IBM Software
Thought Leadership White Paper

July 2012

Lessons from testing service-oriented architectures

Eight issues to consider when testing service-oriented architectures

IBM

Executive summary

Leading companies now have over a decade of experience in testing complex service-oriented architectures and other distributed systems. Eight key lessons have emerged about how to avoid some of the most common testing pitfalls and provide recommendations on how to mitigate certain pervasive issues.

Distributed computing is not new and, as time has gone by, successful design patterns have emerged. The principle of considering an organization's systems as a collection of business services provides a useful platform on which to build new applications. These new applications are not dependent on any single underlying application, implementation or structure, thus protecting this new development from future changes. Examples of useful business services include "stock query," "place order," or "get account balance."

As each service may be used by many applications, it makes sense to test the service itself. However, this creates new challenges for the test team as these services do not naturally have an interface that can be operated by a human being. Instead, these services have been designed for invocation by other systems. Until recently, such invocation methods typically would be considered proprietary to an individual organization.

In the past, developers typically created test programs to demonstrate that *their* components worked. Test teams typically would be limited to reviewing these test programs and, in many cases, would be unable to test a new version of the component until the developer also created a new version of the test program. The validity of such tests could also be questioned, as often the developers would prove to themselves that their components interfaced correctly, with no guarantee of how well they might interoperate with the real system.

However, the popularity of open standards makes it more likely that technologies such as Extensible Markup Language (XML), Simple Object Access Protocol (SOAP), REpresentational State Transfer (REST), Hypertext Transfer Protocol (HTTP), web services and Java Message Service (JMS) will be employed to invoke services and provide responses.

The use of these open standards now makes it possible for test teams to learn core technical skills that can be applied to many different projects. As service-oriented architecture (SOA) becomes more widely used, test teams are able to add more value. This value comes not just from the fact that the testing of services is no longer a burden on development teams alone, because quality is everyone's responsibility. Testers can also ensure that the services created are reusable by others. After all, SOA is only worth implementing if reuse is not only expected but actively encouraged and made as easy as possible. For this reason, it is essential that service interfaces be verified independently to achieve the return on investment promised by SOA.

While SOA applications have been prevalent for quite some time, effective testing of these types of applications has remained elusive. Given that this area of testing is in its infancy, we have gathered together lessons from our customers and our consultants. The lessons contained herein apply not only to testing and quality professionals, but should also be read by project managers, developers and architects. Quality needs to be designed in and built in to a project. If it is simply added on at the end, the results will be much poorer.

Lesson 1 – Schema mismatches mean no communication and no integration

Schemas aid interoperability between services and their clients. They describe the interface that the service expects to be adhered to by the client. Schemas are generally designed up front, and documented, often by architects. The documentation of the schema is passed to the people on either side of the interface and this is used for both the client of the service and the implementer of the service. The seeds of disaster can be sown here. Although the architect has carefully documented the schema, this is not enough. The document can be misinterpreted. These misinterpretations can lead to massive project delays. Our experience is that projects might spend as much time in "integration" as they did in "unit development," primarily as a result of mismatches in schemas (see lesson #2).

The success of any kind of integration project, including SOA projects, depends on the clear and successful communication of schemas. The output of the design process must be a schema that can be used, without alteration, by the parties on either side of the interface. Luckily, the open standards employed in SOA projects, such as XML, XML Schema Definition (XSD), SOAP and Web Services Description Language (WSDL), make it more likely that a useable run-time deliverable can be output from a design-time tool. If you have the luxury of choosing your Unified Modeling Language (UML) design tools, include open standards on your list of requirements. It is still common for services to be built over legacy integration approaches such as fixed-width data structures, and these are particularly susceptible to these types of problems.

Lesson 2 – Schema changes restart project tasks

Changing a schema can have a disastrous impact on a deliverable. While it may look simple from the perspective of a designer remember that these mismatches are not only present in the interface, but permeate the component's data structures, program structure and code. Altering them once development has begun may have far-reaching consequences. Please note that if components have been designed properly, they can be immune to expected schema changes such as the addition of fields. This is one of the major benefits of XML. Unexpected fields are often ignored. Schema changes are considered to be changes to structure and field types. So how can you minimize the possibility of schema changes? Lesson 3 has a few examples.

Be aware that there are a great number of situations where a system *appears* to understand XML, and yet it does not. Instead, in early attempts to XML-enable legacy systems, parsers were written which only understood XML formatted in a specific way. Even when you are told a system understands XML, it is worth clarifying exactly *how* whitespace, element and attribute ordering are going to be handled.

Lesson 3 – Create examples early

As outlined in lesson 2, we want to avoid schema changes as they can cause task rework. A common cause of early schema changes in a project is a schema's lack of suitability for its intended purpose. How was this discovered? By trying to *use* it. It makes sense, therefore, to use a schema as early as possible in the process.

Ideally, examples of service invocations and corresponding responses, at least one for every situation in which the schema is likely to be used, should be created as soon as the schema is drafted. In this way, any design issues can be resolved before development begins. The example invocations and responses can also form the basis of the first test cases. Together, these requests and responses start to make up a physical implementation of a message (invocation) catalog, a key asset for encouraging service re-use as it shows examples of the service being used.

