Test-Driven Development in Enterprise Integration Projects

November 2002

Gregor Hohpe
Wendy Istvanick

Copyright ThoughtWorks, Inc. 2002

Table of Contents

Summary 1

Testing Complex Business Applications 2

- Testing – The Stepchild of the Software Development Lifecycle? 2
- Test-Driven Development 2
- Effective Testing 3
- Testing Frameworks 3
- Layered Testing Approach 4

Testing Integration Solutions 5

- Anatomy of an Enterprise Integration Solution 5
- EAI Testing Challenges 6
- Functional Testing for Integration Solutions 7

EAI Testing Framework 11

Design for Testability 13

Web Services and Service-Oriented Architectures 14

Non-Functional Testing 15

Conclusion 15

ThoughtWorks® is a registered service mark of ThoughtWorks, Inc. in the United States and/or other countries. All other product and company names and marks mentioned in this document are property of their respective owners and are mentioned for identification purposes only.

Summary

If testing software applications is a good idea, testing enterprise integration solutions must be an excellent idea. After all, integration solutions form the backbone of many modern enterprises, linking vital systems and business processes with real-time data interchange. Any defect or outage in an integration solution is likely to affect large portions of the enterprise, potentially causing loss of revenue and data.

Besides the obvious damage done to the business by defective software, the lack of a structured testing approach also causes delays and unnecessary work during the software development cycle. A test-driven approach integrates automated tests into the requirements and analysis phases of a project, resulting in higher-quality applications and more expeditious integration and deployment cycles.

It is surprising then that so many integration solutions are deployed with little or no testing. Testing, if any, is usually done manually and sporadically. One of the reasons integration solutions are not tested thoroughly is the fact that testing asynchronous, message-based middleware solutions is challenging. These solutions are complex, distributed, heterogeneous and asynchronous in nature. To make things worse, there are very few tools available to aid in these testing efforts.

This paper examines the challenges of testing enterprise integration solutions and proposed a testing approach to tackle the inherent complexities. This approach is accompanied by a framework that makes integration testing more effective and efficient. Additionally, the paper discusses the implications of Web services and service-oriented architectures on the proposed testing framework, and provides guidelines for the design of new integration solutions to improve testability.

The presented approach and framework have been developed and enhanced over a series of integration projects. The tools have helped accelerate a number integration projects and are continuing to undergo refinement as integration technologies continue to evolve.

Copyright ThoughtWorks, Inc. 2002

Testing Complex Business Applications

Testing – The Stepchild of the Software Development Lifecycle?

Testing computer software is a good idea. While pretty much everybody involved in building or using computer systems would agree with this, software riddled with hidden defects continues to be deployed into production environments. The consequences range from frustrated users to millions of dollars' worth in system outages and software maintenance expense. Testing is obviously not a novel idea and a number of vendors sell testing methodologies and tools in pretty much any shape or form. So why are we not more successful in creating bug-free, high-quality software?

The first part of the answer lies in the expression "high-quality". Before we can claim that a piece of software is of high quality we need to define what quality means to us. A high-quality system should not have any defects. We can define defects as deviations from the functional specification. So if the system behaves exactly like the specification we should be able to call it "high-quality". What if the system functions as specified, but not as expected? Also, what do we call a system that functions as specified but is difficult to enhance and maintain?

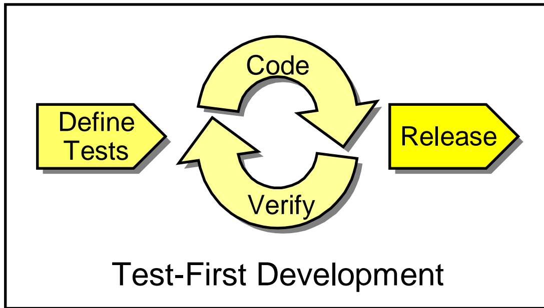
Apparently, "quality" and "testing" cover a long list of possible characteristics. We typically divide the list of requirements for a system into functional and non-functional requirements. Functional requirements define the business functions the system needs to support. Non-functional requirements (sometimes referred to as "ilities") define attributes that relate to the operation or maintenance of the system, such as scalability, maintainability, reliability, etc. Likewise, we can divide our testing efforts into functional tests and non-functional tests. Functional tests verify functional specifications while non-functional tests verify non-functional requirements. In addition to functional tests, developers execute "white-box" unit tests to verify individual code segments that may or may not be directly related to a piece of functionality.

Even though requirements may be well defined and the correct approach to testing is well understood, many projects still release buggy products. One common reason is the perception that "we don't have time to test more extensively". This is usually the fact because testing tends to be the last phase before the (firm) release date, and any slippage in the requirements or development phases are compensated for by shortening the testing phase. In most cases, any "savings" in the testing phase are then paid for in multiples during the maintenance of the prematurely released software.

Test-Driven Development

One effective way to avoid unclear requirements and a "too little too late" testing approach is to integrate testing into the requirements analysis and development phases of the project. By integrating these efforts we can help close (or reduce) the gap between requirements definition, solution development and testing. One of the reasons testing is usually deferred until the end of the lifecycle is the inevitably high rate of change during the development phase. Testing tends to be labor-intensive and it is not realistic to change the test scripts and re-exercise all test cases every time a code change is made. This perceived conflict can be resolved if we can automate testing to the point where a suite of test cases can be executed automatically, actual results are compared to desired results and deviations are summarized in a report. Now we can run the tests as often as we like since no or very little human intervention is required.

But won't it take an inordinate amount of time to code and maintain all the test cases? There is no question that coding test cases initially requires additional analysis and development time. On the other hand, we can consider these test cases part of the requirements specification process. What better specifications than a set of executable test cases? Also, coding test cases during the analysis phase encourages communication between customers, analysts and developers. It helps minimize the development of unnecessary functionality and provides the foundation for fast-paced, iterative development. Thus, it contributes to project scope management, project risk management, project schedule management, project cost management and project communications management. If we can leverage a common testing framework, these benefits more than outweigh the time required to code the individual test cases.



