Championing Test Automation At A New Team: The Challenges and Benefits

Alan Leung

AlanLeung@sierrasystems.com

Abstract

Testing System-to-System (S2S) messaging requires painstaking attention to detail and a level of technical skill beyond verifying most GUI interfaces. Regression testing requires the same disciplined approach despite being a tedious and time-consuming task. S2S messaging is an obvious candidate for test automation but automating any process takes another level of understanding, expertise, and effort.

The mature project that I joined was responsible for several SOA (Service-Oriented Architecture) interfaces involving SOAP-based and REST-based web services, XSL transforms, and an EMPI data source. With limited staff and time to test these interfaces, I wanted to promote automated testing as much as possible in order to ensure software quality with constrained resources.

While the test team was aware of the potential benefits of automation, they needed some initial assistance to begin their efforts. Over the course of approximately a year, my assistance to the test team decreased from frequent interactions until the team was relatively self-sufficient and taking the testing framework in ways I had never explored.

The result of the test team's work is that five of seven interfaces have fully automated testing and the number of test scenarios has increased by 305%. Considering one interface, testing that was not as complete and that took approximately 7 days to complete now takes 7 minutes to execute. Because of the result of the team's work, an automated regression test suite (rather than one-time manual efforts), their product has also helped other teams downstream of the system testing phase. The test team has also gained valuable skills that can be deployed on other projects and other IT problems.

Previously, everyone on the team knew automated testing would be ideal and the benefits great, but to reach that state, we had to overcome many obstacles. However, just as automating tests pays off with every subsequent cycle of testing, changing the team's mindset to accept automated testing as the preferred way to test improves overall productivity and velocity of the team.

Biography

Alan Leung is a senior consultant with Sierra Systems. He is an expert systems integrator with architecture skills that has been a leader on numerous initiatives. A former member of IBM Global Services, his engagements have included health (Alberta Health), education (Athabasca University), and financial industry (Bloomberg) clients in Canada and the USA. Having been a member of a troubled project, written poor code early in his career and code reviewed a lot of suspect source in years past, he is now a code quality evangelist and a best practices advocate. Previously, Alan has presented at Edmonton (Alberta) Java user group events and at an internal IBM conference (Blue Horizon). He is Sun Java Certified and ITIL certified.

1 Challenges of testing S2S messaging

Unlike testing web-based GUIs or other graphical user interfaces, testing System-to-System (S2S) messaging interfaces requires tooling as well as a high degree of technical competence. To test S2S messaging, the tester must simulate an external system sending messages in the correct protocol. Constructing a valid request with proper security tokens, message IDs, and timestamps is much more complicated than merely interacting with a web page or desktop window. There is little to no room for improperly constructed messages: Not copying the entire security token string obtained from a secondary web service, neglecting to include a unique message ID, or missing a double quotation mark when building a test message will result in an error, and often the error response is more suited to a developer than the typical test analyst. Nearly every single detail needs to be correct over hundreds of executions of submitting test messages. Quickly verifying that the SOA interface's response fits expectations is another problem, as the result is not formatted for human readability: for example, checking for the response code and returned error message in an XML blob. The key pieces of the response that need verification are buried amongst other XML clutter that lacks whitespace, unlike a web page.

Our project team develops and supports seven interfaces that consume and return patient demographic data, for a provincial health ministry. Our interfaces are on the receiving end of S2S messaging from Electronic Medical Record applications (software used in physician clinics), Pharmacy Management System applications (software used in pharmacies), and internal ministry applications that rely on our registry. Five of these interfaces are SOAP-based and REST-based web services, and XSL transforms and an Enterprise Master Person Index (EMPI) data source are some of the technologies involved in the mix. Manually testing these interfaces was barely manageable when there was only five interfaces; the testing schedule was often exceeded. With two additional interfaces, either more resources and/or an increased testing schedule was necessary, but I believed that it was necessary to attempt automation of the testing.