Lesson 4 – Components will be delivered late—test virtualization can help

SOA projects automatically involve dependence. Some of these dependencies will be owned or delivered by teams with their own projects and pressures. Even if a component is being developed as part of this project, you may need to test components that depend on it in isolation, or even before the new component has been completed. Emulating missing dependencies for the purposes of testing has been called *service virtualization* by some industry analysts, but in acknowledgment of the fact these emulations can use non-service based approaches, we prefer the term *test virtualization*.

In test virtualization, a real component is replaced by a virtual component, sometimes called a stub. Virtual components should be made available for key components to allow various scenarios to be simulated and tested more easily. If a tool is being used, the test team may well be able to create and manage these virtual components themselves. If key components are delivered late, the availability of virtual components ensures that problems in dependent components can still be detected, allowing the project to "flow around" blockages that have traditionally halted projects.

Although virtual components could be programmatically created, there are several reasons why this should be discouraged.

- First, it can encourage systematic errors.
- Second, it can be time consuming to use resources that could be better deployed solving the actual business problem.
- Finally, maintaining these virtual components can become a job in itself.

Tools specifically designed to provide virtual components help eliminate these issues and provide the same benefits with a minimum amount of effort. Additionally, a purpose-designed tool allows the test team to maintain these components. These tools often allow virtual components to be developed from a specification (for example, a WSDL or a COBOL copybook) or recorded from existing system behavior.

Lesson 5 – Measure performance early and often

Example – Mobile telecommunications company

Telecommunications projects can involve coordination between many interconnected systems, and this project was no exception. On one specific project, a large team was assembled, an impressive architecture was designed, and the implementation begun. Deadlines were set and commitments were made to internal stakeholders far into the future. As they drew near, and deliverables slipped, shortcuts were taken. Due to the complexity of the overall solution, the system was not performance tested until it went into production with real users and real customers. Instead of enabling the sign-up of dozens of customers per minute, the system could only handle two per minute and order fulfillment was delayed at times by more than 24 hours. Sometimes orders did not occur at all, forcing the support staff to complete orders manually. There was no easy fix to the problem as the architecture was fundamentally flawed from the beginning.

Integration projects are often carried out with promises of faster transaction processing and greater throughput. These promises need to percolate into the architecture of the system and, from there, to individual components. These performance requirements need not be onerous—some systems are over-specified and over-tested in these areas. Unfortunately, the simple measure of requests per second is not enough to guarantee well-defined results. Instead, performance needs to be analyzed for several characteristics. First, what is the target time to receive a reply to a request, that is, what is the component's response time? Second, if two requests are received "at the same time"[1], when are the responses received? Third, can this performance be maintained over a prolonged period, and with higher levels of simultaneous requests?

The idea of performance testing at this early stage identifies catastrophic design errors. Clearly, the exact run-time environment and load may not be available. However, if problems are already apparent in isolation, they can be fixed now, earlier, rather than in a few weeks or months when full system performance testing can be carried out. In extreme cases, fundamental design faults have been carried into many hundreds of implementations, making a timely recovery impossible. Ideally, performance measurements should be undertaken at the same time as functional testing, and the component load tested before it is accepted.

Preventing a problem is always a better strategy than fixing a problem.

Given that the actual performance of the component will ultimately be affected by its design, the required performance should be part of the design process. We are not advocating over-engineering, but a design fit for the purpose.

Here is an example. A severe performance problem in a component communicating trades in real time to an exchange was seriously degrading performance. The team could not understand why it was only able to process ten transactions per second. It had been specified and tested to several hundred trades per second. Detailed examination of the log and comparison with the code showed that even short sections of the program were taking as much as eleven milliseconds to execute.

Perhaps less well-equipped teams would have commenced on an immediate redesign of the problem areas. However, knowing the component already met the required performance criteria made it easier to look for other problems. Ultimately, it was discovered that the process flushed its log file after every line, and someone had moved its logging area to a network-mounted file system. Moving the logging back to a local file system restored the component performance to normal.

In conclusion, target performance criteria should be generated for the unit and development should not stop until the unit meets or exceeds them. Performance criteria can substantially affect the design of a unit and must therefore be specified during requirements gathering to avoid impacting the deliverable.

Lesson 6 – Test interleaving, concurrency and state-related behavior

Example

A junior developer, who had been working with real-time systems for a number of years, was asked to develop a new component. He was always thorough and tested it for a few weeks before it was deployed. Almost immediately, problems became apparent. Updates destined for a particular customer were actually changing the accounts of different customers on certain occasions.

In the previous example, examination of the code showed that the essence of real-time programming had been ignored and simple global variables were being used to store information. In other routines, information was being read from these global variables rather than the incoming message. The developer's testing had only used a single transaction followed through to completion, and then tested with a different set of data through to completion. The transactions had never been interleaved—a test that would have revealed the problem immediately.

