Proposals/EnhancedTestReports

Enhanced Test Reporting

This page looks at ideas for enhancing the output of the JUnit test reports that we generate in <junit> and turn into HTML with <junitreport>

Goal: *an evolutionary enhancement of the XML files to keep up with changes in test frameworks and development processes*

Strengths of current format

1. Ubiquitous
2. Scales well to lots of JUnit tests on a single process
3. Integrates with CI servers
4. Produced by lots of tools : <junit>, antunit, maven surefire, testng
5. Consumed by: <junitreport>, maven surefire-reports, CruiseControl, Luntbuild, Bamboo, Hudson, IntelliJ TeamCity, AntHill, Parabuild, JUnit PDF Report
6. Includes log output, JVM properties, test names
7. Reasonable HTML security (we sanitise all output but test names in the HTML)

Limitations of current format

- Storing summary information as root node attributes prevents us from streaming output
- JVM crashes mid-test result in empty XML files; no opportunity for post mortem
- No metadata about tests stored other than package/name, which we assume is always java package format
    - The list of test cases is flat, even though the test run may have had a complex hierarchy of suites bug 24106
    - No record of test parameters (data-driven testing); the same test appears 7 times, like testFoo[0], testFoo[1], testFoo[2] (even if parameters are strings).
- No machine information recorded other than JVM properties, hostname and wall time.
- No direct integration with issue tracking systems (of which bug is this a regression test, who to page)
- No notion of skipped tests, timed out, other failure modes. bug 43969 bug 35634
- Output is logged but log4j/commons-logging/java-util log output is not split up into separate events
- Output is logged by testCase, link with executed test method is difficult to make
- Output from a single process is logged; no good for multi-process/multi-host testing
- XSLT transform uses too much memory for Java5 XSLTC engine unless you add more memory with -Xmx.
- Normal transformed layout doesnt work for running the same test across many machines/configurations.
- There could be more datamining opportunities if more system state was recorded (e.g. detect which platforms/configurations trigger test failure)
- stack traces saved as simple text, not detailed file/line data with the ability to span languages
- No way to attach artifacts such as VMWare images to test results
- Only one failure is allowed per test case. (JUnit requires there be only one failure per test case, but other test frameworks e.g. Selenium allow for the possibility of multiple failures per test case.)

Summary: it was good at the time, but as testing has got more advanced, we need to evolve the format (carefully)

Radical Alternatives

Here are some fairly radical alternate designs that SteveLoughran has played with in the context of running tests under SmartFrog.

- Standard serializable Java types for tests. Must include log entries and exceptions. These can be marshalled over RMI or serialised to text files. Enables a tight coupling of reporting across processes. It is however, hard to maintain stability, especially with OSS code that can be changed by anyone. The limit of these type's use would probably be the junit test runner and ant itself, both from the same point release of Ant.
- Streamed XHTML with standard class names. Here an inline CSS sheet provides the styles, and tests are streamed to disk as marked up div /span clauses. Does not directly scale well to presenting very large test runs; postprocessing is required. XSL can still generate alternate reports, though the XPath patterns are more complex //div[@class="error"] instead of //error.
- Atom. Here every test result would be presented as an Atom entry, possibly using streamed XHTML as above. Enables remote systems to poll for changes, and for browsers to present results as is. Does not directly scale well to presenting very large test runs; postprocessing is required. XPath is even more complex.
- Perl Test format : TAP is a text only format; very simple.

Evolutionary Alternatives

- add the key missing features from the reports: metadata, skipped tests, and scope for logging more machine configuration data
- add a placeholder for test runners to add their own stuff
- add stack trace data in a more structured form when tests fail on java5+
- base metadata to include:

    - test parameters (data-driven @Parameters)
    - machine info
    - test description
    - links
    - issue IDs (for binding to issue tracking systems)
- *add new features in streaming friendly manner*
- *add handling for JVM crashes* Stream results to a different process for logging, so that JVM crash truncates the log instead of killing it. Or just save the (incomplete) DOM on a regular basis. Add a flag to note that the test is incomplete (maybe an element saying "starting"+test name.
- *make it easy to integrate* Numerous tools already support the old format; ideally they should not need to write a whole bunch of new code or understand a lot of radically new concepts in order to support the new format.
- create a root tag that might span several testsuites. So that one report may have contain information on many testsuites.

