Compliments of

Service Virtualization

for dummies®
A Wiley Brand

2nd IBM Limited Edition

Define service virtualization

Improve your traditional approach to testing

Deliver faster, high-quality software

Judith Hurwitz

To learn even more about service virtualization and how IBM can help you deliver higher quality software faster, visit **http://ibm.co/servicevirtualization.**

Service Virtualization

2nd IBM Limited Edition

by Judith Hurwitz

for dummies®
A Wiley Brand

Service Virtualization For Dummies®, 2nd IBM Limited Edition

Published by
John Wiley & Sons, Inc.
111 River St.
Hoboken, NJ 07030-5774
www.wiley.com

Published by John Wiley & Sons, Inc., Hoboken, New Jersey

ISBN 978-1-119-41593-0 (pbk); ISBN 978-1-119-41591-6 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Publisher's Acknowledgments

Some of the people who helped bring this book to market include the following:

Project Editor: Carrie A. Burchfield
Editorial Manager: Rev Mengle
Business Development Representative: Sue Blessing
Production Editor: Tamilmani Varadharaj

Table of Contents

Introduction

Welcome to *Service Virtualization For Dummies*, 2nd IBM Limited Edition. Service virtualization helps companies create more efficient testing environments by eliminating many of the roadblocks that testing teams typically encounter. While testing teams want to test early in the application development process, it's hard to make this plan a reality based on the increasing complexity of software environments. In order to reduce project risk and guarantee higher quality outcomes, your company needs a new proactive approach to testing. You need an approach that improves the overall level of testing and increases the efficiency of removing defects.

Your company can benefit from service virtualization if your teams develop and deliver complex applications with multiple dependent components that must be tested. Instead of waiting for dependent services to become available for testing, your teams can use service virtualization to emulate these missing elements. With service virtualization your test environments can use virtual services in lieu of the production services, increasing the frequency of integration testing. As a result, deploying service virtualization can help you decrease testing costs, improve team productivity, and ultimately improve software quality.

About This Book

This book gives you insight into what it means to leverage service virtualization in your testing environments. By simulating service components, you can quickly validate the behavior and performance of an application's components and determine how they interact. In this book, you discover the key challenges that companies face when developing complex applications with multiple dependencies and how you can increase test team efficiency with service virtualization to enable more sophisticated and accurate testing earlier in the life cycle.

Icons Used in This Book

The following icons are used to point out important information throughout the book:

TIP

Tips help identify information that needs special attention.

WARNING

Pay attention to these common pitfalls of managing your foundational cloud.

REMEMBER

This icon highlights important information that you should remember.

TECHNICAL STUFF

This icon contains tidbits for the more technically inclined.

IN THIS CHAPTER

- Introducing service virtualization
- Seeing service virtualization in action
- Discovering how service virtualization differs from other types of virtualization
- Understanding where service virtualization adds value
- Realizing the benefits of service virtualization

Chapter 1
What is Service Virtualization?

Imagine a world where software development teams consistently deliver new applications on time, under budget, and with exceptional quality and performance. For many development and operations teams, the demands of testing today's complex applications in their test environments prevents this goal from becoming an achievable reality.

In this chapter, I introduce a new technology called *service virtualization* to the development and testing communities and talk about how companies are using it as a key part of their testing strategy to reduce risk, decrease testing costs, and deliver higher-quality software. Service virtualization helps organizations overcome many of the challenges associated with testing today's complex and interdependent systems. Because the term *virtualization* is quite popular in different circles, I describe how service virtualization is different from other kinds of virtualization. You take a look at situations where service virtualization can add the most value and dive into the various uses of service virtualization and its key benefits.

Defining Service Virtualization

Service virtualization simulates the behavior of select components within an application to enable end-to-end testing of the application as a whole. Test environments can use virtual services in lieu of the production services to conduct integration testing earlier in the development process. Service virtualization can be useful for anyone involved in developing and delivering software applications. Integration testing of these applications is often delayed because some of the components the application depends on aren't available. Service virtualization enables earlier and more frequent integration testing by emulating the unavailable component dependencies.

