



TestMaker 5 Completion Specification

PushToTest™ TestMaker version 5 has been in development for the past year.

This document specifies the projects remaining to be completed.

Frank Cohen
Email: fcohen@pushtotest.com

November 22, 2006

PushToTest is a trademark of the PushToTest Company.
This document is © 2006 PushToTest. All rights reserved.

Table of Contents

1. Current Status	1
2. Code Repository	2
3. Development Environment	2
4. Functional Specifications	3
com.pushtotest.testmaker.controller (2 days)	3
TestMaker 5.5 Includes Distributed Test Environment	4
TestNetwork Client Code for XSTest (5 Days)	4
TestScenario.xml Test Configuration Files (11 days).....	6
New Agent Wizard Improvements (11 days).....	15
Protocol Handler Changes (3 days)	16
Operations and Deployment Improvements (7 days).....	18
5. Bug Fixes	20
6. Additional Ideas for Future TestMaker Versions	21
7. Appendix A: TestScenario.XML Example Document	22

1. Current Status

TestMaker 5 has been in development for the past year. The new TestMaker is a compendium of bug fixes and new features developed by the TestMaker developer community and new features funded by PushToTest. This document specifies the remaining projects needed to finish TestMaker 5.

TestMaker 5 will feature these new functions and improvements:

- XML-based load test scenario description document, including graphical interface for scenario creation and editing
- New graphical interface to operate tests
- Modify, pause, stop tests while they are running
- Run tests on distributed set of TestNodes
- Enhanced protocol handlers, including new SIP, enhanced SOAP, and enhanced HTTP with non-TLD cookie handling
- Enhanced XML support for testing REST and AJAX applications
- Live results charts as a test runs
- Results analysis (Bar Charts and CSV files) after a test runs
- Email notification when running tests have problems
- Dynamic JAR loading of test resources
- Resource monitoring (CPU, Network, Memory) as a test runs
- Multidimensional load test scenarios
- HTTPS support in the test recorder
- New log handler, including automatic archival of test results
- An update of all documentation, plus new tutorials and example test code

2. Code Repository

TestMaker 5 source code is found in the cvs.pushtotest.com server using :pserver:anonymous@cvs.pushtotest.com:/var/cvsroot. Please send email to info@pushtotest.com to request committer privileges.

- tm5 – the TestMaker project
 - a. Subproject for pttmonitor (resource monitor to record and return CPU, network, and memory use)
 - b. The code currently uses JAR libraries for a branch of the j text editor, a branch of the Maxq proxy recorder, and the TOOL protocol handler and resource library. Just as in the tm44 (TestMaker 4.4) repository the source to j, MaxQ, and TOOL need to be part of the tm5 repository directly.
- testnetwork – server side environment to remotely execute test scripts, deployed as a WAR file.

3. Development Environment

TestMaker 5 is a NetBeans 5.5 project; However, there are no NetBeans dependencies so as to allow developers to build TestMaker with any IDE or Ant/Java compiler. TestMaker 5 requires:

- Netbeans-style Ant build scripts for everything
- Apache Ant 1.6.5 or greater
- Java 1.5 or greater
- SWING
- Matisse SWING enhancements

4. Functional Specifications

The following projects are either partially started or needed for the TestMaker 5 release. We estimate a full-time effort would take 37 days to complete TestMaker 5.

com.pushtotest.testmaker.controller (2 days)

We often hear that new users are not sure how to get started with TestMaker. TestMaker 5 introduces a new GUI-based controller to make it more apparent how to create scalability and performance load tests, test scenarios for multi-step functional tests, and jUnit TestCases. The new controller also provides an immediate way to run tests.

TestMaker saves window size and location in the TestMaker.properties file.

See the com.pushtotest.testmaker.controller package for supporting classes. Much of the controller is complete.

A key new feature is in the guiTestScenarioPanel's use of a JSlider to control a running test. The slider scrolls through the dimensions of a load test. While a test runs, the user may move the slider to a new test case. The test continues to operate from the chosen test case slider position. To make this function, the TestNode master needs to communicate through the SnapShot communications protocol to determine the next test case to run.

guiTestScenarioPanel has a bug in the vertical placement of new guiTestRunPanel elements. The vertical spacing is larger than the vertical size of the panel.

guiTestRunPanel replaces the previous functions in the com.pushtotest.testmaker.gui.TestNetwork package, including showing as-the-test-operates status, live results charts, and text-output.

The controller functions for stop, pause, shrink and grow need to be implemented.