Interested Parties

Add the names of people actively participating

- Ant team (ant-dev) (SteveLoughran to do the prototyping for junit reporting)
- Ant team for antunit support: DavidJackman
- SmartFrog team (SteveLoughran)
- TestNG: AlexandruPopescu
- Maven SureFire: DanFabulich, BrettPorter
- Cactus framework: Petar Tahchiev

Plan and Timetable

For Ant we'd be looking at ant1.8.0 and any 1.7.2 release for these changes; gives us time to test with all the CI servers. We could do a junit4 test runner with early output. Testng are on a faster release cycle.

1. Start wiki, discussions on various lists (ant-user, testng-user)
2. collect test data: XML files from all the tools that currently generate the format
3. define first relaxng schema
4. build functional test systems: Public EC2 image with all the OSS CI tools installed; Private VMWare images with Bamboo, TeamCity.
5. add new <junit> reporter, test its output works with existing consumers
6. add antunit support (pull out description; if/unless => Skipped)
7. evolve XSL stylesheets
8. add junit4 test runner. Let TestNG, SmartFrog do their prototyping
9. ship!

Ideas

Improved Logging information

- Retain the current stdout+stderr logs, but also allow people to bind to custom log4j/commons-logging/java.util.logging back ends that grab the raw events and log them as structured XML events (host,process,thread,level,timestamp,text). These would be renderable at different levels. This is not something we'd do automatically as it can change application behaviour. We'd have to provide the custom back-ends for the loggers and offer a switch to turn this on in the task; the switch would set the properties for the forked process (and it would have to be forked) to log through our system. There's an

example in the smartfrog repository to capture log entries and accompanyingexceptions that can be captured serialized and shared between processes.

- XSL style sheets to merge output, ordering by clock (received time) and displaying log messages from different machines/levels in different colour
- easy turn on/off display of different levels in viewers; maybe the test runner itself, though log4j.properties and similar can do this (actually, given that java.util logging is such a PITA to set up, we could add helper operations in the <java> task for all to use.

Improved Host information

As well as logging the jvm properties, grab other bits of system state

- wall time and time zone
- Ant-determined OS (using standard names)
- environment variables on java5+
- Java5 JMX info on JVM state pre- and post- run -eg, memory footprint of every test.

The risk here is that we could massively increase XML file size by duplicating all this stuff for every test. We may only need some stuff per test suite.

Improved Fault Information

Java faults should be grabbed and their (recursive) stack traces extracted into something that can be zoomed in on

```
{{{<throwable classname="java.lang.Throwable">

        <message>text here</message> <stack>

                <stackrow>org.example.something.Classlet:53</stackrow> <stackrow>org.example.sp2.Main:36</stackrow>

        </stack> <nested>

                <throwable classname="java.remote.RmiException">

                        ..

                </throwable>

        </nested>

</throwable>}}}
```

This would be better placed for display on demand operations; other postmortem activities. We should still render the fault in the existing format for IDEs to handle (e.g. IDEA's "analyze stack trace" feature)

DanFabulich: Many popular tools parse exceptions directly; they're already reasonably structured, even if they aren't really XML.

SteveLoughran: True. But I'm thinking about better XSL support. Also the standard exception dumps don't scale well to deeply chained exceptions; gets hard to see what is going on.

Open ended failure categories

success and skipped are fixed, but there are many forms of failure (error, failure, timeout, out of performance bounds, and others that depend on the framework itself) Can we enumerate all possible categories for failure, or could we make it extensible?

DanFabulich: I recommend against attempting to enumerate failure categories. Allowing a user-defined failure type (with user-defined semantics) makes more sense to me.

MattDoar: I like the idea of being able to define my own test states, but I would like to see perhaps half a dozen common ones predefined. Perhaps:

- Pass
- Fail
- Error - unable to determine pass or fail, which includes timeout
- Skipped
- In Progress, aka not-yet-finished.
- Not yet implemented. Junitour - http://junitour.sourceforge.net has this notion; you can stub tests out and they do not get misinterpreted as actually working. Example Report: http://junitour.sourceforge.net/junitour/junit-noframes.html. Any tests that implements the UnitTestIncomplete int erface qualifies as an incomplete test.

SteveLoughran: I like failure itself to be categorised. So every state can be a pass state, a warning or a failure. Warnings could include "passing, but took 50% longer than usual". Failures can include "tests passed but outside allowed time" "tests passed but memory consumption was over the limit", as well as simple "some assertion failed". Maybe we decouple result name from outcome, with a limited set of outcomes, but an unlimited set of outcome categories.