REMEMBER

Service virtualization solutions have the following characteristics:

- **Application emulation:** Virtual components can simulate the behavior of an entire application or a specific component.
- **Multiple test environments:** Developers and quality professionals may create test environments by using virtual components configured for their needs.
- **Same testing tools:** Developers and quality professionals can use the same testing tools that they have used in the past — the tools can't tell the difference between a real system and a virtual service.

TIP

These virtual components are created to simulate a real environment through two basic entry points:

- **Observing the system in action:** Construct a virtual component by listening to the network traffic of the service that you want to emulate.
- **Reading the descriptions of the system:** Construct a virtual component by utilizing other sources of information such as service specifications. An example is a Web Services Description Language (WSDL) file, which describes the operations offered by a service along with the parameters it expects and the data it returns.

Service Virtualization in Action

One of the best ways to understand the benefits of service virtualization is to look at examples. In this section, I give you a make-believe example and a real customer scenario.

The Whiz Bang International example

The URGoodForIt Credit Check service (a make-believe service; good name, huh?), provided by a third-party vendor, must be deployed to test the new application. However, it isn't readily available in the test environment. The team can't begin testing without this dependent component. As a result the team is forced to choose between de-scoping tests or slipping the delivery schedule.

Figure 1-1 depicts a sample online ordering application that implements the URGoodForIt Credit Check service. Whiz Bang International has embraced Service-Oriented Architecture (SOA), and the implementation of this application takes advantage of a variety of services such as an ordering handling service, a third-party credit-checking service, a third-party payment service, a custom service to provision a new device, and a database. This complete picture of the system reflects the production environment without service virtualization.

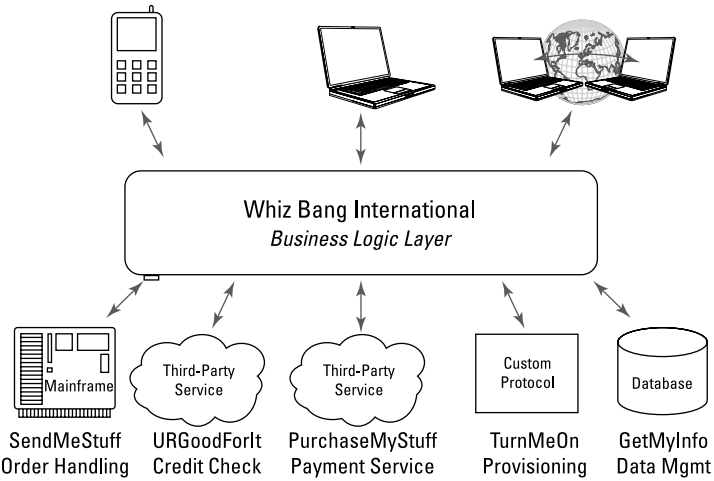

FIGURE 1-1: A commercial application in production without service virtualization.

The URGoodForIt service is a good candidate for service virtualization because

- Test environment availability is delayed and the team has to wait for the service to be available before testing can begin.
- The URGoodForIt service costs money each time it's executed.

  The team needs to test at user levels of 100,000 for performance purposes. Because URGoodForIt is provided by a third party, the business needs to pay a fee per use each time the service is called in a test. The fees for performance testing with 100,000 users add up very quickly.

The fact that this dependent service is unavailable for testing creates a testing bottleneck for the whole team, and to test (function or performance) end to end, you can't begin testing until you have all the required pieces. Virtualizing the unavailable service unblocks the team. Illustrated in Figure 1-2, a production component from Figure 1-1 is replaced with a virtual component.