TestMaker 5.5 Includes Distributed Test Environment

TestMaker 4.x sold the TestNode installation separately as a commercial software license. TestMaker 5 delivers a new version of the TestNetwork client and the TestNode installation. TestMaker 5 runs tests in the following environments:

XSTest Load Test	Operates a scalability and performance test. A TestScenario.xml file defines the operating parameters of the test. By default TestMaker runs a local instance of the TestNode to run load tests. Users run a load test using the controller interface.
Multi-step Functional Test	The TestMaker Proxy Recorder creates multi-step functional tests in a Jython script. TestMaker 5 runs Jython scripts in the local virtual machine in a thread using the <code>com.pushtotest.testmaker.gui.AgentRunner</code> class and <code>com.pushtotest.testmaker.gui.ScriptEditor</code> package.
Jython script	Runs the main method of a single Jython script using the <code>com.pushtotest.testmaker.gui.AgentRunner</code> class and <code>com.pushtotest.testmaker.gui.ScriptEditor</code> package.

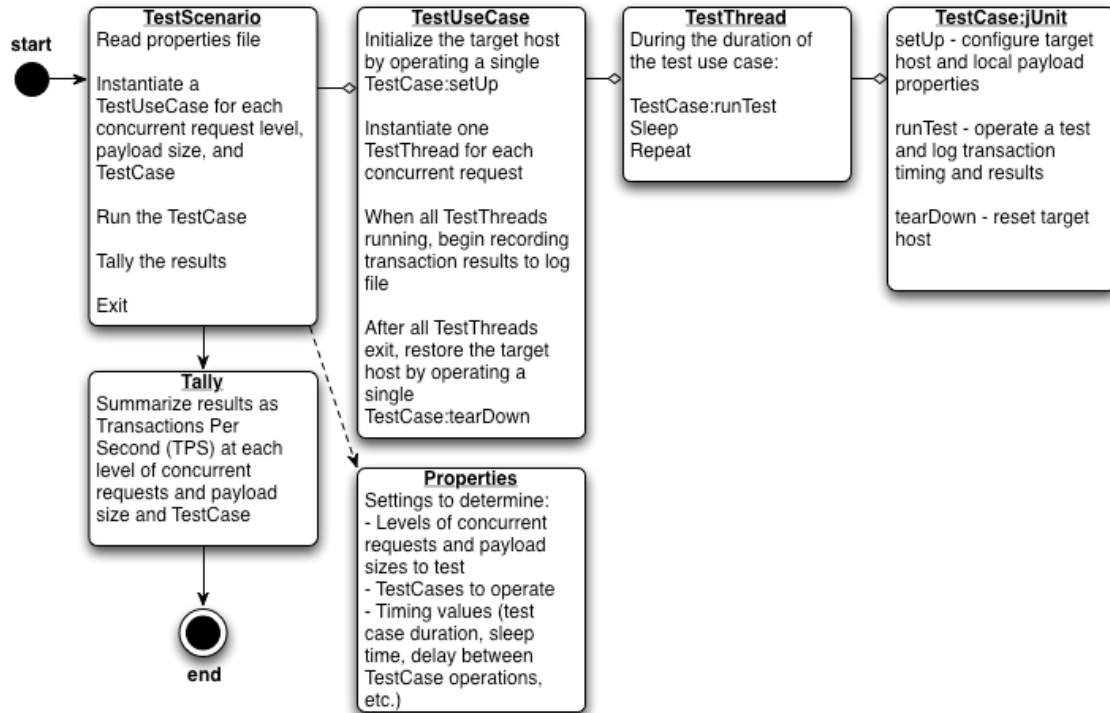
TestNetwork Client Code for XSTest (5 Days)

TestMaker 5 consolidates the client code for XSTest into three packages:

<code>com.pushtotest.testmaker.controller</code>	Graphical interface to provide user interaction with load tests.
<code>com.pushtotest.testmaker.filehandlers.testscenario</code>	Classes to work with TestScenario.xml definition files
<code>com.pushtotest.testmaker.XSTest</code>	Classes to run tests, including the client to operate the test on a group of Test-Nodes

XSTest operates a functional test at various levels of concurrent request sizes to determine the scalability index of the target host. The following chart explains the use case that XSTest implements.

XSTest



`com.pushtotest.testmaker.XSTest` implements the XSTest framework.

TestMaker 4.x uses a combination of Jython and Java code to implement the client. TestMaker 5 uses an entirely Java client. Most of this conversion for the client to Java is complete. However, `com.pushtotest.testmaker.XSTest.client.TNTally` is not yet finished.