2 Tool Selection

SOA interface test automation requires good tooling and when choosing a testing tool (or for that matter, any software tool), we should consider the following checklist:

- Mature product - to be confident that most tool functionality is free of defects
- Feature-rich and powerful – to satisfy scope of testing and be able to do it in an automated manner
- Good support - in the (likely) event that questions/problems arise when using the tool. Accessing good support through third-party channels such as StackOverflow can occur if the product is widely used and/or is a Mature product
- Extensible – when there is no built-in functionality that aids with testing an aspect of an SOA interface

Following is a summary of several tools in use (SoapUI, PIN S2S Tool, RESTClient, other home-grown tools) or considered (SoapUI Pro) at our health ministry.

	SoapUI	SoapUI Pro	PIN S2S Tool	RESTClient	Other home-grown tools
Description	Publicly available but not commercially supported tool for testing SOA interfaces	Commercial tool for testing SOA interfaces	In-house tool used by primary testing group for verifying and data loading one of ministry's mission-critical applications	Publicly available (http://rest-client.googlecode.com/) tool for testing HTTP/REST services	In-house tools created by project development team primarily for unit testing
Product Maturity	Developed since 2008 by a small team[1], at version 4	At version 4	Developed since approximately mid-2000s	Developed since 2008 by 1-2 developers[2], at version 3	The various tools have been developed and maintained by one developer over the course of approximately 4 years. User base is limited to the project team so the tools have not been tested as extensively as a widely available tool

[1] http://www.ohloh.net/p/soapui

[2] http://www.ohloh.net/p/freshmeat_restclient

	SoapUI	SoapUI Pro	PIN S2S Tool	RESTClient	Other home-grown tools
Features	Can construct requests for a variety of protocols (e.g. HTTP, REST, SOAP) and test cases can be configured with assertions for test automation	Features in addition to those offered by SoapUI (http://www.soapui.org/About-SoapUI/compare-soapui-and-soapui-pro.html)	Many features but primary goal is to support testing one particular application. Test cases can be configured with assertions for test automation	Limited to testing HTTP/REST services. Test cases can be configured with assertions for test automation, but built-in support is extremely limited such that a lot of custom code is necessary	Each tool is able to send messages for a specific protocol and sometimes limited to a specific SOA interface. Test scenarios cannot be configured with assertions for test automation
Support	Wide user community online, able to find documentation on SmartBear's web site as well as forums, blogs, etc.	In addition to support as per SoapUI, the Pro edition is commercially supported	Developers of tool have departed project	Documentation and user community is limited	Support is on a best-effort basis but communication of issues is easy
Extensibility	Test analysts can augment test automation at several points in a test lifecycle (test initialization, as explicit operations during test execution, during verification of the response, and test completion) by adding Groovy (a concise Java-compatible dynamic language) code. The SoapUI framework also has documented extension points that can be taken advantage of with Java code.		None, as far as known	Possible but built-in support is extremely limited such that a lot of custom code is necessary	Not possible by test analysts
Price	Free	$399 USD/year/license	No cost as in-house tool	Free	No cost as in-house tool

2.1 Free edition versus Pro edition

When I joined the project, SoapUI was the chosen testing tool, but was not being used to its full capability. The question is whether SoapUI Pro would have been a better choice. One of the practices recommended by Joel Spolsky (http://www.joelonsoftware.com/articles/fog0000000043.html) is to "use the best tools money can buy". However, due to policy issues, our project was never afforded the use of SoapUI Pro (http://www.soapui.org/About-SoapUI/compare-soapui-and-soapui-pro.html), but SoapUI Pro may be the better choice for a number of reasons:

- Ease of use, especially important for non-hardcore developers starting on the test automation journey. For example, "Coding Free Test Assertion", the ability to construct simple XPath assertions using a GUI rather than understanding XPath expressions.

- Less time developing scripts already available with the Pro edition ("Scripting Libraries"). The testers estimate 25-50% of scripting necessary for their testing is available in SoapUI Pro (based on looking at SmartBear web site documentation that indicates a feature is only available in the Pro edition). When comparing the Pro edition license cost versus the time savings for the development and test teams, it is evident why Joel Spolsky advocates spending money for the best tools.

- Team support so that test analysts can share a common SoapUI project file if testing the same SOA interface (http://www.soapui.org/Working-with-Projects/team-testing-support.html). Our testers resorted to having their own individual SoapUI project files per interface. This meant duplication of common scripts and properties which can be an issue for maintainability of the testing artifacts.