Add partial-failure, not-yet-finished as test outcomes

- not-yet-finished is for tests in progress; makes sense if you are doing a live view of the system on a continuously updated web site.
- partial-failure is when you run tests on lots of machines in the style of grid unit, and some tests fail on some of the hosts, but not others. These are the troublesome tests, as they are hardest to replicate. It's an interesting out come as it is not one you can infer from a single test -its the result of aggregating the same test over different machines

Extend the viewer

- for every test, test case, show the description. Should descriptions be plain text only? Needs to support UTF8 at least.
- list skipped tests
- list tests by tag
- list tests by execution time
- list tests that are partial failures, unfinished

- might be a good idea to have a 'sort-by-column' feature. This way you sort columns by test-name, execution-time, test-status, etc. (Petar Tahchiev)
- for tests that are published to a web site, integrate with searching (allow a search box in the HTML)
- list failing tests by exception name
- For live output, provide an ongoing status page that updates regularly.
- allow test results to include links to images, videos that are then embedded inline. Your test runs, on youtube!

Publish under atom

- Allow an individual test to be served as a single Atom entry. This may require xmlns games. Perhaps we could define an optional xmlns for test results; if served through atom this would be required.

test results as wire format

Could we use a test result as a wire format for exchanging the outcome of a single test between processes/hosts? Rather than just something for XSL and other reporting tools to handle.

1. Could glue together bits of junit+Ant for cross process handling
2. Could integrate test runners + CI tools.
3. Could provide a library to aid developers to do this

Arguments against

1. if it becomes a wire format, it becomes even less flexible
2. CI tools are better off with their own custom listeners (they get to stop the run too blocked URL
3. We don't want to endorse JAXB (brittle) or DOM (painful), but XOM is off limits to apache projects (LGPL), and we don't want to intro new dependencies to ant either.

This may be something for things other than Ant to consider.

Add support for artifacts

- Formalise the the idea that every test run/test case can include artifacts off the results.
- artifacts to include inline SVG content and linked images
- other artifacts would be specified by mime type, URL and description
- viewers/atom feeds could embed SVG content and images, other content would be turned into links.
- relative links are best for publishing reports that are HTML viewed

Integration with Issue Trackers

What would this look like? Perhaps and element with URLs to related issues?

Petar Tahchiev: This could be easily integrated with Maven, since Maven keeps track of the Issue Management System in the POM. This way we might only keep the ISSUE # in the xml and the maven surefire-reports plugin can construct the URLs.

Integration with SCM

I would really like the idea of polling some info from the SCM, like for instance the user, who added the test-case(or the test-method). This way if a test is failing and you can't realize why is this happening, you can consult with the guy who wrote the test-case.

Generate Google Charts

The Google Charts API dynamically creates a PNG image from an appropriate URL. We could use an XSL transform to create a result image showing

- A pie chart of outcomes
- history charting

History analysis really needs full history logging and is something that some CI tools already do; its an analysis service, not something the XML/XSL files can do.

Examples of existing tests

multi-site tests

http://people.apache.org/~stevel/cddlm/interop/ current presentation of multi-host interop tests]. This directory contains the results of running the same client against three SOAP endpoints, implementing the set of tests agreed by a standard's working group.

1. Some of the tests fail 'unimplemented' meaning they haven't been unimplemented; these should really be warnings or skipped tests whose absence is noted, and which all would have the tag 'unimplemented'.
2. The data was collected on a single machine, 3 separate JVMs each with a different endpoint property set to test against a different remote endpoint.
3. The tests throw faults whose toString() method return the SOAPFault. This is being embedded into the error text unescaped. For example, in one test the output appears as

```
ns1:Server
Server Error

Server
Error at org.apache.axis.handlers.soap.SOAPService.invoke(SOAPService.java:473)
```