The TestNetwork package for the client and the TNMaster package for the TestNode server do not yet support TestScenario.xml test description options. The Snapshot communications protocol enables the `com.pushtotest.testmaker.XSTest.client.TNMaster` class to coordinate test cases and options with the TestNode.

Additionally, `com.pushtotest.testmaker.XSTest.client.TNMaster` does not yet implement support for the operating parameters in a TestScenario.xml description file.

Remove the `com.pushtotest.testmaker.gui.TestNetwork` package.

TestScenario.xml Test Configuration Files (11 days)

com.pushtotest.testmaker.XSTest.TestScenario parses TestScenario.xml definition files using classes in com.pushtotest.testmaker.XSTest.dimensions. The TestScenario.xml definition contains the following elements. See the example TestScenario.xml definition document in the appendix.

<basics>

Defines the test scenario name and default directory for test scenario.

```
<basics>
  <name>Geospatial Test</name>
  <defaultdirectory>./geospatialtest</defaultdirectory>
</basics>
```

The <name> element defines the name for the test scenario. The <defaultdirectory> element defines the default path to find scripts, JAR files, and other resources.

<testnodes>

Defines the name and URL to the TestNodes that will run the test scenario.

```
<testnodes>
  <node name="localhost" location="http://localhost:8080/TestNetwork/TestNode" />
</testnodes>
```

<arguments>

The <arguments> element defines a set of name/value pairs that are passed-in to the running test to allow for parameterized test scenarios.

```
<arguments>
  <argument name="targethost" value="10.0.0.1/gis"/>
  <argument name="id" value="admin"/>
  <argument name="password" value="admin"/>
</arguments>
```

The running test has these name/value pairs available through a dictionary (Jython) and HashMap (Java.)

<resources>

Identifies jar files to be dynamically loaded when TestMaker runs the test.

```
<resources>
  <jar path="./lib/test.jar"/>
  <jar path="./lib/xmlbeans2/xmlbeans.jar"/>
</resources>
```

<logs>

Defines one or more log handlers to handle logged entries while a test runs.

```
<logs>
  <log type="file" path="logs/mylog.txt" level="2"/>
</logs>
```

The <logs> element defines the global logger location and logging-level.

<tally>

Defines options for results analysis.

```
<tally enabled="true" destination="./results" archive="true" basefilename="gisresults_"/>
```

<chart>

Defines the live and tally charts values for bar charts and comma-separated-value (CSV) files

```
<chart liveresults="true"/>
```

<monitor>

Defines parameters for the resource monitor (CPU, Network, Memory.)

```
<monitor enabled="true"/>
```

<options>

Defines default and optional values for the overall test.

```
<options>
  <sleeptime>0</sleeptime>
  <delayBetweenStartingUseCaases time="10"/>
  <delayBetweenTestCases time="60"/>
</options>
```

The sleeptime defines the default number of seconds for TestMaker to sleep between operating <testcase> values.

<notifications>

Sets the notification parameters for email notification when a test moves into a different state.

```
<notifications>
  <email on="start" email="fcohen@pushtotest.com"/>
  <email on="end" email="fcohen@pushtotest.com"/>
  <email on="interrupted" email="fcohen@pushtotest.com"/>
  <email on="testcasestart" email="fcohen@pushtotest.com"/>
  <email on="hourly" email="fcohen@pushtotest.com"/>
</notifications>
```

<testusecase>

Defines the functional unit test that will be used during the test operation. <testusecase> operates a single unit test repeatedly, a sequence of unit tests repeatedly, and a mix of sequences repeatedly during the duration of the test. The following sections show how to configure these tests in the TestScenario.xml file.

<testusecase> For a Single Dimensional Functional Test

The simplest form of a test looks like this:

```
<testusecase name="simpletest">
  <test>
    <run name="geo_insert" testclass="com.pushtotest.example.test" method="runTest" langtype="java"/>
  </test>
</testusecase>
```

In the above example TestMaker stages a single test to run for the default time of 1 minute. At the start of the test period TestMaker will instantiate the Java class named *test* in the *com.pushtotest.example* package. During that 1 minute of the test TestMaker will call the *runTest* method of the *test* object, then sleep for the default 1 second, then repeat the call to *runTest* until the time is complete. The *langtype* attribute determines the language in use, including Java and Jython.