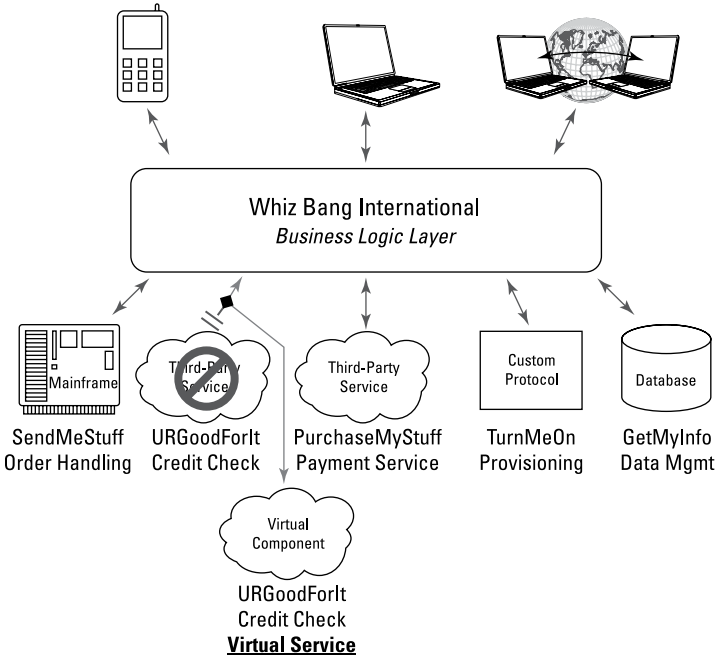

FIGURE 1-2: Service virtualization makes the unavailable available.

REMEMBER

A good service virtualization solution makes it easy to create a virtual component that

- Mimics the behavior of the real component providing the service
- Responds with realistic data
- Processes requests within configurable throughput ranges
- Can be turned on or off, as the real service becomes available, without having to reconfigure the deployed application

Of course, at some point in time you're going to need to test your system against a real production application. Service virtualization isn't a substitute for testing the actual source code deployed as the composite application. Meaning, you're not going to bring your software to market without real end-to-end testing. The idea behind service virtualization is to catch the majority of defects *much earlier* in the process when they're easier and cheaper to fix. You may still discover errors during your end-to-end testing, but they're likely to be fewer in number, and these bugs can probably only be discovered when the complete application is tested by using the real components.

A large financial services organization

In this section, you take a look at a real-world example. (I removed the company name for privacy reasons. I hope you understand!) A large financial services organization set out to test its new billing application, including all its integrations with internal legacy applications and external third-party dependencies. The integrations between the different application components needed to be tested continuously throughout the development process.

For example, the interfaces between the billing application and the ordering application needed to be independently tested. Each time a required test environment was configured, long delays occurred because at least one of the required application components wasn't available. As a result, the team faced an end-of-cycle test crunch. IT requested additional resources to execute the tests, and because some of the test suites contained thousands of individual tests, testing delays put the entire project at risk for missing important completion deadlines.

The team decided to adopt service virtualization, and it was able to create a major turnaround in the testing economics for the new billing application. The quality professionals created virtual components emulating aspects of the real-world environment by recording messages and responses in that environment. This virtualized environment had the behavior of the live application, but the effort to create the test environments was significantly reduced, so developers and quality professionals didn't have to use the actual dependent applications when testing their changes. Instead, they used virtual components to perform integration and performance testing of the components that didn't change in combination with the components, which they modified or developed new.

The result? The company reduced the time — from a few weeks to just minutes — to stand up its test environments. The time savings resulted in sharp reductions in testing costs and allowed the professionals to do a more thorough job of testing and validating software quality.

Seeing How Service Virtualization Differs from Other Types of Virtualization

When people hear the term *virtualization,* they often automatically think of "virtual machines" or "hardware virtualization." In fact, the term *virtualization* can be applied to many aspects of computing, such as servers, applications, network, or storage. In general, virtualization means using computer resources to imitate other computer resources.

In hardware virtualization, for example, one physical server is partitioned into multiple virtual servers. The virtualization software enables each virtual machine to present the appearance of dedicated hardware. This can help reduce hardware costs, but there are other costs associated with deploying hardware virtualization and creating virtual machine images.