This approach can be summarized as *Test-First Development* or *Test-Driven Development*. Test-driven development implies that rather than deferring testing until the end of the project we move it to the beginning of the lifecycle. Before developers code a new piece of functionality they define and develop the test cases describing the required functionality. Then, they develop code modules until all test cases succeed. Development occurs in rapid iteration cycles between developing, verifying and correcting code.

Effective Testing

In order to support a test-driven development approach, testing has to be comprehensive, accurate and efficient. This generally requires the presence of a testing framework. In order to support an effective testing strategy, such a framework needs to support the following requirements:

Simple Creation – Developers are more likely to write tests if they are easy to create (and managers are also less likely to tell them that there is no time to write test cases).

Definitive Pass/Fail Criteria – A pass or fail answer gives developers and analysts a definitive answer on whether a test is working or not. Often tests simply create logs that developers must analyze to decide if the test passed or failed. Instead, the framework needs to verify results and deliver binary pass/fail results.

Reliable Failure – Tests need to fail when something is actually broken, and not when something has merely changed. Traditional automated testing approaches are often brittle, breaking with every change. This provides little useful feedback, and often creates more work than it saves.

Push Button Repeatability – Tests must be easily repeatable or they quickly become neglected. An automated test framework enables repeatability by defining a common way of executing test cases and reporting test results.

Extensibility – The testing harness needs to be extensible so developers can extend the framework to include specific technical and business requirements.

Testing Frameworks

Automated testing relies on the programmatic execution of test scenarios and the verification of the results against the expected outcomes. Efficient development and execution of test cases can be accomplished by using a testing framework. A testing framework provides reusable components to organize and execute test cases and verify and report test results.

A testing framework consists of the following core components:

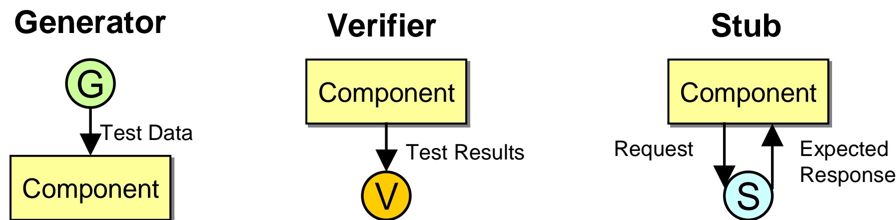


Figure 1: Testing Framework Components

A test data *Generator* creates test data and feeds it into the component under test. There are a number of strategies for the creation of test data. Test data can be read from a script file that contains instructions on the test execution as well as test data. The data is translated into function calls (or messages) to the component. Another common source of test data is a capture / replay mechanism which captures test data output by an external application or another test case and stores it for replay. A third option is the random or algorithmic creation of test data. This option has the advantage that a long series of unique test cases can

be created, but generally requires additional development effort. Complex systems may require thousands of test cases so that it makes sense to use additional tools to organize the suite of test cases hierarchically.

A test result *Verifier* compares actual test results with expected results. It is important that a verifier makes an accurate, binary decision whether a test was successful or not. Expected results can be prescribed in a way similar to the test data generator. In most cases, the expected results are part of the test script. If an algorithmic or random test generator is used, the verifier needs to communicate with the generator and compute the expected results for each test case. For example, if a test data generator generates random line items for an order the verifier needs to compute the total and compare it to the actual billed amount resulting from the order. In most cases, verifiers are tied into a reporting mechanism that reports on the number of test cases that were executed and the exceptions that were encountered.

A *Stub* is a testing component that simulates part of the system that is not under test. The component under test may depend on another component that may not have been completed yet or would introduce undesirable side effects into the test. Therefore, we replace the external component with a substitute called a stub. The stub receives data from the component under test and returns "dummy" data so that the component under test can continue processing. The response can be fixed, driven by a script or algorithmically derived from the request.

Layered Testing Approach

An effective functional testing strategy needs to account for dependencies between components. As much as possible, we want to unit-test components in isolation so that we are not distracted by possible faults in other components. As we integrate the system and conduct integration testing we want to proceed in a way that we integrate new components into an already tested subset so that we can constrain the possible sources of error to the new component or the interface between the new component and the existing components.

Well-architected systems tend to be organized in multiple layers with each layer depending only on 'lower' layers. The lower layers contain generic functions that are used by the more specific functions in the upper layers. Forcing one-way dependencies between upper and lower layers avoids circular dependencies between components. It also clarifies the process of unit and integration testing as outlined in the following sections:

Unit / Component Testing

Unit testing focuses on a specific component in the system. We want to eliminate outside dependencies as much as possible when testing a single component in isolation. This is easy if we are dealing with low-level components that do not have many dependencies. Therefore, we can use a test data generator to feed test cases into the component and use a verifier to examine the results (see component A in Figure 2). For higher-level components we need to stub out any dependent components to focus solely on the component under test (also, some of the dependent components may not have been built yet). Therefore, we use a stub to simulate any external components that it may depend on. The advantage of this testing approach is that we can test in isolation, allowing us to test multiple components from different layers in parallel. Because unit testing is limited to looking only at a small portion of the system at a time it may not discover integration issues between components. However, testing at this level ensures that when integration begins we can have confidence in the functionality of each component being integrated.

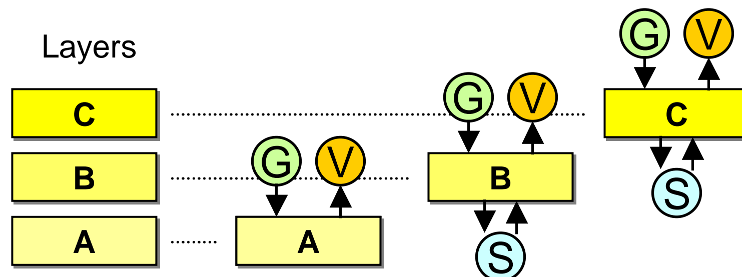


Figure 2: Unit Testing a Layered Architecture

Integration Testing

Integration testing focuses on the interaction between multiple components. The integration testing strategy focuses on the lower layers first because they have the least external dependencies. Testing the lower layers first also reduces the amount of uncertainty when testing the upper layers. If an integration test case fails in the upper layers, it is unlikely that the reason is a faulty lower-layer component since these components have already been tested. Therefore, we can narrow our scope for debugging to the new component or the interaction between the new component and already tested components. As a result, this integration testing approach is referred to as "bottom-up" testing (see Figure 3).