Often the service under test or the test itself requires setup and teardown instructions.

```
<testusecase name="simpletest">
  <setup name="geo_insert" testclass="com.pushtotest.example.test" method="setUp" langtype="java"/>
  <test>
    <run name="geo_insert" testclass="com.pushtotest.example.test" method="runTest" langtype="java"/>
  </test>
  <teardown name="geo_insert" testclass="com.pushtotest.example.test" method="tearDown"
    langtype="java"/>
</testusecase>
```

In the above example TestMaker stages a single test to run for the default time of 1 minute. At the start of the test period TestMaker will instantiate the Java class named *test* in the *com.pushtotest.example* package. It will run the *setUp* method of the *test* object once. Then during that 1 minute of the test TestMaker will call the *runTest* method of the *test* object, then sleep for the default 1 second, then repeat the call to *runTest* until

the time is complete. At the end of the test period TestMaker will call the *teardown* method once and then end the test.

When you need to test a service by calling several functions in series, use the following:

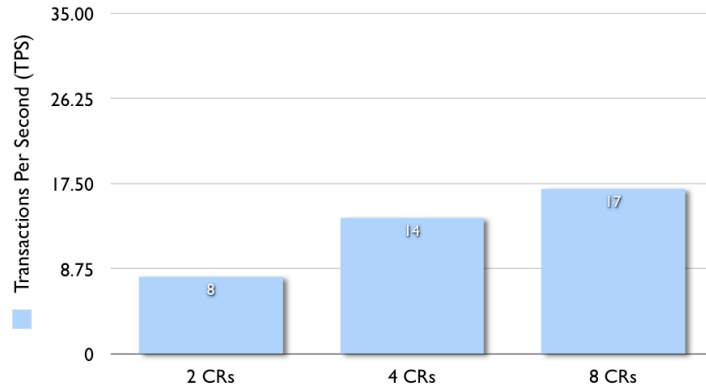
```
<testusecase name="simpletest">
  <setup name="geo_insert" testclass="com.pushtotest.example.test" method="setUp" langtype="java"/>
  <test>
    <run name="geo_insert" testclass="com.pushtotest.example.test" method="insert" langtype="java"/>
    <run name="geo_insert" testclass="com.pushtotest.example.test" method="query" langtype="java"/>
    <run name="geo_insert" testclass="com.pushtotest.example.test" method="update" langtype="java"/>
    <run name="geo_insert" testclass="com.pushtotest.example.test" method="delete" langtype="java"/>
  </test>
  <teardown name="geo_insert" testclass="com.pushtotest.example.test" method="tearDown"
    langtype="java"/>
</testusecase>
```

In the above example TestMaker stages a single test to run for the default time of 1 minute. At the start of the test period TestMaker will instantiate the Java class named *test* in the *com.pushtotest.example* package. It will run the *setup* method of the *test* object once. Then during that 1 minute of the test TestMaker will call the *insert* method of the *test* object, then sleep for the default 1 second, then call the *query* method, sleep, run the *update* method, sleep, run the *delete* method, sleep, then repeat the call to *insert* until the time is complete. At the end of the test period TestMaker will call the *teardown* method once and then end the test.

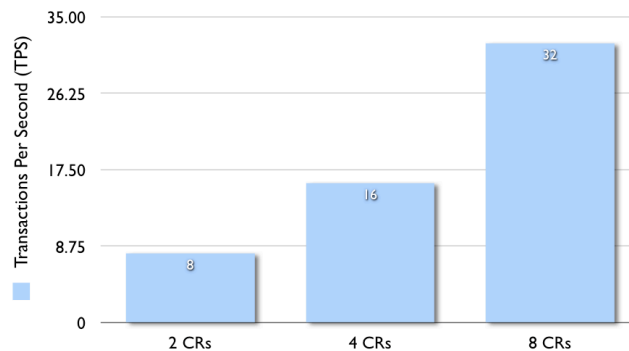
The previous example use cases are functional tests to determine that a service operates correctly. The recorded results logs show how long each operation takes the service to complete. The next examples show how to turn a functional test into a load test to understand the scalability and performance of a target service.

<testusecase> For Load Tests To Determine A Scalability Index

A Scalability Index shows the performance of a target service at a variety of load levels. For instance, TestMaker will run a testusecase at predefined concurrently running threads (called Concurrent Requests and abbreviated as CRs.) By contrasting the number of completed testusecases (measured as Transactions Per Second and abbreviated as TPS) the Scalability Index. For instance, the following chart shows the TPS results as measured at 3 levels of CRs.



The chart above shows that the service does not scale in proportion to the number of CRs. For example, if a service performance 8 TPS at 2 CRs one would expect the same service could perform twice as many TPS at twice the CRs. The chart above shows that at 4 CRs the service performs only 14 TPS. Even worse, at 8 CRs the service performs only 17 TPS. A perfectly scalable system would produce a Scalability Index looking like the following chart.