Note that SoapUI Pro would not have alleviated the need for some of the custom extensions made to SoapUI which were very particular to the SOA interfaces that we were testing or the applications that we were leveraging for verification. Also, one advantage of using the free SoapUI edition is that there were no issues transitioning SoapUI Projects to other teams that were not using the Pro edition as well.

Other commercial tools in the same space (SOA interface testing) include ITKO LISA and Rational Tester for SOA Quality but neither of these products was evaluated.

2.2 Piloting the tool

In order to determine whether SoapUI had the facilities to automatically test our SOA interfaces, I read the available online documentation from SmartBear and other web sites, but more importantly, I experimented with the tool by building a prototype against Google Maps REST APIs. Once I was relatively familiar with SoapUI's parameter substitution, assertion, and project organization capabilities, I was able to create an example for one of our interfaces, Active Query, to verify that SoapUI could be used on our project.

Due to commitments with primary tasks, there was little time allocated for one-on-one training. Instead my approach was to construct working examples of test cases against the Active Query interface, allowing the test team to experiment with SoapUI and the custom test automation framework. Having said this, it is important to note that not any tester can handle this approach. Testers without a development background would have had tremendous difficulty without constant one-on-one assistance. But the tester analysts on our team have a technical, development background, e.g., familiarity with SQL, relational databases, and procedural programming. While automating tests is not at the level of developing the application, "testing-scale" development/programming still requires many skills for which there existed gaps. How these gaps were addressed follows.

2.3 Custom framework

The customizations built on top of SoapUI consist of SoapUI Test Cases that can be executed from other Test Cases using the “Run Test Case” Test Step feature. This Test Suite of reusable Test Cases act as a library of subroutines heavily used for testing our SOA interfaces. The parameters and return values are transferred via Project-level Properties:

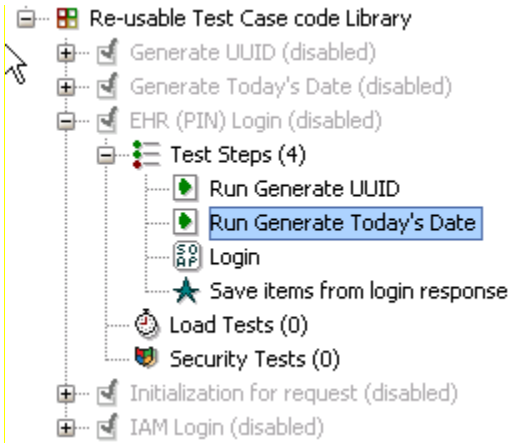

Figure 1: Reusable SoapUI Test Cases

For example, the above figure shows the library of subroutines, implemented as a SoapUI TestSuite. The subroutines are TestCases that are all disabled as the TestSuite is not executed directly. To use this library, for example, the test analysts will include in their TestCases a “Run Test Case” Test Step to execute the “EHR (PIN) Login” TestCase. This TestCase will in turn execute other TestCases or TestSteps. For example, the “Run Generate UUID” TestStep will execute the “Generate UUID” TestCase. This TestCase contains a Groovy TestStep which executes some Groovy language (a concise Java-compatible dynamic language) code. The code modifies Project-level Properties as a way of “returning values” to the caller.

The second aspect of this SoapUI-based framework is Java code extensions that are either called directly from Groovy Test Steps or automatically executed by the SoapUI engine (http://www.soapui.org/Scripting-Properties/custom-event-handlers.html). Our testing eventually required three custom event handlers which I developed due to their complexity. Below are screen-shots of the beginning of the source code for each handler. The code comment describes the handler and the class from which the custom handler derives illustrates how it is implemented.

```
/**
 * Logs result of test case execution to a file for debugging or auditing purposes.
 * If the name of Test Suite of the test case contains "Re-usable Test Case code", the test case run is not recorded
 */
public class AfterRunResultsLogger extends TestRunListenerAdapter {
```

Figure 2: AfterRunResultsLogger event handler

```
/**
 * Enables PD GUI testing within soapUI by supporting the back button trap implemented by PD GUI.
 *
 * @author alan.leung
 */
public class PDGuiBackButtonTrapSupportListener implements TestRunListener {
```