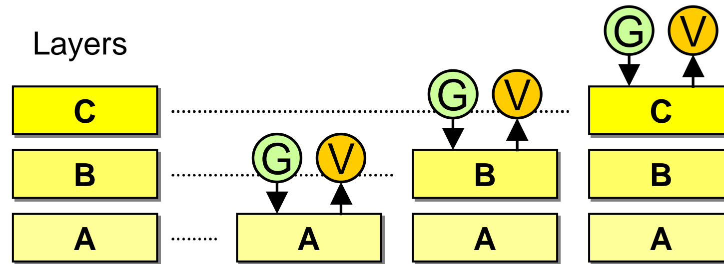


Figure 3: Integration Testing a Layered Architecture

Again, we use test data generators and verifiers to drive the automated test.

Testing Integration Solutions

Anatomy of an Enterprise Integration Solution

Test-driven programming has been used successfully in many application development efforts. In order to understand how to leverage some of these benefits in enterprise integration projects, we first need to have a closer look at what these integration solutions look like.

An enterprise integration solution typically consists of the components depicted in Figure 4.

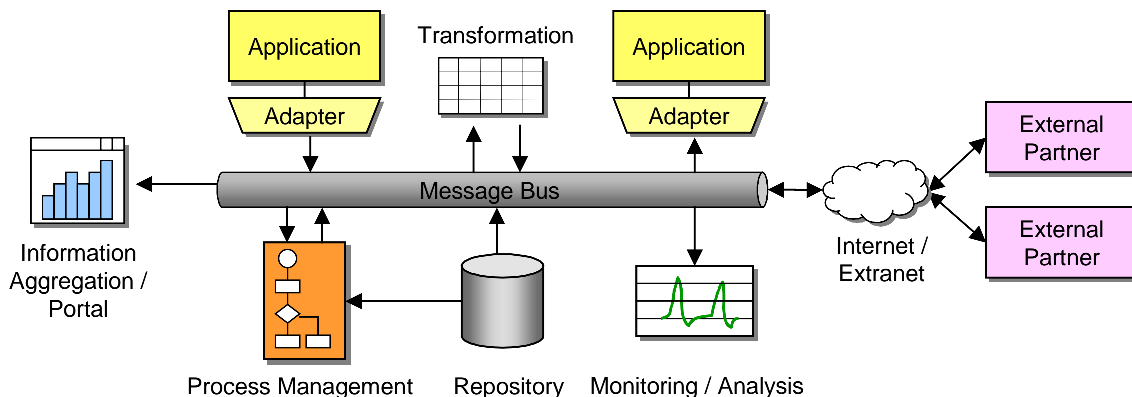


Figure 4: Anatomy of an EAI Solution

A common *Messaging Infrastructure* or *Message Bus* enables secure and reliable communication across the enterprise network. The message bus has the ability to *route* messages between message publishers and message subscribers. It queues messages if a subscriber is not available, relieving the publisher from the need to store and resend messages.

Applications publish data to or receive data from the message bus via *Adapters* or *Connectors*. These connectors make an application's proprietary interface available to other applications via the message bus.

Copyright ThoughtWorks, Inc. 2002

Because different applications represent data in different ways, *Transformation* functions translate an application's proprietary data format into a common data format that can be understood by other applications. Without this transformation capability the message bus resembles a telephone system that provides a common network, but connects participants speaking different languages.

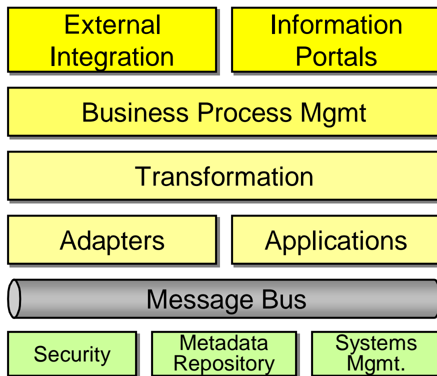
Once we have a message bus, adapters, and data transformation functions to implement reliable data exchange between multiple applications, we can model *Business Processes* that represent a series of actions, each carried out by a different application. For example, a 'New Order' process may need to verify the customer's credit rating, check inventory, fulfill the order, track shipment, compute taxes, invoice the customer and track payment. Naturally, a number of systems would collaborate in such a process, some of which may be external to the enterprise. The business process management component of the EAI solution provides tools to model the rules that guide such a process and manages the execution of long-running business transactions.

A common *Repository* stores business rules and business object definitions, such as 'Order', 'Customer' or 'Line Item'. These common definitions allow for loose coupling between applications and make the business process definition independent of any application's proprietary data format. This type of information describing the format of message data is referred to as *Metadata*.

Once we integrate applications and business processes through an EAI solution, we can interface more easily with *External Partners* such as suppliers, distributors or customers. Special adapters convert internal messages into Internet-compatible protocols such as XML over http. Additional encryption secures information traveling over the public Internet.

Another significant advantage of an integrated message bus is the ability to display information from different applications in a single *Portal*. Portals aggregate information from multiple systems and display it in a consistent, user-definable format. Some portals merely display information (e.g., management information portals) whereas others allow users to generate business transactions in the form of middleware messages (e.g. an order entry portal).

A typical EAI solution spans a significant number of applications, often times distributed across multiple locations. Centralized, real-time *Monitoring* of all components quickly becomes a critical function of successful enterprise integration. Monitoring evaluates the health of all components and triggers a restart or fail-over action if a failure is detected.