Utilizing virtual machines as staging environments for testing has gotten a lot of press recently because they can provide a good representation of what's going into production with a lower infrastructure footprint than full physical pre-production environments. The IT or testing organization can create virtual images of the production environment to run on virtual machines without some of the manual effort or cost required to provision and build a physical pre-production server. This can give the testing group confidence because theoretically the virtual image is very close to the real thing. The downside is that the creation and ongoing management of those images (for example, tracking license usage, installing the OS, and keeping it up to date) can still add significant cost to a project. And you still have to wait for and deploy every component needed by your application.

In service virtualization, however, *software components* are virtualized by emulating their service interface and mimicking the component's behaviors. Service virtualization focuses on emulating only what's needed by your test environment and, compared with hardware virtualization, eliminates the additional effort to license, configure, and run all the other bits required on a virtual machine (for example, the operating system). There is, of course, some investment to create virtual components for your services, but the virtual components are available throughout the application life cycle and have a very small footprint (much, much smaller than a virtual machine). They also are easier to share and faster to deploy because they're hosted on a server optimized for this purpose.

Exploring Where Service Virtualization Can Add Value

Service virtualization can dramatically change the economics and flow of the entire application development process. Application quality is everyone's responsibility — from development to testing to deployment — so service virtualization can be used by the entire team throughout the application life cycle.

A key benefit of service virtualization is that you can do testing much earlier in the application development process by reducing test bottlenecks. However, you may miss out on many additional benefits of service virtualization if you only use this technology in your formal testing process.

Testing

One of the most important principles adhered to by successful testing teams is "Test early and often in the development process." Why? Because the earlier you find and isolate defects, the better your ability to fix them. Now, I realize testing early and often is simple to say, but it's not so easy to execute. The longer it takes to begin testing or "stand up" your test environments, the less likely you are to test at the right time to achieve high-quality outcomes. Service virtualization can help reduce testing costs and speed up testing start and execution times.

Service virtualization delivers benefits for all types of testing including functional (manual and automated), integration, and performance testing. Take performance testing as one example. The performance of today's composite applications is really the sum of the individual application component's capability to respond in a timely manner. Degradation in the response time of any application components can slow performance of the application, negatively impacting user experience. However, if you defer performance testing until the entire application is deployed (the traditional approach), it becomes much harder to identify the root cause of slow performance and poor user experience.

Performance testing using a virtual component (or a set of virtual components) enables you to fix the most obvious problems earlier. Testing in the traditional way, where everyone builds a piece and you test when everything is ready, you may never meet your performance goal. You won't know until the end of the development and testing cycle that there's one component, for example, that's taking too long to respond, adversely impacting the user experience. I give you a more detailed view of this process in Chapter 4.


TIP

Make sure you understand customer or end-user expectations right from the beginning of the development process, including the functionality, usability, reliability, supportability, and performance, and the business logic involved. You need to do the math and determine the individual performance specifications that

each component must have. With that understanding, you can simulate the exact conditions for the development team from day one in your virtual test environment.

Development

Why does a developer need service virtualization? In addition to the formal testing process managed by the testing team, developers should be testing their own code all the time. Service virtualization can be used to simulate any environment required by developers for testing, going beyond compilation and unit testing, while they're writing code.

Service virtualization eliminates the need for developers to manually write their own simulation stubs or mocks (for example, fake objects that try to mimic real objects in testing). These stubs may require future maintenance, which will take time away from developers writing new functionality for the business. Removing the need to manually write stubs can be especially helpful in agile development or iterative development environments where teams wish to conduct continuous testing of new functionality throughout the development process.

Non-production usage

Service virtualization can also be used to create a realistic environment for training without the need to connect with your live production environment resources. To understand how service virtualization can help create more effective training programs, this section gives you a scenario.

DON'T GRADE YOUR OWN WORK!