The `<testusecase>` definition of a load test builds on the prior examples. For instance, the following definition stages a load test at 2, 4, and 8 CRs

```

<dimensions>
  <crlevels>
    <crlevel value="2" />
    <crlevel value="4" />
    <crlevel value="8" />
  </crlevels>

  <testusecases>
    <testusecase name="simpletest">
      <test>
        <run name="geo_insert" testclass="com.pushtotest.example.test"
          method="runTest" langtype="java"/>
      </test>
    </testusecase>
  </testusecases>
</dimensions>

```

In the above example TestMaker stages a load test to run for the default time of 1 minute. TestMaker instantiates 2 concurrently running threads at the start of the test period. Within each thread TestMaker instantiates the Java class named *test* in the `com.pushtotest.example` package. Within each thread and during that 1 minute of the test, TestMaker will call the *runTest* method of the *test* object, then sleep for the default 1 second, then repeat the call to *runTest* until the time is complete. At the end of the test period TestMaker ends the threads. TestMaker then starts the next test by repeating the test at 4 threads, running the test, and then doing the same at 8 threads.

<testusecase> For a Multi Dimensional Test

TestMaker uses the <crlevels> element to define a single dimension of a test. TestMaker supports multiple dimensions in a test. For instance, TestMaker may vary the message size of a call to a SOAP-based Web Service by using the <messagesizes> dimension element.

```
<messagesizes>
  <messagesize value="1" />
  <messagesize value="2" />
  <messagesize value="3" />
</messagesizes>
```

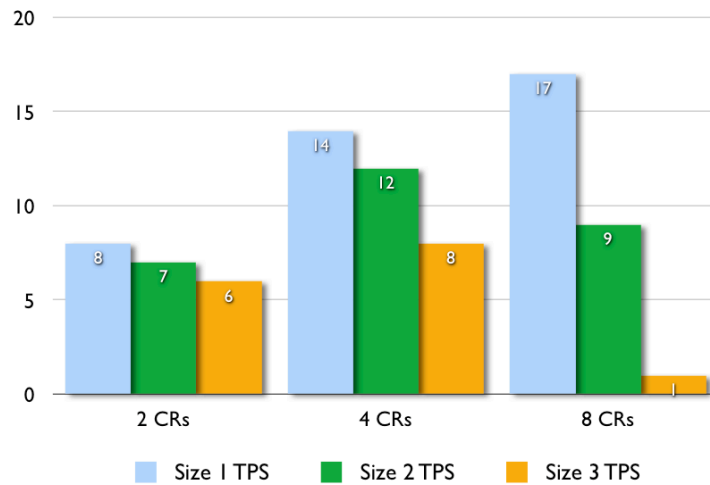
In the above example, TestMaker will operate a test at message size of 1, and then repeat the test at message sizes of 2 and 3. The called test case object defines the meaning of the message size value. Each used dimension geometrically increases the number of tests. For instance, the test definition below causes TestMaker to run 9 test cases (3 levels of CRs times 3 levels of message size.)

```
<dimensions>

  <crlevels>
    <crlevel value="2" />
    <crlevel value="4" />
    <crlevel value="8" />
  </crlevels>
  <messagesizes>
    <messagesize value="1" />
    <messagesize value="2" />
    <messagesize value="3" />
  </messagesizes>

  <testusecases>
    ...
</dimensions>
```

The following chart illustrates the resulting Scalability Index of the above-defined multidimensional test.



The chart above shows a scalability index as the CRs increases and the message size levels increase. Consider the contrast of TPS results between the 2 CR level and 8 CR level. At the 2 CR level the change in message size reduces the TPS levels, but at the 8 CR level the increase in message size dramatically reduces the TPS level.

<sequence> For Multiple Test Use Cases

An earlier example shows testing a service by calling several functions in series. TestMaker considers the set of functions a <sequence>. TestMaker may operate multiple sequences for each <testusecase>. Real world use cases often need more than one sequence to be operating at a time. For instance, testing a database doing insert and update operations concurrently may uncover contention problems. The following script creates two sequences.

```

<crlevels>
  <crlevel value="2" />
  <crlevel value="4" />
  <crlevel value="8" />
</crlevels>

<testusecase name="sequencetest">
  <setup name="geo_insert" testclass="com.pushtotest.example.test" method="setUp" langtype="java">
    <argument name="geoval" value="1738"/>
  </setup>
  <sequence name="inserts" proportion="80">
    <test name="test1">
      <setup name="geo_insert" testclass=" com.pushtotest.example.test " method="seqSetup"
        langtype="java">
        <argument name="geoval" value="1738"/>
      </setup>
      <run name="geo_insert" testclass="" method="" langtype="java">
        <argument name="geoval" value="1738"/>
      </run>
      <teardown name="geo_insert" testclass="com.pushtotest.example.test"
        method="geoteardown" langtype="java"/>
    </test>
  </sequence>
</testusecase>

