Extensibility API Patterns

This topic describes the Extensibility API, which allows for capturing events by polling at specified time intervals, polling after test execution, and subscribing to an event producer.

Sections include:

Where are the Javadocs?

The Javadocs for the Monitor Tool API can be accessed by choosing **Parasoft> Help**, then opening the **Parasoft SOAtest Extensibility API** book . The resources that directly concern the Event Monitor tool are `com.parasoft.api.IEvent`, its default implementation adapter `com.parasoft.api.Event`, and `com.parasoft.api.EventSubscriber`

Available Patterns

The Extensibility API allows for capturing events using one of three different patterns:

Poll at Specified Time Intervals

This pattern is useful when the events in the system you are trying to monitor are not synchronized with your test execution. For example, you may be interested in pulling data from a logging framework which logs events at specified time intervals, and therefore you wish to capture the logged events in SOAtest at the same intervals in order to ensure that the events are actually retrieved.

An analogy to this is somebody being a fan of a particular monthly magazine, but rather than subscribing to it in order to receive every issue that is published, the reader would visit a book store every month and obtain the latest issue.

A point of interest with this pattern is that if you poll for events before new ones have been made available, then you may get the same events as the last poll or no events at all—depending on how your target framework behaves. This is just like our example reader possibly going to the store and finding only last month's issue since the current month issue has not been released yet.

With this pattern, you can specify a time amount (in milliseconds) which will serve as the wait period in between the executions of your script. For example, if you specify the interval to be 1000 milliseconds, it will cause the Event Monitor tool to execute the code you provide every 1 second. This will continue until one of two events take place:

1. If the Event Monitor is being executed as part of a test suite with other various tests, the periodical user code execution will stop as soon as the last test in the test suite (or last row in the data source, if the test suite tests are iterating over a data source) has finished execution.
2. The maximum monitor execution duration (the value is configured under the Options tab) has been reached.

Poll after Each Test Execution

This pattern is useful when the events in the system you are trying to monitor are generally in sync with your test suite's test execution events. For example, your system's logging framework is triggered immediately after events occur in the system and the events are made available for you to obtain.

With this pattern, the code you provide will be executed immediately after each test in the test suite executes. This pattern is not applicable if you run the Event Monitor tool independently (apart from executing the entire test suite which contains it). In this case, the Event Monitor will stop once:

1. The last test in the test suite (or the last row in the data source if the test suite tests are iterating over a data source) has finished execution.
2. The maximum monitor execution duration (value is configured under Options tab) has been reached.

Subscribe to an Event Producer

This is probably the most common pattern, and Parasoft recommends its use whenever possible. This pattern is useful when the framework you are trying to monitor can allow a subscriber to be triggered immediately upon the occurrence of an event—in other words, it can perform a “call back” function.

For example, the JMS publish/subscribe message pattern is an example of this pattern and it is the pattern used to drive the Sonic, TIBCO and other built-in platforms supported in the Event Monitor tool. In our magazine reader example, this is similar to subscribing to the publication so the latest issue gets delivered to the reader as soon as it is published.

Using “Poll at specified time intervals” and “Poll after each test execution” Patterns

When using these two patterns, the method you select in the User Code section will be executed in accordance to the respective pattern. You may have a method name with any name you wish (be sure it is selected in the **Method** menu).

```
IEvent getEvent(String url, String username, String password, Object connection, ScriptingContext context)
```

- url (String): the value that is provided in the URL field of the Event Monitor Connection section.
- username (String): the value that is provided in the username field of the Event Monitor Connection section.
- password (String): the value that is provided in the password field of the Event Monitor Connection section.
- connection(Object): this object can be optionally provided in order to maintain and reuse the same connection over the multiple script executions of the Event Monitor. See to Maintaining Connections for details.
- context (com.parasoft.api.Context): standard SOAtest scripting context that allows for access-ing variables, data sources values and setting /getting objects for sharing across test executions.

Example (Jython)

```
from com.parasoft.api import *

def getEvent(url, username, password, connection, context):
        return Event("Hello!")
```

The code under getEvent() method would basically handle event retrieval from the system you wish to monitor and return an implementation of com.parasoft.api.IEvent. In this example, we return com.parasoft.api.Event, which is an adapter implementation to that interface and takes a simple String object “Hello!”.

Maintaining Connections

In practice, it is often useful to be able to create a connection to the remote system you want to monitor and reuse that connection for retrieving events (instead of creating a new connection on each User Code invocation). Therefore, in addition to have a method for retrieving an IEvent object as described above, you may choose to add two additional optional methods:

```
Object createConnection(String url, String username, String password, com.parasoft.api.Context context)

Object destroyConnection(Object connection, com.parasoft.api.Context context)
```

createConnection would create and return an object handle to the monitoring connection, while destroyConnection takes that same object and allows for you to provide code for a graceful disconnection.