The different components of an integration solution can be arranged in a layered 'stack' in a way that each layer builds on the layers below (see diagram): the message bus is the basis for applications and adapters to communicate. Data transformation translates messages exchanged between these applications. Business processes operate on top of transformed, standardized messages, whereas external integration and portals are enabled by the integrated processes. Lastly, system functions such as security, metadata management and systems management support the overall solution.

EAI Testing Challenges

Enterprise Application Integration (EAI) solutions enable the connection of separate systems via a common information infrastructure. Integration solutions consist of a significant number of components and spans across many different pieces of functionality. In order to test an integration solution effectively, all parts of the solution have to be addressed. Besides sheer scope and complexity, effective testing of an integration solution provides a number of additional challenges as compared to testing business applications:

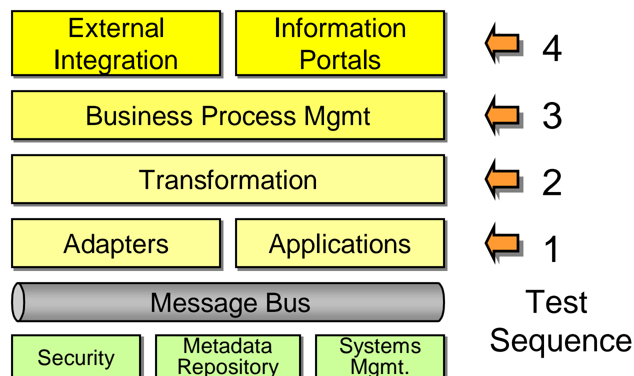
- EAI environments are inherently heterogeneous. Application development environments tend to be much more homogeneous so that a single testing tool such as jUnit /xUnit can cover the vast majority of unit test cases. An EAI solution, however, may span across multiple languages and platforms, including mainframe legacy systems. This makes the generation of test cases as well as the capture and comparison of test results significantly more difficult.

Copyright ThoughtWorks, Inc. 2002

- EAI solutions typically integrate a set of packaged or custom applications. Generally, access to the internal structures of these applications is limited, making it difficult to verify the success of a test case. Worse yet, some of the applications may be external to the enterprise, making access even more challenging. For example, how can we verify that the test case 'Charge the customer's credit card' succeeded?
- Since an EAI solution works with complex applications it may not be possible to re-create a dedicated test environment. Internal as well as external services may not provide separate environments just for testing. How can we test charging a credit card or shipping and tracking a package if we have no dedicated test environment?
- Most application testing is synchronous: a test harness calls a specific function and the actual results are available immediately and can be compared to the expected result. EAI solutions typically rely on asynchronous messaging. As a result, an EAI test case may send a message to a component, but may not receive an immediate response.
- The asynchronous nature of message-based integration also causes a new category of potential problems. Asynchronous, message-based systems may suffer from concurrency problems and temporal conditions similar to distributed or highly parallel applications. These problems are very difficult to test and diagnose.
- Many business processes depend on timed events. For example: send an e-mail to the customer if the shipping of an item is delayed by more than three days. In order to test this function we need to either wait three days or be able to manipulate the system time across multiple systems.
- The minimal unit of test in an EAI solution tends to be larger and more data driven than in application development. Testing a single method of a Java or C# class generally requires only a moderate amount of preparation. On the contrary, testing a single integration function may require complex data setup inside multiple packaged applications. For example, to test an order function, we may have to configure a catalog of products as well as a customer list.
- Testing tools are not yet readily available. Many EAI vendors have been focusing primarily on integration functionality and messaging rather than testing support. Most generic testing tools on the other hand focus on application testing or user interface testing. Also, the vendor-specific APIs of most EAI suites require testing tools to be customized to each specific vendor.
- Integration solutions are becoming increasingly complex. Many integration tools and methodologies are still geared towards situations where EAI meant simple data exchange between two applications. Contemporary integration solutions typically consist of a combination of complex transformations, processes and actual business logic, which makes testing much more time consuming and complex.

Functional Testing for Integration Solutions

We described the benefits of a layered, bottom-up test approach which tests the lower layers first and then works "up the stack" layer by layer. We can apply this approach to the integration stack as defined in the previous section, beginning with the message bus and working our way up to external integration and information portals. In most cases, the EAI vendor provides the message bus layer as a black box layer that is not customizable. As a result, the bus itself does not have to be subjected to functional tests. However, the bus is often times the focus of non-functional testing such as scalability, performance, fail-over behavior, etc. Accordingly, the first phase of functional testing of an EAI solution will focus on the applications and associated adapters. These tests confirm that the applications



can produce or consume the correct messages. Once the behavior of the participating applications has been confirmed, transformations between different message formats can be tested. Once the communication in a standard message format is confirmed, we can test the business processes that operate on top of the standard messages. Last, we can focus on integration between external partners and information portals.

The fact that we integrate a solution from the bottom up does not mean we have to build the whole integration solution layer by layer. In fact, this would be contrary to an agile and iterative development approach. The layered testing and integration approach can be used for functional slices of the solution, i.e. we can integrate and test the adapters, transformations and process definitions for a single interaction of an overall integration solution. In the next iteration, we can develop, test and integrate another interaction.

The following sections describe how each of the individual layers of an EAI solution can be tested.

Testing Applications and Adapters

The lowest-layer building blocks of an EAI solution are usually packaged or custom business applications -- pretty complex pieces in themselves. Since EAI solutions act as the glue that holds together many disparate systems, they must provide interfaces or adapters through which each of these systems can be invoked or accessed.

Before we start testing an integration solution, we need to be sure that the applications themselves function correctly. This is less of an issue with packaged applications that are likely to have been tested before they were shipped. If we deal with custom applications, we need to make sure that these applications have been tested using traditional test methods before we start integrating. This is important to reduce the number of potential sources of error and to simplify defect analysis and correction.

Assuming that the applications work correctly internally, the EAI tests focus on the added interfaces and adapters. In the basic case, we can test the interface by sending a message in the data format expected by the adapter, invoking the component through the interface, and comparing the result returned through the interface to what would be returned by the application if it was invoked directly from the API (Scenario 1).