Having developers write their own testing simulations or stubs to validate their own code changes or new code is a bad idea! It's like having a student mark his own homework or exam. Testers, who know how the functionality should be tested and the data required to properly test the various scenarios, can easily create virtual components and tests, making them available to the development team.

A company wants to train its newly hired customer service agents on how to use its Customer Relationship Management (CRM) system. These agents are given access to a live version of the CRM system to help simulate real situations encountered on the job. However, the CRM system is connected to various back-end systems. To answer a series of exam questions, the agents need to create queries about product and pricing information managed by several back-end systems. With service virtualization, you can emulate the back-end systems so CRUD (create, read, update, and delete) transactions can occur without interfering with the live production systems.


REMEMBER

Beyond isolating the production from the trainees, service virtualization makes it easy to reset the training environment to a known state with known data. Achieving this with service virtualization can be significantly less expensive than deploying the complete system to an isolated training environment.

Benefits of Service Virtualization

Service virtualization enables earlier and more parallel, continuous testing of complex applications across the development life cycle. It can be especially useful in applications consisting of interconnected services in an SOA environment, where testing is often delayed waiting for all the services to be ready and deployed. As a result, service virtualization is likely to deliver benefits explained in this section.

Reducing costs

Test lab infrastructure costs can be pricey. Instead of provisioning large servers or mainframes, a virtual test environment can run on low-cost commodity hardware. The environment can easily be reconfigured for different testing needs or projects.

Costly crunch time end-to-end testing can also be reduced with service virtualization because functional, integration, and business process level tests have been executed many times before and the majority of the defects should've been found. The fact that you've done these tests throughout the development life cycle can help drive down the time required to perform the full end-to-end test and eliminate the need to bring on additional testing staff.

Many of today's composite applications utilize services provided by a third party. These third-party vendors may assess a charge each time that service is executed. Virtualizing those services for the purposes of testing can decrease the third-party access fees from the testing budget without limiting testing activities. It also solves another common problem with third-party services: The provider may not make the services available for testing when you need them. Service virtualization makes them available whenever you need them.

Improving productivity

Constraints on developers and quality engineers can limit productivity. In a physical test environment, you have constraints:

- Time to provision environments
- Time when environments are unavailable to developers and testers
- Limitations on the amount of test cases that can be put through a system

With service virtualization you don't have restraints in the way you do testing or development. Virtual components are available 24/7. This means that productivity can be greatly increased, and resources can be freed up for other value add activities or additional testing process improvements such as the inclusion of exploratory testing.

Reducing risk

Service virtualization can also help reduce risk. You can test software earlier in the process, which means defects can be addressed earlier, producing fewer surprises toward the end of the schedule. The final product may be put into production earlier and with fewer errors. Additionally, large teams can effectively work in parallel, collaborating on different parts of an application by virtualizing components of the complete system, and put a plan in place to ensure that the system is tested properly and the piece parts work together.

Increasing quality

Service virtualization can improve the overall quality of the application because it increases the efficiency of any testing being performed. As a result, teams are able to do a more thorough job of testing their applications and get higher quality software to market faster.

IN THIS CHAPTER

- Encountering the expectations of enterprise applications
- Wrapping your head around service-oriented architectures
- Taking advantage of mobile capabilities
- Responding to changing customer expectations

Chapter 2
The Driving Forces of Change

Information Technology (IT) departments have always had to choose between how much time and resources to devote to quality management practices (often dominated by testing) versus the risk of delivering poor quality applications. In the past, the IT department has often managed decisions about the quality process with limited involvement from the business stakeholders. However, as software has become an integral part of a company's strategy and persona in the marketplace, testing can no longer be treated as a standalone activity. In many situations, the quality of the software is an important way that customers, partners, and suppliers measure a company and define its success in the marketplace. It's no wonder that software quality management has emerged as a foundational element of the enterprise application life cycle.

