param args The arguments passed to the method
 * @return The stub return value to be used or <code>Stubs.NO_STUB_GENERATED</code>
 * to specify that the method call should not be stubbed.
 * @throws Throwable Stubs may throw any Throwable
 * @author Parasoft Jtest 8.3 Tracer
 */

```

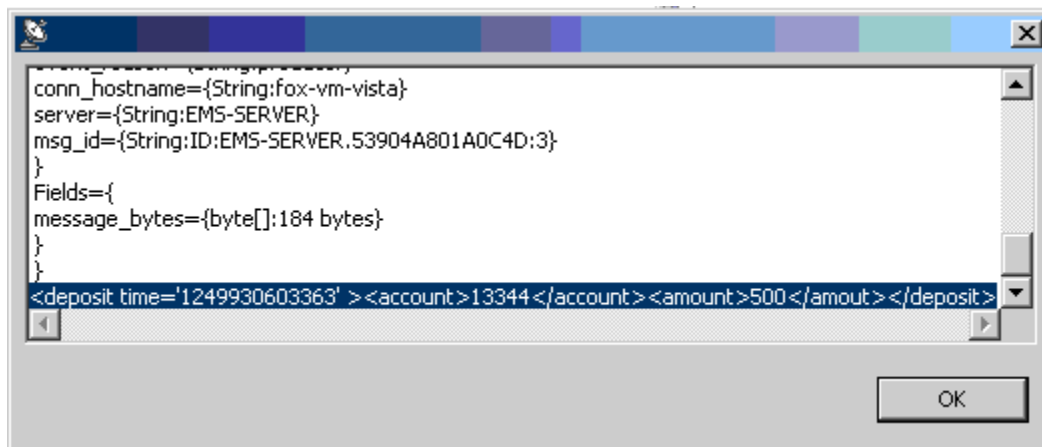
Notice how the testDeposit1() method was automatically generated to invoke the method with the same values that were traced during execution within the QA environment. Also note that the environment context around the class is emulated and stubbed out during the JUnit test execution; this allows the developers to reproduce and analyze the problem without worrying about the database, TIBCO bus, or other dependencies—dependencies that would make the diagnosis and resolution much more complicated.

Re-executing the Scenario after Development Resolves the Problem

Once the problem has been identified and fixed at the code level, we enable the TIBCO bus Event Monitor again and run the scenario to verify that a JMS message is posted by OBA. The following screenshot shows that SOAtest now detects the JMS message on the TIBCO bus:



When we double-click the message in the event viewer, it displays event details—including the message contents.



Creating a Regression Test

A key element in any testing strategy is the ability to capture system behavior in a series of repeatable tests that can be executed on a regular basis in order to ensure that the desired behavior does not regress (change) from the pre-established specifications. Now that we have a working end-to-end scenario for a deposit use case—including validation at the web interface level, server logs, code execution, and JMS messaging layer—various success assertions can be defined at each of these layers. Such assertions ensure that whenever this test is executed in the future, it will validate all these points in the system and alert us to regressions from the desired behavior.

Configuring Validations

Building upon our working scenario, we can configure success criteria on the various scenario steps so the verification is automated. For example, an assertion can be defined on the deposit SOAP response message to validate that the value begins with the “Successfully deposited” string.