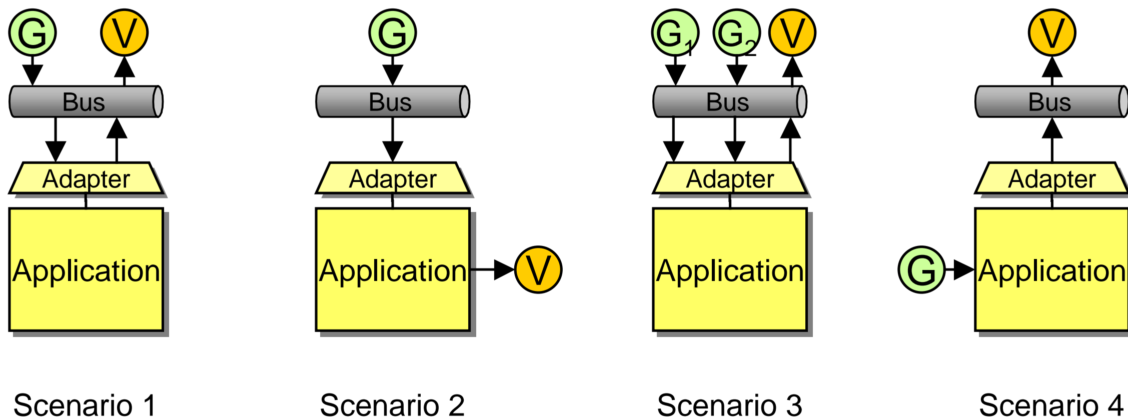


Figure 5: Testing Applications and Adapters

In some cases, however, the application will not return a response to the test message or may produce a message that contains insufficient data for a meaningful test result. In this case we may need to use a verifier that has direct access to the application interface or the application's data store (Scenario 2). Verifying the internal structures of business applications can be very complex depending on the nature of the application. Most applications store their internal state in a relational database so that we can use database verifiers to ensure the correctness of the business transaction. However, this approach requires detailed knowledge of the applications data model and can be cumbersome in situations where the application's data model is highly normalized. In some situations, we can use a separate message to query the state of the application instead of accessing the application directly (Scenario 3). In this case, we use two test data generators. The first generator sends the test data to the application in the form of a message. The second generator sends a 'read' message to the application to verify the results of the first message. In

Copyright ThoughtWorks, Inc. 2002

most cases, the second generator can be quite simple as long as it obtains unique key information from the first generator. If the application does not provide a function to retrieve this data, it can be a good example of design for testability (see below) to add this function to the adapter.

Other test cases require the publication of messages to the message bus based on business events that occur in the application. In this case, we need to use a test data generator that can trigger events inside the application (Scenario 4). Similar to Scenario 2, we can trigger these events via the application's API or by manipulating internal data structures. We use a message verifier to check the correctness of the published event. Similar to the previous cases we may be able to trigger the event by sending a specific message through the adapter rather than modifying the internal data structures directly. The feasibility of this approach depends on the application architecture and the adapter functionality.

Testing Transformations

Transformations can be tested relatively easily. We can simply provide input data for the transformation, run the transformation, and compare the results to the corresponding expected output (see Figure 6).

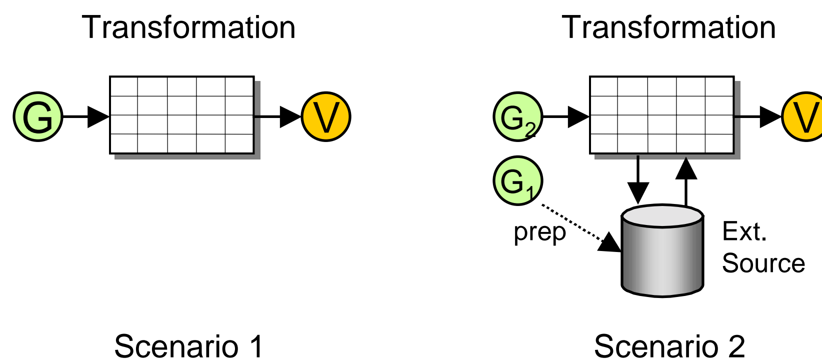


Figure 6: Testing Transformations

Transformation tests can benefit from random test data generators to ensure robustness across a wide range of input data. If the integration solution includes a well-structured metadata repository, tools for automated test case generation can be employed. For example, some tools exist to create example XML documents based on an XML schema definition. The success of this approach depends primarily on the level of detail provided by the metadata repository. For example, automatically generated test cases will be more meaningful if the repository allows the modeling of value constraints or dependencies between data elements as opposed to basic data types such as 'text' or 'integer'.

Some transformations rely on external data sources (e.g. look-up tables). In these cases, we need to make sure that the test data correlates to the data contained in the external sources. We can either use a fixed set of external data or prep the external data source with a separate generator (Scenario 2). Sometimes it makes sense to equip transformation with a *control port* that allows the configuration of the transformer via messages. This simplifies testing because we can send a message with prep data as opposed to manipulating the database directly.

Testing Business Processes

The stateful nature of business processes makes testing more complex than transformations. Business processes typically include decision points so that a single process can result in different outcomes based on the decision criteria. Therefore, we generally need to run a suite of test cases against a single process in order to obtain coverage of all possible execution paths of the process. To make things worse, many decision points inside a process depend on external events or timed events. In many cases, we can trigger external events by inserting stubs in place of the external system (see Figure 7). Since we have control over the stubs we can instruct the stub to return different information depending on the test case we are running. This feature can also be used to simulate failures. If the business process communicates directly

Copyright ThoughtWorks, Inc. 2002

with an external system's API things become more complicated. We would have to manipulate the external system to provide the desired message back to the business process.