In this chapter, you look at how the relationship between IT and the business has changed, where IT no longer merely supports the business but takes a leading role in creating value. You also see the enterprise architectures that have emerged to address these business pressures and dynamic landscape, the opportunities and challenges presented by the emergence of new application

delivery channels, and how the IT organization itself — from development processes to operations — has evolved. The need to improve software quality is a constant undercurrent that requires organizations to rethink their traditional approaches to testing in order to be successful in this environment of rapid business and technical change.

Meeting the Rising Expectations of Enterprise Applications

As a business leader you need to keep focused on your business strategy — whether your objective is outsmarting your competition, growing your business, or keeping your costs under control for maximum profitability. Meeting these objectives in today's dynamic consumer-focused environment requires a high level of collaboration between business and IT. In the past, IT focused primarily on developing applications designed to meet the needs of internal departments. However, IT has seen two changes in recent years:

- The expectations of users have increased based on their experience with consumer technology.
- The role of IT has grown to include developing and deploying new, innovative externally facing applications.

Today these applications are required to develop new partnership channels or to enhance the way customers interact with the business. In other words, IT has moved from supporting the business strategy to becoming a critical part of the business strategy. Business leaders expect the IT organization to meet changing requirements and deliver on the business objectives — quickly. They are no longer willing to accept that technology will be an impediment to change.

Because of this deep relationship between business strategy and IT innovation, you need to understand some of the key business pressures companies are facing. Applications designed to support new business models need to process large and diverse volumes of data, and they need to integrate with a broad range of new and legacy systems. Companies also need to streamline and automate

manual processes so they can rapidly develop and deploy high-quality applications.

To support this new collaborative environment between business and IT, many companies are incorporating new technologies to improve responsiveness and user value, including

- Modernization of traditional back-end systems to deliver new functionality to the marketplace
- Broader use of modular, shared services for the rapid assembly of new composite applications
- Increased support for new application delivery channels, particularly mobile devices
- Cloud computing for increased business flexibility and scalability

Embracing Service-Oriented Architectures

Enterprise architectures have evolved to keep up with the fast pace of business innovation and growth. Many companies find that by implementing a service-oriented approach they're better positioned to support the business requirements for flexibility and scalability.

TIP

A service-oriented approach delivers other benefits, as well, including the following:

- Increased ability to create more sophisticated applications by combining reusable modular business services
- Improvements in IT responsiveness and performance
- Ability to exchange data with outside organizations, for example suppliers and partners
- Flexibility to consume services from third-party suppliers
- Support for a variety of deployment topologies including using a public or private cloud
- Increased standardization in the IT environment

Service orientation is an architectural approach based on implementing business processes as software services. These business services consist of a set of loosely coupled components — designed to manage dependencies and foster reuse — assembled to implement a well-defined business task. Designing systems with modular business services results in more efficient and flexible IT systems. Service orientation is also a *business* approach and methodology that helps businesses scale and adapt to changing market forces.

The key characteristics of service orientation are modularity, reusability, and flexibility:

- **Modularity:** Moving from large monolithic, complex, and unmanageable applications to componentized reusable business services
- **Reusability:** The rules and logic of application components that are common to key business processes and encapsulated to create a reusable business service

  Using a tested and proven component speeds development, enables a higher level of security and trust, reduces risk, and saves money.
- **Flexibility:** A function of the modularity and reuse of business services

Service-oriented architectures have led to an entire industry that provides businesses with well-designed business services that handle everything from payment services to credit check and inventory availability. This creates new alternatives to developing and deploying the underlying software in-house. However, it creates new challenges to managing releases and ensuring quality.

The Rise of Mobile Applications

The entire life cycle of application development and deployment is changing because of changes in consumer expectations and platform requirements. This major shift in IT and business happens due to the following factors:

- The demand for mobile applications
- The need to integrate those new "systems of engagement" with existing back-end "systems of record"
- The opportunity to innovate by taking advantage of unique mobile capabilities (such as location awareness)

Today, customers expect to be able to interact with a company and its services in a variety of ways, including the ability to access information from a laptop, tablet, or smartphone. Each of these devices is powered using one of a variety of operating systems and comes in several form factors. Increasingly, the end-user is in control of what platform they select to interact with your organization. Development and testing require new approaches to support the wide range of customer devices.