If you look at the source, it is more informative, showing that the XSL didnt strip the ?xml declaration, or escape any XML

```
<code>
 <?xml version="1.0" encoding="UTF-8"?><br/>
 <Fault xmlns="http://schemas.xmlsoap.org/soap/envelope/"><br/>
  <faultcode xmlns:ns1="http://xml.apache.org/axis/" xmlns="">ns1:Server</faultcode><br/>
  <faultstring xmlns="">Server Error</faultstring><br/>
  <detail xmlns=""><br/>
    <stackTrace xmlns:ns2="http://xml.apache.org/axis/" xmlns="http://xml.apache.org/axis/">Server<br/>
      Error at org.apache.axis.handlers.soap.SOAPService.invoke(SOAPService.java:473)<br/>
      at org.apache.axis.server.AxisServer.invoke(AxisServer.java:281)<br/>
      at org.apache.axis.transport.http.AxisServlet.doPost(AxisServlet.java:699)<br/>
      at javax.servlet.http.HttpServlet.service(HttpServlet.java:709) <br/>
      at org.apache.axis.transport.http.AxisServletBase.service(AxisServletBase.java:327)<br/>
      at javax.servlet.http.HttpServlet.service(HttpServlet.java:802) <br/>
      at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:252)<br/>
      at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:173)<br/>
      at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:213)<br/>
      at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:178)<br/>
      at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:126)<br/>
      at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:105)<br/>
      at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:107)<br/>
      at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:148)<br/>
      at org.apache.coyote.http11.Http11Processor.process(Http11Processor.java:856)<br/>
      at org.apache.coyote.http11.Http11Protocol$Http11ConnectionHandler
         .processConnection(Http11Protocol.java:744)<br/>
      at org.apache.tomcat.util.net.PoolTcpEndpoint.processSocket(PoolTcpEndpoint.java:527)<br/>
      at org.apache.tomcat.util.net.LeaderFollowerWorkerThread.runIt(LeaderFollowerWorkerThread.java:80)<br/>
      at org.apache.tomcat.util.threads.ThreadPool$ControlRunnable.run(ThreadPool.java:684)<br/>
      at java.lang.Thread.run(Thread.java:595) </stackTrace><br/>
    <hostname xmlns:ns3="http://xml.apache.org/axis/" xmlns="http://xml.apache.org/axis/">cddlm</hostname><br/>
  </detail><br/>
</Fault><br/>
<br/><br/>
        at org.smartfrog.projects.alpine.transport.http.HttpTransmitter.transmit(HttpTransmitter.java:184)<br/>
        at org.smartfrog.projects.alpine.transport.Transmission.call(Transmission.java:128)<br/>
        at java.util.concurrent.FutureTask$Sync.innerRun(FutureTask.java:269)<br/>
        at java.util.concurrent.FutureTask.run(FutureTask.java:123)<br/>
        at org.smartfrog.projects.alpine.transport.DirectExecutor.execute(DirectExecutor.java:32)<br/>
        at org.smartfrog.projects.alpine.transport.TransmitQueue.transmit(TransmitQueue.java:106)<br/>
        at org.smartfrog.projects.alpine.transport.Session.queue(Session.java:204)<br/>
        at org.smartfrog.services.deployapi.alpineclient.model.PortalSession.beginCreate(PortalSession.java:109)
<br/>
        at org.smartfrog.services.deployapi.alpineclient.model.PortalSession.create(PortalSession.java:138)<br/>
        at org.smartfrog.services.deployapi.test.system.alpine.deployapi.api.creation.
Api_05_system_create_destroy_Test.
            testCreateDestroySystem(Api_05_system_create_destroy_Test.java:39)<br/>
</code>
```

What we are seeing here, then, is a Java Exception that contains a SOAPFault raised by Axis1.x at the far end of an HTTP link, but most of this is lost in the textual report. While adding custom XSL for SOAPFaults is probably out (esp. because JAX-WS 2.0 always tries to hide them), we should present the returned XML as (a) it's informative and (b) what we do now is a security risk. What if an Exception.toString() generated HTML with script tags?

Examples of Extended tests

Examples of what a new Test Report could look like.

example 1

```
<testsuite total="1">
  <metadata>
    <description>Tests of OS-specific things</description>
    <xml>
     <rdf xmlns:rdf="http://w3c.org/rdf">
      <!-- random triples here -->
     </rdf>
    </xml>
  </metadata>

 <testcase name="testIgnorable">
   <metadata>
     <description>A test that only works on Win98</description>
     <links>
       <uri>http://svn.example.org/ex/4</uri>
     </links>
     <tags><tag>win98</tag><tag>windows</tag><tag>trouble</tag></tags>
     <issues>
       <id>JIRA-117</id>
       <id>JIRA-113</id>
     </issues>
    </metadata>
    <skipped>
     wrong OS
    </skipped>
  </testcase>

 <summary>
  <total>1</total>
  <skipped>1</skipped>
 </summary>

</testsuite>
```