The Event Monitor looks for the presence of these two optional methods. If you want to add them, be sure to use the exact method signatures. createConnection() is invoked once at the beginning of the Event Monitor execution and destroyConnection is invoked once at the end. The event retrieval method—for example, getEvent() above—is potentially invoked multiple times during Event Monitor test run in accordance with the selected pattern. The connection object you create with the createConnection method is passed to the event retrieval method so you can potentially use that connection to return an event.

Example

```
from com.parasoft.api import *

def getEvent(url, username, password, connection, context):
        return connection.getEvent()
```

Using the “Subscribe to an event producer” pattern

With this pattern, Event Monitor expects a single method (with any name you wish—as long as the name is selected in the **Method** drop down menu) and with the following signature:

```
EventSubscriber getEventSubscriber(String url, String username, String password, Context context)
```

The arguments descriptions are provided in the previous patterns.

In this case, you need to provide a Java implementation of an EventSubscriber (by inheriting from com.parasoft.api.EventSubscriber). The methods to implement are:

`public boolean start() Throws Exception` and `public boolean stop() Throws Exception`

The start method will be invoked automatically by Event Monitor when the monitoring begins, and the stop method will be invoked automatically when the Event Monitor Execution finishes. The assumption under this pattern is that your EventSubscriber implementation would take care of connecting to your target system and subscribe to its event producing framework in start() and then unsubscribe and disconnect in stop(). An example implementation for subscribing to TIBCO EMS message monitoring topics is provided below. This actually mirrors the pattern used by the built-in TIBCO EMS platform of the Event Monitor.

Example

This example is also available under the examples scripting directory that ships with SOAtest. It is included as an Eclipse project in a zip archive that can be imported to your Eclipse workspace. It requires tibjms.jar from TIBCO EMS and com.parasoft.api.jar to be added to the classpath in order to build and run. com.parasoft.api.jar is available at `<SOAtest Installation Directory>\plugins\com.parasoft.xtest.libs.web_<SOAtest version>\root`.

```
import com.parasoft.api.*;
import com.tibco.tibjms.*;
import javax.jms.*;

public class TIBCOEventSubscriber extends EventSubscriber {
            private ConsumerRunnable _consumerRunnable;
        private Connection _connection;
            private String _dest;
            private boolean _started = false;
        public TIBCOEventSubscriber(String url, String username, String password, String destination) throws
JMSException {
                _dest = destination;
                QueueConnectionFactory factory = new TibjmsQueueConnectionFactory(url);
                _connection = factory.createQueueConnection(username, password);
        }
        public boolean start() throws Exception {
                Session session = null;
                session = _connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
                Destination destination = session.createTopic(_dest);
                MessageConsumer msgConsumer = session.createConsumer(destination);
                _connection.start();
                _started = true;
                _consumerRunnable = new ConsumerRunnable(msgConsumer);
                Thread thread = new Thread(_consumerRunnable);
                thread.start();
                Application.showMessage("Monitoring started on " + _dest);
                return true;
        }

        public boolean stop() throws Exception {
                _started = false;
                Thread.sleep(1000);
                if (_connection != null) {
                        _connection.close();
                        Application.showMessage("Monitoring connection closed");
                }
                if (_consumerRunnable != null && _consumerRunnable.getException() != null) {
                        throw _consumerRunnable.getException();
                }
                return _started;
        }

        private class ConsumerRunnable implements Runnable {
                private MessageConsumer msgConsumer;
                private Exception t;

                public ConsumerRunnable(MessageConsumer msgConsumer) {
                        this.msgConsumer = msgConsumer;
                }
                public void run() {
                        while(_started) {
                                try {
                                        MapMessage msg = (MapMessage)msgConsumer.receive(500);
                                        if (msg == null) {
                                                continue;
                                        }
```

```
                                        Message actualMessage = null;
                                        try {
                                                byte[] bytes = msg.getBytes("message_bytes");
                                                if (bytes != null && bytes.length > 0) {
                                                        actualMessage = Tibjms.createFromBytes(bytes);
                                                }
                                        } catch (JMSException e1) {
                                        }
                                        // you can provide your own extension of Event in order to customize the
event
                                        // getLabel(), toString() and toXML() outputs
                                        IEvent event;
                                        if (actualMessage == null) {
                                                event = new Event(msg);
                                        } else {
                                                event = new Event(actualMessage);
                                        }
                                        event.setTimestamp(msg.getJMSTimestamp());
                                        onEvent(event);
                                } catch (JMSException e) {
                                        Application.showMessage(e.getClass().toString() + ": " + e.
getMessage());
                                        try {
                                                msgConsumer.close();
                                        } catch (JMSException e1) {
                                                Application.showMessage(e1.getClass().toString() + ": " + e1.
getMessage());
                                        }
                                        t = e;
                                        _started = false;
                                }
                        }
                }
                public Exception getException() {
                        return t;
                }
        }
}
```