```

```

</sequence>
<sequence name="queries" proportion="20">
  <test>
    <setup name="geo_insert" testclass="com.pushtotest.example.test" method="seqSetup"
      langtype="java">
      <argument name="geoval" value="1738"/>
    </setup>
    <run name="geo_insert" testclass="" method="" langtype="java">
      <argument name="geoval" value="1738"/>
    </run>
    <teardown name="geo_insert" testclass="com.pushtotest.example.test"
      method="geoteardown" langtype="java"/>
  </test>
</sequence>
<teardown name="geo_insert" testclass="com.pushtotest.example.test"
  method="geoteardown" langtype="java"/>
</testusecase>

```

In the above example, TestMaker operates the test at 2, 4, and 8 CR levels. At each CR level, TestMaker instantiates the corresponding number of concurrently running threads. The threads instantiate one of the two defined <sequence>s. TestMaker sets 80% of the threads to operate the *inserts* sequence and the remaining 20% of the threads to operate the *queries* sequence.

Additional Details for Test Scenario Description Files

- TestCase operations support instantiating and calling Jython and Java class.
- Communication between the TestNetwork client and the TestNode server uses the *snapshot* protocol to tell the TestNodes which <TestCase> to run.
- TestMaker 5 needs a command-line and programmatic API to start a test and return the results.

New Agent Wizard Improvements (11 days)

The TestMaker New Agent Wizard is a proxy recorder that watches communication between a browser and target host and creates a Jython functional test script that implements the JUnit TestCase interface. TestMaker 5 features improvements to the Wizard to handle:

- Services using HTTPS protocols
- Services using AJAX, REST, and other-XML payloads
- SOAP-based Web Services that provide WSDL definitions

Recorder supports HTTPS (2 days)

The proxy recorder will act as an HTTPS host by providing a digital certificate over an HTTPS request to the browser and decrypting the HTTP request. The proxy will record the message and then make an HTTPS request to the target. The target host response is repackaged as the response to the browser in the HTTPS response. This system is not expected to work when the host requires client-side certificates.

Recorder supports XML: SOAP, AJAX, REST (4 days)

The proxy recorder understands when the payload of an HTTP request is in XML format. The recorder writes the Jython TestCase using the HTTPProtocol handler (or HTTPS) and uses the Body class to encode the XML payload. The expectation is that customization to the XML payload is possible using the JDOM XML parser APIs that are included with TestMaker.

Test Recorder support multi-part forms (1 day)

The proxy recorder – a branch of the MaxQ project – and the TestMaker TOOL library supports multi-part forms. However, TestMaker 4 does not implement a way to create a Jython TestCase that uses multi-part forms. TestMaker 5 supports multi-part forms.

Test Record and New Agent Wizard supports skinny TestCase skeleton (1 day)

The default skeleton Jython TestCase uses an `__init__` method with several parameters. The parameters are unnecessary and clutter the resulting test script. TestMaker 5 uses a simpler "skinny" skeleton script.

SOAP New Agent Wizard (1 day)

The New Agent Wizard for SOAP services that offer a WSDL description document needs to be revised to output the new "skinny" skeleton script.

Notes: A subsequent TestMaker release will add the ability of the SOAP Wizard to create TestCases for SOAP services that use both SOAP RPC Literal and SOAP Document Literal encoding, including XML Schema defined message formats.

Protocol Handler Changes (3 days)

TestMaker 5 enhances the TOOL protocol handler support in three ways:

- Adds a new HTTP protocol handler that uses Apache HTTPClient
- Adds a new SOAP protocol handler that uses Apache SOAP
- Adds a SIP protocol handler

SIP protocol handler (1 day)

(The following is a design from Kyle Bell, kylebell@tango-networks.com)

Thinking about your protocol handler interface and how they implement HTTP, the same could be provided for SIP/SIPS. Get a PH for SIP, add the body and headers for a message which implements a particular SIP method much the same as GET and POST for http only these methods would be INVITE, MESSAGE, INFO, SUBSCRIBE, etc for the Request messages. There would be a necessity to build the URI for the Request much the same as an HTTP message and I would image your add header interface would work for standard SIP headers to the message where you specify the header type and the user provides a string representing the header content. The body of the message would likely contain standard SDP for the application part or plain text in the case of an instant message for the MESSAGE method.