Testing timed events requires some additional thought. It is fairly common that a business process branches based on a time delay. For example, if an order cannot be filled within three days, we want to notify the customer with an e-mail message. There are a number of approaches to test these types of functions:

- We can wait for three days. This test will be the most authentic, but will not work very well as part of an automated test suite that we want to execute after every build cycle.
- We can make the delay configurable and set the parameter to 3 minutes (or 10 seconds) for testing purposes. A short delay in the test script will then cause the process to branch off. Since we are sure of the quality of the timer function itself, we can assume that the same thing would happen if the delay were set to three days. However, these kinds of tests cannot simulate other timed dependencies. For example, some other part of the process may be programmed to change some relevant data after two days, causing our reminder e-mail to fail. Since we set our timeout to 3 minutes, we would not catch this test case. To make this type of test more realistic we can change all timed parameters to a fixed scale, e.g. we can change all 'day' units into minutes.
- We could also advance the system clock so that the business process thinks three days have passed. This causes a more realistic simulation of timing dependencies between tasks. However, many integration solutions span across multiple systems so that we have to adjust the time on all systems in a consistent fashion. Changing the system clock can be a dangerous exercise since many processes rely on the time increment to manage time-outs as well as software licenses. A safer way to manipulate the system clock is to design the solution such that all integration functions retrieve the current time from a single component. We could then manipulate this component to simulate timed events. The potential downside is the additional overhead involved in accessing a remote component to retrieve the system time.
- We could also design the business process so that a specific external message forces the process into the exception path. Adding such a function for testing purposes simplifies testing a great deal, but it does not correctly simulate timing dependences between multiple timed events.

Some would argue that by this level the testing has gone beyond unit testing and reached into the integration testing area. While this would be true in the non EAI world, the nature of EAI forces testing to occur in a more integrated way.

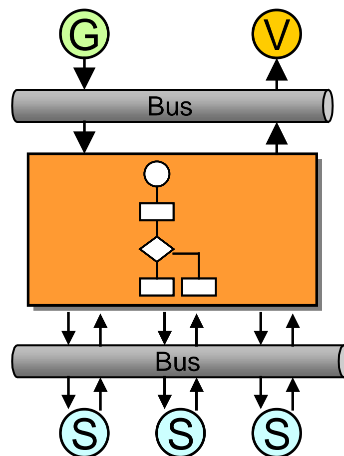


Figure 7: Testing Business Processes

Testing business processes is a complex task. In many cases we will not be able to completely automate the testing of a business process component from beginning to end. However, just because we can't automate everything doesn't mean we shouldn't automate anything.

Copyright ThoughtWorks, Inc. 2002

Testing External Interfaces / Portals

Since most portals are Web-based, testing portal solutions resembles testing generic Web-based applications. Therefore, we can leverage common testing tools, including commercial products such as WinRunner or SilkTest, as well as open-source tools such as httpUnit. Alas, user interface testing can be an arduous task and can result in brittle test scripts, so it is important that all back-end functions be properly tested so that the portal testing can focus primarily on the presentation and the interaction between user interface and the messaging back-end.

Because portals can access a number of individual applications, setting up test data may become a labor-intensive task. In many cases, though, we can leverage some of the data setup routines that we used to test the application adapters or any business processes.

Testing integration with external partners is similar to testing integrated applications. The bigger challenges usually lie in the lack of control over the test environment because it is usually owned by another entity. This can eliminate any options to access these applications directly and can make automated testing efforts more time consuming. Due to the stronger separation, design for testability is even more critical for external interfaces. For example, we will likely not be able to access the external application's database to verify the successful completion of a transaction. Therefore, it is useful to add a function to the interface that allows us to retrieve this information via messages.

EAI Testing Framework

Testing integration solutions is no simple task. In order to enable test-driven development and rapid iteration, we need to leverage an automated testing framework so we can get consistent test results without burdening the development or test teams with laborious test execution. Furthermore, automated testing has the ability to change the dynamics of the development team in a way that testing is no longer seen as a burden but becomes a natural part of the development process. This change in developer mindset is key to test-driven development – for application development and enterprise integration alike.

In order to make the test framework efficient, we need to make sure it allows the developers to:

- easily write tests for new code they are creating
- quickly run those tests and the entire regression-test locally on a development machine before adding new code to the code base
- automatically add new tests to the regression-test repository at the same time as new code is added to the code base
- automatically and quickly run the entire regression-test suite against a new build of the code base
- quickly view the results of the automated regression-test run, both on a development machine and on the build server
- receive automatic notification if their new code has "broken" any tests in the regression-test suite

As described earlier, an automated test framework has to provide test data generators, verifiers and test stubs. When examining different testing approaches for applications and adapters we realized that sending messages to the messaging bus alone might not be sufficient to thoroughly test an integration solution. We also need the ability to stimulate events in applications or verify an application's state directly. In addition, we want the generator and the verifier to be driven by easy-to-maintain test scripts.

As a result of these requirements, we designed an EAI test data generator with the following internal architecture (see Figure 8).

Copyright ThoughtWorks, Inc. 2002

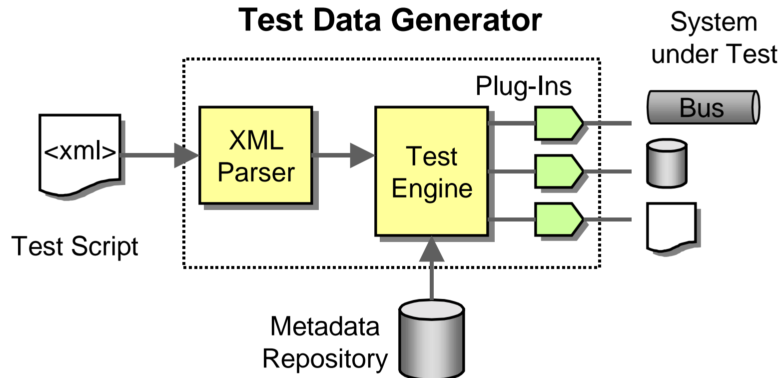


Figure 8: Test Data Generator Internal Architecture

The test data generator is driven by an XML input file which is human-readable, easily extendable and can be parsed by any XML parser. The parser drives the test execution engine that interprets the commands contained in the test script and converts the test data into the required format. The following listing shows an example of a simple test script:

```
<test>
    <tibco>
            <publish message="test\xml\tibcoMessage1.xml"
                     subject="subject.name.publish"/>
            <subscribe var="inbound" subject="subject.name.subscribe"/>
    </tibco>
    <assert var="inbound" path="order.total" value="56.10"/>
    <assert var="inbound" path="order.tax" value="2.34"/>
</test>
```

Since we need to be able to send middleware messages as well as trigger events in applications or pre-configure look-up tables for transformations, we architect the data generator independently from the transport and use plug-ins to support the individual transport mechanisms. Initially, we need to support sending messages, making updates or inserts to a database or place a trigger file in a directory. File-triggered testing is difficult because the file may be required on a machine different from the one executing the test script. In this case we have three choices:

- We can 'mount' the remote file system (i.e. make it visible to the local machine)
- We can use a file adapter to create the file via the messaging infrastructure.
- We can use a file transfer protocol such as FTP to transmit the file.