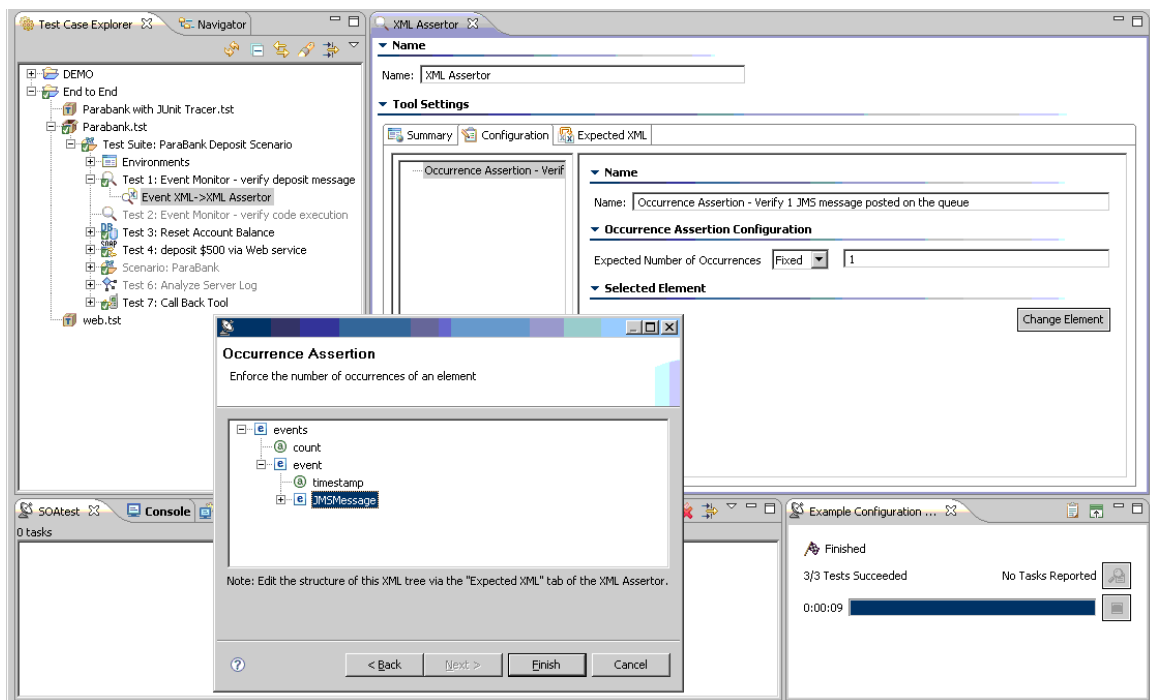
Name:

▼ **String Comparison Assertion Configuration**

Element must Expected Value:

☐ Ignore case

Another validation point can be added to the output of the Event Monitor; this way, the test will fail if no message is detected on the TIBCO bus during the scenario execution.



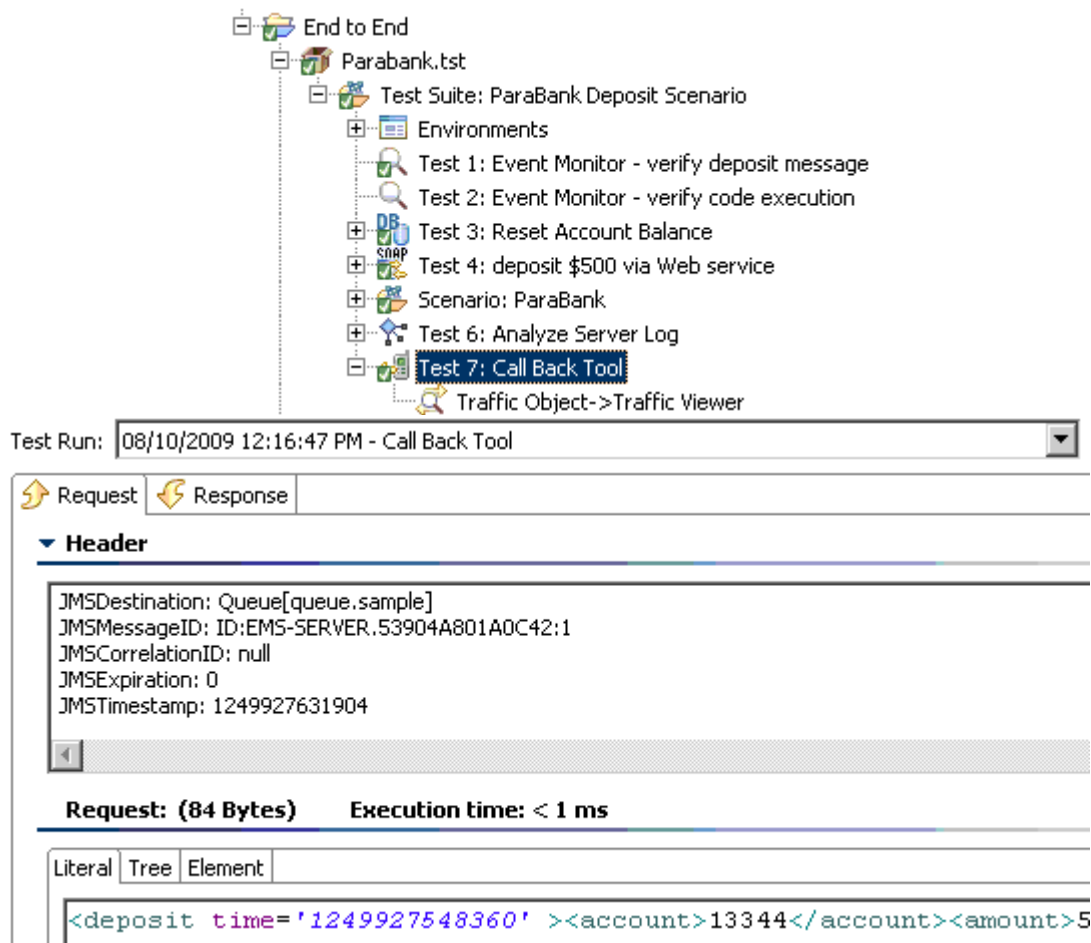
Backend System Availability During Testing

For our sample scenario, we assumed that the backend accounts system would be available for testing. However, it is common that dependent systems are not accessible or available during development and testing—especially when they are managed by other teams, deployed at different geographical locations, or built on a legacy mainframe platform. To address this problem, we can configure a "virtual asset" that will act like the backend system and consume the messages that are produced by OBA.

Virtualizing Application Behavior

In order to make our regression scenario runnable in a test environment that does not include the backend mainframe system, messages posted by the OBA need to be captured and verified as if the original system was actually available. In other words, we need to emulate or virtualize the interaction with the backend mainframe system to remove this dependency.

There are a few different ways that we can configure such emulation. One approach is to add a new test suite scenario step that would consume the JMS message and take it off of the TIBCO queue, just like the actual backend would.



The screenshot shows the Parasoft test suite configuration in a tree view. The hierarchy is: End to End > Parabank.tst > Test Suite: ParaBank Deposit Scenario. Under this suite, there are several tests: Test 1: Event Monitor - verify deposit message, Test 2: Event Monitor - verify code execution, Test 3: Reset Account Balance, Test 4: deposit \$500 via Web service, Scenario: ParaBank, Test 6: Analyze Server Log, and Test 7: Call Back Tool (which is highlighted). Below the tree, a 'Test Run' window shows