These emerging applications don't execute in isolation; they must connect with existing back-end systems. Customer-facing applications have to be implemented with the right performance level and the right quality level based on the right business process. This may place new demands on the supporting systems, which must be verified.

Mobile devices offer new unique capabilities but also have some constraints. Features such as location awareness, voice-based interfaces, and near field communication open new avenues of innovation. However, compared to modern PCs, small screens, limited bandwidth, and high network latency are back. Not to mention significant variability based on device, carrier, and location. Given the pace of innovation in this area, businesses can't anticipate how their customers' requirements may change in the future. Businesses have to make sure that their approaches to development, testing, and deployment can keep pace as preferred models for interaction continue to evolve.

Agile Transformation Continues

Many software development teams are relying on agile development approaches to speed up the development process. Accordingly, the testing process needs to speed up as well if the new applications are to be introduced to the market quickly with increasing quality. Agile development processes focus on short

development iterations that include continuous planning, testing, and integration. The goal is to keep the project moving forward at a fast pace by leveraging a highly collaborative environment.

This approach becomes impractical if testing lags behind development. Unfortunately, this lag in testing is a common occurrence. For example, the testing team may need to spend many hours each night manually resetting the test environment. Additionally, application testing may be delayed while the testing team waits for dependent software to become available. Other delays in testing occur when the duration of a manual test execution cycle exceeds the length of a development sprint. If the testing team is not able to adapt to the rhythm of an agile development approach, IT is likely to encounter numerous delays and miss its application delivery deadlines.

Just a few years ago, the typical application was changed only a few times a year. Today, software development, deployment, and operations environments face constant change. It is quite common for a single application to be changed on a weekly or sometimes daily basis. It is no longer possible for software development and operations (production) to act as independent organizations each with their individual tasks, deployment procedures, and schedules.

IN THIS CHAPTER

- Increasing quality in the application life cycle
- Taking another look at test automation
- Looking at the challenges of complex test environments
- Understanding how service virtualization can help

Chapter 3
Escaping the Past

As the complexity of applications increases, with more interfaces and delivery options, continually improving software quality management practices becomes more important than ever before. However, while companies are attempting to become more nimble and responsive to market demands, testing often struggles to keep pace, creating a bottleneck in the overall software delivery process.

In this chapter, I introduce approaches for companies to improve their quality management processes. I introduce you to some best practices that companies can put in place to become more quality focused and sophisticated in their testing practices. Because application and testing environments are becoming more complex, service virtualization addresses key challenges of these complex testing environments and increases test team efficiency by enabling more sophisticated and accurate testing earlier in the life cycle.

Improving Quality in the Application Life Cycle

Software quality is a costly problem in virtually all industries. Fixing software issues costs billions of dollars each year. The problem seems to be that many organizations don't realize that

the activity of testing is only one part of delivering high-quality software. In order to deliver reliable, usable, available, maintainable, and scalable software that addresses business objectives, there needs to be improved planning, collaboration, traceability, and information accuracy throughout the application life cycle — from requirements to deployment.

Too often, organizations are reactive in their approaches to improving quality instead of implementing a proactive and optimized quality process based on understanding changing requirements and business risks. Collaboration and traceability allow teams to be proactive by having insight into what components of the application have changed during the development effort. This allows teams to focus testing on the specific areas of the application that have changed and minimize risk to the business.

Even with the best quality management practices, software (and hardware) can still be released into the marketplace with some defects — many of which may go undetected prior to release. In today's world of accelerated software delivery, you can't test every code path and condition that a piece of software may encounter, and let's face it, today's users have changed — they're tolerant (to a point) of some initial defects, but they expect these defects to be resolved quickly through frequent releases.