Figure 3: PDGuiBackButtonTrapSupportListener event handler

```
/**
 * For applicable TestSteps in a soapUI Project testing the PCR EMR Passive Messaging interface, will abort the TestCase
 * if a potential duplicate Message ID is detected (by querying the PCR database directly).
 * <p />
 * It expects the Project-level property "UUID1" to be used as the Message ID for a HL7v3 message.
 *
 * @author alan.leung
 */
public class AbortDuplicateMessageIDSubmitListener implements SubmitListener {
```

Figure 4: AbortDuplicateMessageIDSubmitListener event handler

Having this mainly unobtrusive framework in place meant that the test analysts had less work to construct new test scripts within SoapUI. That is, SoapUI automatically calls the handler code at defined points in a test execution life-cycle and therefore the test analyst does not need to explicitly add anything to their scripts. Instead, "it just works". Secondly, we reduced duplication of Groovy test code by moving such logic into a Java Archive (JAR) file and then having Groovy "one-liners" make calls to the Java library.

Unfortunately the "Re-usable Test Case code Library" SoapUI Test Suite needs to be present in each SoapUI Project, so the Test Suite can become inconsistent between SoapUI Project files if its contents are modified by the testers. This has not caused any issues yet but theoretically can cause the same problems as any duplication of code: code fixes or code enhancements in one copy need a way to be propagated and merged into other copies. On the other hand, the Java code extensions stored in a .JAR file is consistent across SoapUI Projects because it is shared across all Projects in a SoapUI application installation.

3 Process of championing test automation

Initially the test analysts had reservations about the degree to which to automate the tests: How much time would it take, how usable was this framework, how many roadblocks would be faced and could the roadblocks be overcome? As with any procedure, a test case can be broken down into many steps. Some steps could be automated and some could be performed manually. My push was for complete hands-off automation but at the beginning, because of the high learning curve, there was some trepidation on how much could truly be automated. In the face of the uncharted territory and frankly, the large amount of upfront work necessary to automate tests, it was easy to consider falling back on manual one-off steps in certain situations.

3.1 Tasks in testing SOA interfaces to automate

There are many tasks involved in testing SOA interfaces and therefore many tasks to consider automating. At a high-level these tasks are defined below and our particular context is described:

1. **Use full capabilities of testing tool(s)** - in our case, the primary tool is SoapUI, and facilities it offered (that is available in the best testing tools) included:

    - Parameter substitution into the web service requests, automatic assertions, and endpoints per environment
    - Reading from data pool, e.g. from properties file or a relational database
    - Repeatable test execution - construction of test steps so that data is set up, test is executed, and clean-up of execution results occurs

2. **Construct valid web service requests** – our tested applications communicate with XML so it is important to understand what is valid XML. In particular, our health IT domain uses the HL7v3 messaging specification, which is XML-based with its own namespaces and XSD schemas

3. **Generate/manipulate test data** – SoapUI supports user-developed Groovy scripts to construct sections of the web service request based on reading information from a data pool, for example

4. **Interpret error messages returned by the application** – some of our SOA interfaces lie behind a Health Information Access Layer (HIAL), essentially an Enterprise Service Bus (ESB), which can return XSD schema validation errors. Often these error message are cryptic. For poorly constructed requests, the HIAL sometimes returns misleading or unfathomable messages

5. **Validate that web service responses match expectations** – SoapUI supports, among others, XPath expression assertions

6. **Validate that request to web service was actioned** - for some S2S requests, we could make a second request to confirm the execution of the first request. And although SoapUI has the JDBC Test Step to execute SQL against a relational database, our system uses a specialized data source, the Initiate EMPI; queries against the underlying relational database is generally frowned upon. However, often the most tell-tale way to verify the web service performed correctly is to use SoapUI to execute HTTP requests against Java Servlets that form the Initate EMPI end-user interface. The resulting web page can validate that, e.g., an Add request was stored into the EMPI.

7. **Logging of test efforts** - to automatically record evidence of testing and the results. The test analysts need to sign off the application before it can proceed to the next environment (i.e. “system test” to “user acceptance test” environments) and the logged test results provide evidence that the application performed properly, from an interface consumer’s perspective, when it was tested. The logs can also be used to show how the application behaved historically.

Sometimes a background in related technologies confused matters. For example, Selenium IDE behaves differently than HTTP testing with SoapUI. For another example, Oracle PL/SQL date function syntax differs from Groovy date function calls.