It is important to establish these testing exceptions during test planning.

Some components in a system will not require this level of extensive testing. They may be simple transformation and translation components, simply passing data in and out of another system. Examples would be one customer's information being updated into another customer's account.

An early code review by an experienced team member might catch this type of mistake, depending on the size of the program. Standardizing on variable naming conventions can also help avoid this type of problem.

Lesson 7 – Automate for agility

Example 1

A leading financial institution's systems must be updated constantly to keep pace with changes in the financial markets. To support this requirement, releases need to occur every two weeks. The testing team would be unable to cope with this rate of change without test automation. Through integration with a test management tool such as IBM® Rational® Quality Manager, or a build tool such as IBM Rational Team Concert™, Maven or Hudson, IBM Rational Test Workbench provides a quick and convenient method of deciding whether or not to deploy.

Example 2

One of the world's foremost energy companies has an automatic build-and-test system. An agent watches the configuration management repository for new versions of system components. When a certain amount of time has passed without new check-ins, the agent checks-out the latest set of code and attempts to compile it. If the code compiles, the agent uses Rational Test Workbench to automatically test it. The results are emailed to the project team.

A tool that helps you create, maintain and execute automated tests needs to be able to cope with complex scenarios.

Services will inevitably change during their lifetime. While test automation of graphical user interfaces (GUIs) can be laborious and changes time consuming to implement, test automation for services is different. In fact, the automation often fits neatly with a common goal of changes namely, backward compatibility. An existing automated test can simply be duplicated, and the copy modified to test the new version, leaving the original to test for backward compatibility.

This is particularly convenient, as visual inspection of some service test execution results may not detect subtle problems. Automating the tests is not simply a case of comparing a previous result with another as some parts of the message may be different on each test execution, such as a timestamp or sequence number.

Another benefit of testing at the service layer is that it minimizes problems such as channel explosion which currently afflict the GUI approach. Tests can also be run earlier in the lifecycle, catching problems when they are less costly to resolve.

Lesson 8 – Buy, do not build

As a software vendor, you may feel we are bound to say this. But consider some of the arguments: rich features, ease of use, no maintenance and the ability to start testing immediately. Some organizations have started to build their own systems, but they rarely meet today's business requirements and lack the flexibility to incorporate future requirements found in an off-the-shelf system. A commercial system will be stable, powerful, proven and available now. The developers at your company are better off building systems for your business, not developing software to help them develop software.

Finally, some mention should be made of open source testing software. Although there is no charge for the software, it is not free. Often the productivity of people using these tools is far lower than those using commercial tools. Therefore, the *real* cost of adopting an open source solution is mostly hidden. Adopting open source may seem attractive, as often testers are familiar with the tools, but off-the-shelf commercial software users typically find that principles transfer easily, with open source tools having taken some of the simpler and easier-to-implement ideas from commercial offerings.

Conclusion

As distributed computing systems have grown more complex, designing test plans that test the entire system—quickly, efficiently and thoroughly—are crucial to a successful project. Automated testing tools and virtual components are key parts of the testing process.

For more information

To learn more about Rational test automation solutions, please contact your IBM representative or IBM Business Partner, or visit the following website:
ibm.com/software/rational/offerings/quality

See also:

- IBM Rational Test Workbench
  ibm.com/software/rational/products/rtw
- IBM Rational Performance Test Server
  ibm.com/software/rational/products/rpts
- IBM Rational Test Virtualization Server
  ibm.com/software/rational/products/rtvs

Additionally, IBM Global Financing can help you acquire the software capabilities that your business needs in the most cost-effective and strategic way possible. We'll partner with credit-qualified clients to customize a financing solution to suit your business and development goals, enable effective cash management, and improve your total cost of ownership. Fund your critical IT investment and propel your business forward with IBM Global Financing. For more information, visit:
ibm.com/financing

© Copyright IBM Corporation 2012

Software Group
Route 100
Somers, NY 10589 USA

Produced in the United States of America
July 2012

IBM, the IBM logo, ibm.com, Rational Team Concert, and Rational are trademarks of International Business Machines Corporation in the United States, other countries or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. Other product, company or service names may be trademarks or service marks of others. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at **ibm.com**/legal/copytrade.shtml

This document is current as of the initial date of publication and may be changed by IBM at any time. Not all offerings are available in every country in which IBM operates.

THE INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING WITHOUT ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND ANY WARRANTY OR CONDITION OF NON-INFRINGEMENT. IBM products are warranted according to the terms and conditions of the agreements under which they are provided.

[1] When referring to coincidence and simultaneousness, care needs to be taken about what can really be achieved physically. For example, consider the testing of a remote service across a LAN. Requests sent "at the same time" can only ever be received one after the other in all but extreme hardware configurations. Such factors are often considered "small" effects in this type of testing. Other effects are not so easily ignored, such as the batching mechanisms employed by various middleware implementations. However, while these effects cannot be ignored and may not produce consistent timing measurements, they do indicate the *real* performance experienced by *real* applications and, therefore, are relevant.

Please Recycle

RAW14305-USEN-00