In the second case, we would send a message containing the file data to a remote file adapter which will then create the file. File system mounting may be the easier option, but it makes the test suite more OS dependent (it needs to know the exact path where the remote file system is mounted) and it may not be available depending on the transport protocols that are available. FTP is a common protocol for file transfer and may be a viable option. New plug-ins can be created depending on the testing needs, e.g. if we have to make a direct API call to trigger a test case inside an application.

Many EAI solutions transfer data using a weakly typed data format that represents message data in a generic tree structure. In order to assure creation of correctly typed messages, the test engine can access the metadata repository and compare the message data from the test script to the metadata in the repository. For the creation of coded test cases, the test framework can create object wrappers in a common programming language such as Java to enable developers to easily create test data. These wrappers are generated automatically from the metadata repository and result in a set of Java or C# classes that mimic the structure of the middleware messages. The wrappers contain all the necessary code to convert between the strongly-typed programming language and the middleware message format. Thus, generated wrappers jump-start EAI testing significantly by allowing developers to code against regular objects rather than the

Copyright ThoughtWorks, Inc. 2002

proprietary middleware API. A more detailed description on object-wrappers and EAI testing patterns can be found in the ThoughtWorks whitepaper “Continuous Integration with TIBCO ActiveEnterprise”.

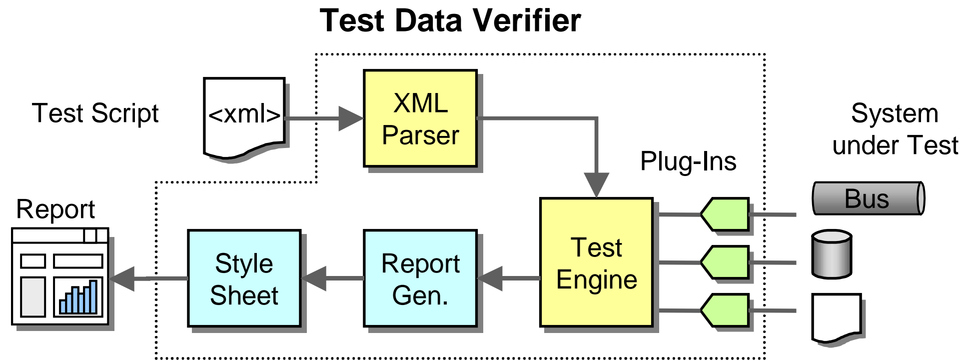


Figure 9: Test Data Verifier Internal Architecture

The test data verifier resembles a mirror image of the test data generator (see Figure 9). Plug-ins receive data from various data sources, including messages, database queries and files. The data is then passed to the engine which verifies the data with the expected outcome. The engine can also monitor the time that elapsed between sending the test message and receiving the result. Based on the comparison between the actual data and the expected data described in the script file, the engine generates an error report which is then rendered into an HTML page.

In many instances, the test data generator and the verifier communicate with each other to exchange information such as timing or unique keys that help the verifier identify the correct response message.

Design for Testability

As discussed in the previous sections, testing EAI solutions requires sophisticated tools and techniques. Nevertheless, tools and techniques alone are not sufficient to test a poorly designed integration solution. We can streamline our testing efforts significantly if we include testability into the development process as a non-functional requirement.

When it comes to design for testability, the software world can learn a fair bit from microchip manufacturers. The manufacturing process for microchips is very complicated and can introduce a variety of errors into a chip. Once delivered and built into a device, there is little we can do to fix a defective chip. Therefore, chip manufacturers design dedicated test interfaces into the chips that can be exercised before the chip is shipped. These special interfaces give the test equipment access to many internal functions that may be hidden from the outside interface. This makes testing more efficient because it can occur at a finer level of granularity.

Translating this approach into the world of enterprise integration, we can design our components to support additional test interfaces. For example, we can design an application adapter to support additional transactions that retrieve the internal state of an application. Thus, we can design test cases that account for an application’s internal state without having to test directly against the application’s database or API. We can also design business processes to support special functions to query the internal process state or allow us to inject specific test cases into the process. For example, we define a special message that triggers the ‘three days passed’ action without having to wait three days.

Besides adding test interfaces, basic design guidelines can also help us increase the testability of an application significantly. For example, a set of small, loosely coupled components can enhance testability enormously. Loosely coupled components interact with other components via messages as opposed to direct interfaces. This allows us to stub out any external dependencies and simulate a variety of responses to execute all possible execution paths within the component under test. Using small components reduces the amount of uncertainty when testing a component. Smaller size means easier analysis and debugging as well. Also, in the case of stateful components such as process definitions, the number of possible internal

Copyright ThoughtWorks, Inc. 2002

states can increase exponentially with the size of the component. Often times, cutting a larger component into two smaller components can cut the total number of required test cases in half.

We must not forget, though, that testability is only one of multiple non-functional requirements. Taken to an extreme, design for testability can conflict with other non-functional requirements like performance and maintainability. Creating a large number of small components can lead to network flooding due to the number of messages being passed around between the components. Also, maintaining a huge collection of components may turn out to be more difficult than a smaller number of medium-sized components. As always, architecture is all about making trade-offs and often times a happy medium is the correct answer.