3.2 Building the confidence and ability to automate tests in the test analysts

For each of the tasks in the previous section, the test analysts needed to learn and become comfortable with constructing test scripts to perform the task in an automated way. On the other hand, we wanted to minimize the amount of assistance necessary from the development team, since this was not time allocated in project plans.

Following is a description of the particular process used to transfer knowledge for each of the tasks previously described:

1. **Use full capabilities of testing tool(s)** - testers started from an example SoapUI Project I carefully defined. I also authored a dozen-slide PowerPoint deck explaining the major intricacies of the Project and how to go forward. From there, the testers could conduct online searches for more information now that they knew what keywords and phrases to search for

2. **Construct valid web service requests** - for XML, the testers easily learned by example. However, for the specialized HL7v3 message specification, I put together a hand-out summarizing the scope of the specification that was important to our team, stepped through the official pan-Canadian Infoway documentation, and noted where to find relevant information.

3. **Generate/manipulate test data** - because of the testers' backgrounds, I did not need to give a Groovy programming tutorial. For more advanced constructs, the testers searched online for Groovy/Java examples and occasionally I fielded a question

4. **Interpret error messages returned by the application** - for schema validation error messages and HIAL error messages, both of which were only occasional, it was more difficult educating the testers on the reasons behind these messages because we did not catalogue these messages (i.e. recording exception text and cause). Thus, for some error messages, seeing them for the second time was like seeing them for the first time

5. **Validate that web service responses match expectations** - although my example SoapUI Project had some simple XPath expressions to perform assertions, I also referred the testers to online resources (http://msdn.microsoft.com/en-us/library/ms256086.aspx). However, for more complex assertion requirements, I needed to start off the testers with custom examples on a case-by-case basis

6. **Validate that request to web service was actioned** - our method of using SoapUI to behave like a web browser required some explanation to the testers so I authored a PowerPoint deck with diagrams to show the difference between a web browser, SoapUI, and Selenium, as well as how to log into the Initiate EMPI web application and use it to perform a search. The testers and I walked through an example SoapUI Project demonstrating this capability and then they had the PowerPoint slides as reference. A couple of weeks passed between the time of this demo and when this capability was tried. One test analyst was struggling to extend the example SoapUI project with his own verification steps. As I sat with him and spent a few minutes re-explaining how to use SoapUI like a web browser in an automated manner, switching between the PowerPoint and SoapUI application, he began to recall what we had walked through weeks earlier. The expression of understanding was gratifying.

   Obviously, the lesson is to deliver training in a just-in-time manner whenever possible, as even a carefully-authored PowerPoint reference may not be enough for a complex topic.

7. **Logging of test efforts** – in fact, this functionality was initially implemented as a Groovy script by one of the testers. But in order to alleviate the need to copy and paste this script into each Test Case, I extended SoapUI with a Java-based Event Handler that was more or less the original Groovy script but "Java-fied".

Much like automating tests, a large investment of developer time and effort was necessary upfront. For a couple of weeks, I estimate approximately two-thirds of my time was spent on a framework within SoapUI (organization of a SoapUI Project structure to support code re-use, Java extensions of SoapUI interfaces) and to a lesser extent, one-on-one assistance with each of the testers. This time commitment eased off thereafter with occasional bouts of activity with the framework or test analysts. After the initial investment of time and effort, the key to the testers' current self-sufficiency are:

- **Testers having a technical background** – giving them the confidence and experience to work towards automating the tests

- **Expanding on working examples** - because of their technical background, the test team could experiment with and expand on the working examples

- **Available online knowledge** - because SoapUI is an industry-wide tool with many users, and Groovy has many online resources, etc. the test team found many answers by doing an online search. However, the key is to establish a foundation in order to know what to search for

- **Just-in-time training sessions** - delivered from developer to test team for new testing capabilities that will be implemented by testers within days. These sessions might involve emailing the test analysts a sample SoapUI project demonstrating the new capability and walking through the sample as a group at someone's desk. Or it might involve booking a meeting room and going through a PowerPoint presentation together. However, the key is that as little time transpires between this education session and when the test analysts actually implement the new test capability in their work.