Once the message has been built, calling the connect function would then connect to the specified URI, send the message and collect a response. These responses would then be given to the user as text as your normal response, or if you are feeling randy, you might parse the response message and provide the user with the standard textual description of the canned response code (SIP standard response) or give the user the capability to look for a particular bit of text in a particular header.

Typical implementations of SIP clients not only have a UAC part, but also a UAS. This means that the client would also be able to listen for messages on a well known port specified by the application. This being the case, being able to give your PH a port to listen on as a UAS, would be very useful. When the UAS is instantiated, it could run in a separate thread and report received requests to the test case. This being the case, the PH for a UAS would also need to parse request messages and build response messages in the same manner.

In summary, you'd likely require a UAC version of the PH which looks like your current implementation for HTTP and another implementation of a UAS so that reception of a message is possible, not just sending.

I'll be on the lookout for a web site which talks about these implementations that I can pass along although I image there are a number of examples on www.sipforum.org.

HTTP protocol handler using Apache HTTP Client (1 Day)

TestMaker TOOL is an extensible library of protocol handlers. The HTTP protocol handler uses the Java `HttpURLConnection` (<http://jakarta.apache.org/commons/httpclient>) and `JCookie` (<http://jcookie.sourceforge.net>) libraries to manage connections and sessions. TestMaker 4 users encounter the following problems with the current solution:

- TestMaker users connecting to hosts in non-Top Level Domains (TLDs) cause `jCookie` to throw an exception. The IETF RFC 2965 specification (<http://www.ietf.org/rfc/rfc2965.txt>) does not defined TLDs other than the top level of `.com`, `.org`, etc. `jCookie` rigorously implements the IETF specification. The Apache HTTP Client implements cookie handling,
- TestMaker users have no easy way to use Windows security protocols to do HTTP authentication.
- HTTP redirect functions are not uniformly implemented in browsers. TestMaker's `HTTPProtocol` class implements a simple browser emulation mechanism for redirects that leaves much to be desired.

The Apache `HTTPClient` library implements its own cookie management, `.NET` authentication capabilities, and redirect commands. Apache `HTTPClient` also has a much larger base of contributing engineers to fix bugs and maintain the code.

The new `http.apache` protocol handler does not replace the existing HTTP protocol handler, it augments it.

SOAP protocol handler using Apache SOAP (1 Day)

TestMaker TOOL provides the SOAP protocol handler to communicate with SOAP-based Web Services. The default SOAP protocol handler uses the Apache SOAP library. Apache SOAP is no longer supported and the Apache Axis library provides bug fixes, content handlers for attachments, and functions not available in the Apache SOAP library.

The new `soap.apache` protocol handler does not replace the existing SOAP protocol handler, it augments it.

Operations and Deployment Improvements (7 days)

New logger, Signals to start/stop (1 day)

A ticket system will make it possible for running tests to output log data and for the master to close the ticket so logging past a test case period is not possible. The new logger will augment, instead of replace, the simplelogger class.

PTTMonitor Windows Service wrapper (1 day)

The CPU, Network, Memory resource monitor utility enables TestMaker to report on resource utilization while a test operates. The monitor needs to be wrapped in a Windows service utility.

Modify J keyboard mapping for Mac OS X (2 day)

TestMaker uses a branch of the j editor. Unfortunately j does not support the Apple Mac OS X Command key. See Apple Technical Note TN2042 and <http://developer.apple.com/documentation/Java/Conceptual/Java14Development/index.html> for details.

GPL Licence and Copyright Notice In Files

All source code (Java, Jython, build scripts, and properties) files must have the following GPL license:

TestMaker is a utility and framework for testing Service Oriented Architecture (SOA) and Web Services for scalability, performance, and reliability.

For more info check <http://www.pushtotest.com> or send email to info@pushtotest.com

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details

TestMaker comes with a copy of the GNU General Public License in the docs/license.html file; if you cannot find the license, then write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Please direct questions regarding this license agreement to PushToTest at 2248 Montezuma Drive, Campbell, California 95008 USA, phone: (408) 374-7426.

PushToTest and TestMaker is a trademark of Frank Cohen.

All documentation files must have the following copyright notice:

(c) 2006 PushToTest. All rights reserved.

Classpath configuration utility (1 day)

A new TestMaker Tools -> Classpath drop-down menu opens a graphical interface to change the classpath TestMaker uses upon start-up.