the date and time: 08/10/2009 12:16:47 PM - Call Back Tool. Below that, a 'Request' tab is selected, showing a JMS message header with the following details: JMSDestination: Queue[queue.sample], JMSMessageID: ID:EM5-SERVER.53904A801A0C42:1, JMSCorrelationID: null, JMSExpiration: 0, and JMSTimestamp: 1249927631904. Below the header, the message body is displayed as an XML snippet: `<deposit time='1249927548360' ><account>13344</account><amount>5`. The message size is noted as 84 Bytes and the execution time is less than 1 ms.

A more flexible and sophisticated emulation could be achieved with service virtualization, which is available with Parasoft Virtualize. Service virtualization provides QA and development teams access to dependent system components that are needed to exercise an application under test (AUT), but are unavailable or difficult-to-access for development and testing purposes. With the behavior of the dependent components "virtualized," testing and development can proceed without accessing the actual live components.

Using service virtualization, you emulate the interactions between the application under test and the dependent applications (here, the backend mainframe system). This behavior is captured as flexible "virtual assets," which can then be customized to suit any specialized testing needs (in



terms of data sources, performance profiles, responses, etc.) and provisioned for ubiquitous access.

At this point, you can sever the ties with the actual dependent applications and test freely against the virtual assets, which you can configure and access however and whenever you want.

How is service virtualization different than stubbing? Virtual assets are simple to create, represent a broad range of realistic behavior, and are easy to update as the components evolve. While stubs are created from the perspective of the test suite in order to "skip" unavailable system components, virtual assets are constructed to make the behavior of constrained components available to the entire team.