I've been on teams where the business analyst team, test team and development team are segregated by an invisible barrier: The development team is reluctant to clarify requirements with the business analyst team, thus incorrectly building aspects of the application; the test team doesn't ask the development team for assistance to test more challenging parts of the application, thus insufficiently testing the application or taking more time than necessary; and the business analyst team has not verified the test team's test scenarios for completeness, thus defects are not detected until much later.

While I offered assistance to the test team to automate their tests, I tread a fine line so as not to intrude on their day-to-day work. However, when I thought there were gaps in the test verification, e.g. using a simple but fallible Contains assertion versus a complicated XPath assertion, I intervened to introduce a failsafe assertion.

3.3 Train the trainer and sharing automated test scripts

In our software development lifecycle, testing takes place by the development team, then the system test team (the team I assisted), and followed by the user acceptance test team (which is sometimes the "business" team that gives direction on what applications to build and manages production operations). Armed with their knowledge of the testing framework, the system test team transitioned their SoapUI Projects to the UAT (User Acceptance Test) team as a basis for more test scenarios during the UAT phase.

The transition from the system test team to the UAT team highlights that a technical background is highly recommended to automate new test scenarios and to even work with existing automated test scripts. Sometimes the UAT team consisted of business subject matter experts who did not have a development background. Then the UAT team consulted the system test team for one-on-one assistance frequently.

Sometimes a UAT team consisting of business SMEs did not use their time to develop new automated test scripts but instead analyzed test results with a big picture understanding of the application within a larger overall system. Often this was valuable and only possible because the system test team delivered essentially an executable product, the SoapUI Projects, to the UAT team. Also, feedback from the UAT team sometimes augmented system test scripts.

4 Quantifiable results and subjective results

Was it worth the time and effort to convert the testing to be automated rather than manual? Our team firmly believes so:

- Considering one SOA interface, due to automation, we increased our test coverage by increasing our test scenarios from 110 to 480. Considering another interface, test scenarios increased from 88 to 125.

- Regression testing during infrastructure upgrade projects (DB2, Initiate EMPI, etc.) only takes 7 minutes of unattended test execution rather than approximately 7 days of tedious manual labor. For another interface, automated execution takes 111 minutes versus an estimated 475 minutes of manual execution.

- UAT team can re-use automated test scripts (although making changes for data available in UAT environment)

- Automated execution, when it needs to be extremely stringent as per S2S testing, eliminates typo type errors (after the test is established once) and does not inadvertently skip steps

- For ongoing support, the application is documented based on the behavior of tests. With automated tests, inquiries about the application are handled much quicker, and there is no need to rely on stale documentation (source code comments, requirements documentation, operation manuals, etc.). Some developers may say that the source code is truth, but it is not the human interpretation of source code that is truth, but rather the "ultimate semantics of a program is given by the **running** code" (http://programmer.97things.oreilly.com/wiki/index.php/Only_the_Code_Tells_the_Truth)

- Automating tests and the ease with which to create and execute more variations of a test scenario helps to update stale documentation to match the behavior of the application and makes requirements documentation more detailed

- Automated tests quickly detect defects that are often created when fixing another defect (http://brockreeve.com/post/2011/03/20/Bug-Fixing.aspx). Because of the intricacies of medium to large software systems, in the rush to correct an urgent defect, a developer can introduce a new defect while fixing the original one. Without a set of automated unit tests, these errors are overlooked and the application is deemed ready for the test analysts. But a comprehensive set of tests can quickly uncover these shortcomings. This improves the team's overall velocity (http://en.wikipedia.org/wiki/Velocity_%28software_development%29).

- When developing automated tests, by serendipity, sometimes defects are uncovered, and the half-developed test was not part of the official test scripts

With so many benefits to automating testing, it is easy to forget the initial hard work required to arrive at a state of high test automation. Therefore, management has to understand and buy-in to the up-front time/cost with automating testing, realizing the longer-term benefits. Some time prior to the official start of the testing phase may be needed to become familiarized with any new automated testing framework capabilities. Subsequent to the testing phase, we need to also consider maintenance efforts, to update the automated test suite/framework when the application's behavior changes or to re-factor the test harness for ease of future use (e.g. additions of test scenarios, less need to copy-and-paste).