Help system opens real browser (1 day)

The TestMaker 5 Help menu commands use the Java 1.5 desktop browser API to open a browser to display help files.

TestMaker.bat and .sh shell scripts launch a local TestNode 1

Users start TestMaker 5 by running the TestMaker.bat (Windows) or TestMaker.sh (Unix) shell scripts. The new scripts automatically start a local TestNode on port 8091.

5. Bug Fixes

Document Encoded SOAP message with HTTP Authorization. I found that with document encoded the Http Header doesn't have the Authorization element, so method setUsername and setPassword was useless. Looking at com.pushtotest.tool.protocolhandler.SOAPProtocol source, I found that the soapDocCall method doesn't assign any SOAPtransport object to the org.apache.soap.messaging.Message() used to call the WebService nor reuse the SOAPTransport created in the connect() method. I don't know if it was done on purpose or if it's a bug, but to make it work with my project I had to add the line msg.setSOAPTransport(call.getSOAPTransport()); before sending the message in the soapDocCall method.

6. Additional Ideas for TestMaker

When time permits, the following new features may be added to TestMaker 5 or greater.

- Wrapper class for the logs that parsed them and supplied things like a list of errors, error counts and other statistics. (Suggested by Bongos Amigos.)
- Register a mime handler to TestMaker runtime. TestMaker invokes the handler to get the content and invoke custom implementation to handle the mime type. For instance, a test runs a decryption function to make sure that the encrypted binary response is not corrupted, or runs an MD5 checksum to accomplish a quick check for binary corruption.
- Service Monitor Systems (SMS)
- TestMaker .NET
- TestMaker Plug-in for Eclipse

7. Appendix A: TestScenario.XML Example Document

```
<?xml version="1.0" encoding="UTF-8"?>
<testscenario version="2">
  <basics>
    <name>Geospatial Test</name>
    <defaultdirectory>./geospatialtest</defaultdirectory>
  </basics>

  <!-- Where to run the test -->
  <testnodes>
    <node name="localhost" location="http://localhost:8080/TestNetwork/TestNode" />
  </testnodes>

  <!-- Optional arguments and parameters -->
  <arguments>
    <argument name="targethost" value="10.0.0.1/gis"/>
    <argument name="id" value="admin"/>
    <argument name="password" value="admin"/>
  </arguments>

  <!-- Dynamically loaded jar files and other resources -->
  <resources>
    <jar path="./lib/test.jar"/>
    <jar path="./lib/xmlbeans2/xmlbeans.jar"/>
  </resources>

  <!-- Test case definition -->
  <testusecase>

    <dimensions>
      <!-- The count of sequences the test operates concurrently -->
      <crlevels>
        <crlevel start="1" multiple="3" steps="5"/>
        <crlevel value="2"/>
      </crlevels>

      <!-- The message sizes -->
      <messagesizes>
        <messagesize start="1" multiple="3" steps="5"/>
        <messagesize value="1" />
        <messagesize value="3" />
        <messagesize value="4" />
        <messagesize value="8" />
      </messagesizes>

      <!-- Also room here for a databasesize dimension, if time permits -->

    </testusecases>

  <testusecase name="simpletest">
    <setup name="geo_insert" testclass="com.geo" method="geosetup" langtype="java"/>
    <test>
      <run name="geo_insert" testclass="" method="" langtype="java"/>
    </test>
  </testusecase>
</testscenario>
```

```
<teardown name="geo_insert" testclass="com.geo" method="geoteardown" langtype="java"/>
</testusecase>

</testusecases>

</dimensions>

</testusecase>

<logs>
  <log type="file" path="logs/mylog.txt" level="2"/>
</logs>

<!-- Results analysis options -->
<tally enabled="true" destination="./results" archive="true" basefilename="gisresults_" />

<!-- Display results in charts -->
<chart liveresults="true" />

<!-- Monitor CPU, Network, Memory utilization during tests -->
<monitor enabled="true" />

<options>
  <sleeptime>0</sleeptime>
  <delayBetweenStartingUseCaases time="10" />
  <delayBetweenTestCases time="60" />
</options>

<notifications>
  <email on="start" email="fcohen@pushtotest.com" />
  <email on="end" email="fcohen@pushtotest.com" />
  <email on="interrupted" email="fcohen@pushtotest.com" />
  <email on="testcasesstart" email="fcohen@pushtotest.com" />
  <email on="hourly" email="fcohen@pushtotest.com" />
</notifications>

</testscenario>
```