To learn more about service virtualization and Parasoft Virtualize, visit <http://www.parasoft.com/virtualize>.

Conclusion

This paper demonstrated how you can apply multiple test and validation techniques to automate end-to-end test and validation for a sample use case. We covered how:

- Automated validations at multiple levels—here, at the web interface, server log, code execution, and JMS messaging layer—can be used in concert to expose and explore functional defects.
- Event monitoring can be used to visualize and trace the intra-process events triggered by tests, facilitating rapid diagnosis of problems directly from the test environment.
- Test case “tracing” from a running application allows you to quickly and easily create test cases that will help development reproduce and resolve the defects that you discover.
- Extending the functional test suite with strategic assertions establishes a regression test suite that, when run regularly, will immediately alert you if system modifications impact the validated functionality.
- Applying service virtualization bridges gaps in the test environment.

These techniques can all be applied through Parasoft SOAtest and Parasoft Virtualize to enable fully-automated continuous validation from a single solution, directly from the test environment—even if parts of the system are incomplete, evolving, unstable, inaccessible, or otherwise unavailable for testing. This allows you to perform more comprehensive testing with your existing resources—ultimately, helping your team to deliver and evolve more secure, reliable, and compliant applications on time and on budget.

About SOAtest

Parasoft SOAtest helps QA teams ensure secure, reliable, compliant business applications with an intuitive interface to create, maintain and execute end-to-end testing scenarios. It was built from the ground up to reduce the complexities inherent in complex, distributed applications.



Since 2002, Parasoft customers such as HP, IBM, Fidelity, Lockheed Martin, and the IRS have relied on SOAtest for:

- Ensuring the reliability, security, and compliance of API and composite applications
- Reducing the time and effort required to construct and maintain automated tests
- Automatically and continuously validating complex business scenarios
- Facilitating testing in incomplete and/or evolving environments
- Validating performance and functionality expectations under load
- Rapidly diagnosing problems directly from the test environment

About Virtualize

Parasoft Virtualize helps development and QA teams create and access any environment needed to develop or test an application. It complements traditional hardware/server virtualization & dramatically reduces the costs associated with configuring & managing test environments. Since it's not feasible to leverage hardware virtualization for every dependent application (e.g., databases, mainframes, 3rd-party systems), service virtualization fills the gap by providing access to their behavior.

Parasoft Virtualize helps development and QA teams:

- Streamline test environment provisioning time and costs beyond traditional virtualization
- Test against constrained dependent resources without scheduling hassles
- Test early and extensively—without access and transaction fees
- Test vs. a broad array of functional & performance conditions—with minimal setup
- Get the exact test environment they need, on demand

About Parasoft

Parasoft researches and develops software solutions that help organizations deliver defect-free software efficiently. By integrating Service Virtualization, Development Testing, and API testing, we reduce the time, effort, and cost of delivering secure, reliable, and compliant software. Parasoft's enterprise and embedded development solutions are the industry's most comprehensive—including static analysis, unit testing, requirements traceability, coverage analysis, functional & load testing, dev/test environment management, and more. The majority of Fortune 500 companies rely on Parasoft in order to produce top-quality software consistently and efficiently as they pursue agile, lean, DevOps, compliance, and safety-critical development initiatives.



Contacting Parasoft

USA	Phone: (888) 305-0041	Email: info@parasoft.com
NORDICS	Phone: +31-70-3922000	Email: info@parasoft.nl
GERMANY	Phone: +49 731 880309-0	Email: info-de@parasoft.com
POLAND	Phone: +48 12 290 91 01	Email: info-pl@parasoft.com
UK	Phone: +44 (0)208 263 6005	Email: sales@parasoft-uk.com
FRANCE	Phone: (33 1) 64 89 26 00	Email: sales@parasoft-fr.com
ITALY	Phone: (+39) 06 96 03 86 74	Email: c.soulat@parasoft-fr.com
OTHER	See http://www.parasoft.com/contacts	