Our team took months to learn the entire testing framework based around SoapUI. They acquired many skills during this time, including SoapUI, Groovy/Java, JDBC, XML, XPath, REST, and SOAP. However, for projects of a shorter duration (those measured in weeks), our approach would not be viable. Instead, either more capable tools are used or dedicated testing framework development and test analyst training time needs to be allocated.

5 Test automation Best Practices

Several best practices we learned from this experience:

1. **Automatically log test execution** - this alleviates the need for screen prints and records a history of how the application was behaving in the past

2. **Parameter substitution for uncertain aspects under development** – as an example, we parameterized two code values that required the Canada Infoway standards organization to approve. When the standards organization decided on other values than we were using ("MDATA" rather than "MERGEDATA"), it was straightforward to alter the test properties.

3. **Eliminate duplication** – for example, of Groovy scripts. SoapUI offers several methods to reduce duplication: the ability to call any code from a Java library (.jar) placed into its program extension directory (bin/ext) and execution of customized SoapUI Event Handler extensions. Also, testers should write more sophisticated Groovy script to eliminate simpler but copy-and-pasted Groovy script Test Steps. Duplication causes problems when for example, a defect needs to be fixed in each copy of the code

4. **Verify validity of assertions** - while it is satisfying to see a Test Case run "green" by passing all its assertions, it is imperative to double-check that the configured assertions only verify the expected result and do not pass a "failure" result. For example, a loose Contains assertion may pass even when a stricter XPath assertion fails (i.e. the matching string is appearing in an unexpected location in the response).

5. **Treat testing artifacts like application source code** – due to the complexity of automated test scripts, they can stop working unexpectedly and it is useful to compare with a known working version. Therefore testing artifacts should be stored in a version control system. A different branch of the testing artifacts should be created when application source code is branched.

6 Conclusion

For a test team accustomed to manual testing on the project, achieving test automation as the general mode of testing requires support from the development team as well as commitment from the test team. Obvious benefits are increased testing productivity and overall development velocity, but a less-obvious benefit is the valuable skills gained by the test team. However, management must realize that automating testing needs sufficient time to establish but the return on investment is realized with every execution of the test suites.

References

Black Duck Software, 2013. SoapUI open source code analysis, http://www.ohloh.net/p/soapui (accessed July 31, 2013).

subwiz (a rest-client author). 2013. rest-client Google Project Hosting web site, http://code.google.com/p/rest-client/ (accessed July 31, 2013).

Black Duck Software, 2013. rest-client open source code analysis, http://www.ohloh.net/p/freshmeat_restclient (accessed July 31, 2013).

SmartBear Software, 2013. "Compare SoapUI and SoapUI Pro", http://www.soapui.org/About-SoapUI/compare-soapui-and-soapui-pro.html (accessed July 31, 2013).

Spolsky, Joel, 2000. "The Joel Test: 12 Steps to Better Code," Joel on Software blog, entry posted August 9, 2000, http://www.joelonsoftware.com/articles/fog0000000043.html (accessed June 10, 2013).

SmartBear Software, 2013. "Team Testing Support", http://www.soapui.org/Working-with-Projects/team-testing-support.html (accessed July 14, 2013).

SmartBear Software, 2013. "Custom Event Handlers", http://www.soapui.org/Scripting-Properties/custom-event-handlers.html (accessed June 10, 2013).

Microsoft, 2013. "XPath Examples," Microsoft Developer Network, updated August 2, 2012, http://msdn.microsoft.com/en-us/library/ms256086.aspx (accessed June 10, 2013).

Sommerlad, Peter, 2008. "Only the Code Tells the Truth," 97 Things Every Programmer Should Know, entry updated October 16, 2008, http://programmer.97things.oreilly.com/wiki/index.php/Only_the_Code_Tells_the_Truth (accessed June 10, 2013).

Reeve, Brock, 2011. "Bug Fixing," blog entry posted March 20, 2011, http://brockreeve.com/post/2011/03/20/Bug-Fixing.aspx (accessed June 10, 2013).

Wikipedia, 2013. "Velocity (software development)," Wikipedia, entry updated March 6, 2013, http://en.wikipedia.org/wiki/Velocity_%28software_development%29 (accessed June 10, 2013).