Another good design principle to increase the testability of components is to design narrow, well-defined interfaces. Narrow interfaces reduce the number of possible test scenarios and aid in the automatic creation of test cases. Rather than creating an interface with a generic 'DoTransaction' function, we prefer to create a specific, more narrowly defined function such as 'CancelOrder'. In some cases, the design of narrow, specific functions seems to contradict the desirable principle of loose coupling. A generic function, however, may reduce the syntactic coupling (I can pretty much do anything with a DoTransaction function), but it does not reduce the semantic coupling (what am I actually doing?). Worse yet, such generic functions disguise the semantics and make the creation of meaningful test cases very difficult.

Design for testability does not only relate to component design, but also to the development and deployment tools. Many integration tools feature graphical editors that allow the user to design data transformations and business process definitions. While graphical tools can be great design aids, they generally are not very well suited to testing and deployment tasks. While designs favor rapid iteration and visual communication, testing and deployment are all about repeatability and no need for human intervention. Good integration suites allow the user to design test scenarios and deployment rules using graphical editors but create executable scripts so that these procedures can be executed in a repeatable fashion once they have been developed.

Web Services and Service-Oriented Architectures

It is difficult these days to talk about enterprise integration without talking about Web services. Web services are a reality and the trend to service-oriented architectures will influence how we architect integration solutions. How does this trend affect testing?

Web-services based solutions share a number of characteristics with EAI solutions. They consist of distributed components or applications, developed in different languages and running on multiple platforms, connected by a synchronous or asynchronous messaging infrastructure. As a result, many guidelines for testing application adapters and transformations apply to Web services-based solutions.

The trend to service-oriented architectures brings some new aspects to testing. Service-oriented architectures are based on the definition of a set of well-defined services that are provided by various packaged and custom applications. New applications are then developed on top of these services (see Figure 10).

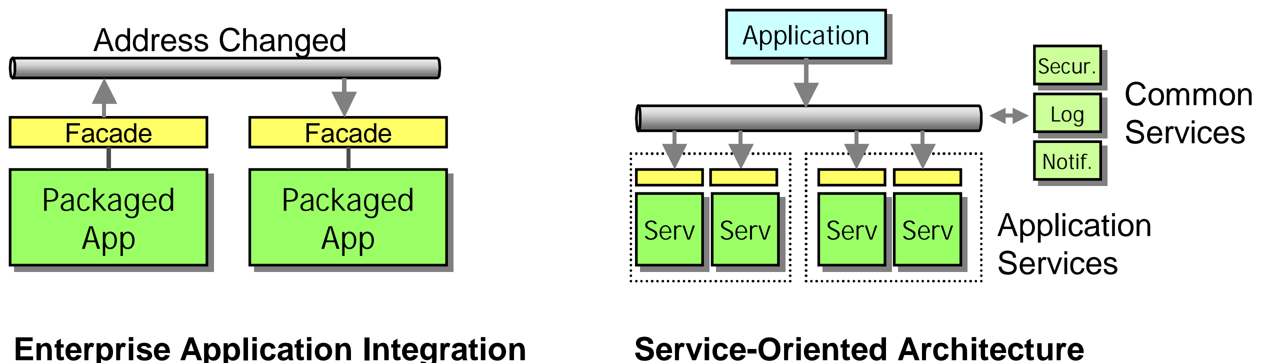


Figure 10: EAI vs. SOA

Copyright ThoughtWorks, Inc. 2002

In a real-life solution we can be certain to see a fair amount of overlap between these two ideals, but the underlying principles are important to consider because they lead us to some of the important testability considerations when developing SOAs. In an EAI solution we generally design two ends of a data interchange concurrently. Therefore, we can design, test and debug adapters and transformations between two specific applications. In an SOA solution, we generally do not have this luxury. We cannot anticipate the usage of the services we provide since the goal of the SOA is to allow the creation of new, complex applications on top of these services. This requires us to be even more stringent in specifying exact interfaces for our services and to test for all possible scenarios.

One advantage that Web services-based architectures are likely to bring is the availability of commercial testing tools. Since Web-services based solutions can rely on a set of standards for transport and interface definitions (SOAP and WSDL) plus a list of associated standards, it is more lucrative for a testing tool vendor to address Web services as opposed to the proprietary API of an EAI suite. We are already starting to see commercial testing tools that focus on automated test case generation from WSDL files.

Non-Functional Testing

So far, we have focused primarily on functional testing. Given that an integration solution forms the backbone of the enterprise, verifying non-functional requirements such as scalability, robustness and fail-over are paramount. In many instances we can leverage the investment in an automated functional test framework to support non-functional tests.

For example, we can use test data generators on a larger scale to generate load on the system for load-testing and scalability tests. Basically, we need a set of efficient test data generators that can be run from an array of test machines to generate sufficient load on the integration solution. We can modify test data verifiers to collect statistics such as message transmission time or latency. We can choose to verify the actual outcomes to detect erratic behavior under high load or we can decide to ignore the outcome because we are simply interested in maximum throughput.

Fail-over situations require new forms of test data generators. These test cases verify that a system can successfully reroute messages to alternative transport channels and processing units when a massive failure occurs on one unit (e.g., hardware failure, network failure or system crash). In order to test this behavior we need to be able to simulate these failures, i.e. we need to be able to simulate a network failure. It is fair to say that in most cases these types of tests cannot be 100% automated and may still require some amount of human participation (e.g., to unplug a network cable).

Conclusion

Test-driven development is an important technique to delivering high quality applications and to enable an agile, iterative style of development. The same holds true for application development as well as integration development. Enterprise integration solutions present us a series of additional challenges such as a heterogeneous environment linked via asynchronous messaging. As a result, the availability of a testing framework is even more crucial to simplify and automate the testing of these solutions.

The framework presented in this paper has been used successfully in integration projects. As integration solutions continue to evolve -- particularly with the advent of Web services and service-oriented architectures – our needs for testing are likely to evolve as well. Following the concept of testability we will continue to enhance the existing framework as integration technologies and our understanding of testing in integration solutions evolve.

Copyright ThoughtWorks, Inc. 2002