To meet the demands of today's end-user, the secret is to have the necessary processes in place that allow for earlier detection, isolation, and remediation of defects. And for those issues that do escape into production, a strategy that addresses defects quickly and gets those fixes to market faster is a must.

REMEMBER

Service virtualization can help you improve your quality management processes because it

- **Allows for earlier integration testing:** Virtual components can simulate service interfaces that the system under test (SUT) needs to call on. You don't wait until late in the development life cycle to test the interfaces; you use a virtual component(s) to test sooner. In fact, service virtualization makes real continuous testing part of the regular build process.
- **Accelerates test environment availability while lowering costs:** You can have a ready-made test environment in

relatively short order. Using virtual components to emulate dependent environments allows testing to begin without further delay. The cost of test environments — hardware, software, and labor — is reduced.

- **Enables development to test earlier in the process:** Developers as well as Quality Assurance (QA) professionals can make use of a shared set of virtual components to test integrations earlier in the process, perform parallel development, and deliver higher overall product quality.

Rethinking Test Automation

Test automation has been done the same way for many years, typically involving a user interface (UI) based approach:

1. **Wait for the UI to be stabilized.**
2. **Build up the test environment by deploying all the components of the application once they're ready.**
3. **Record user interaction via the UI.**
4. **Tweak the recordings, if needed, to improve test scenarios.**
5. **Execute tests.**
6. **Reset the environment and rerun, hoping you don't need to do a lot of tweaking or rerecord.**
7. **Maintain a library of test scripts as the application changes with each iteration, often by rerecording whole scenarios.**

WARNING

UI test automation is faster to run than manual testing, so it can be done more frequently. However, UI testing tends be fragile — changes to the code often break tests, even when those changes aren't visible. This problem can be worse for scripts created by recording user interactions. The trade-off made to mitigate the cost of ongoing test maintenance is to wait until changes to the UI are complete. Of course this introduces the risk that problems aren't found early enough in the development life cycle to be fixed within the project's original schedule.

The UI is also like the tip of an iceberg — the majority of code and complexity hides below the application's interface. The most direct way to identify the root cause of a defect is to find it close to where it was introduced, without other layers of the application potentially masking what really went wrong (for example, by not showing exceptions thrown by code to the user). This requires taking a broader approach to automating test cases. Consider, for example, testing each layer of the architecture independently.

Testing at the service or Application Programming Interface (API) layer, the layer where components and applications "talk" to each other, can improve testing efficiency and reduce business risk because:

- This is where applications often break — at the interconnection points between subsystems.
- These boundaries often correspond with organizational and schedule boundaries, so fixing problems here may be difficult or expensive, especially if found late.
- Service interfaces are, by nature, more stable than user interfaces because many applications depend on the same service specifications. Changes are typically well managed between all stakeholders to avoid breakage in production. As a result, automated tests at this layer require less maintenance.

A natural synergy exists between service virtualization and automating tests of service interfaces. The tests drive a particular service interface by generating requests and validating responses. Virtual components receive these requests, emulate the real-world service's behavior, and provide the appropriate responses on any number of supported protocols and in a variety of message formats. Tools take advantage of this synergy by sharing protocol definitions and data sets between virtual services and automated service tests. In fact, one best practice is to create a test suite against your service interfaces that can be used to validate both the production component and the virtual component.

Facing the Challenges of Complex Test Environments

Creating test environments that host today's complex applications can be difficult, especially in the case where the test application interfaces with other internal and possibly external systems. Other challenges include the following:

- It can be expensive and time consuming to replicate an entire production environment for the purpose of testing. For example, a typical production server running Windows Server 2012 can cost tens of thousands of dollars or more depending on configuration. Capital costs can add up quickly if many servers are needed.
- It requires a lot of knowledge and technical skills to create these environments. Configuring a test environment can require application-specific, as well as system administrative, expertise. Consider, for example, an application that interfaces to an ERP system (SAP, Oracle, Siebel, or the like). The application test team may not know how to deploy an instance of the ERP system for testing.
- It can often be costly and difficult to schedule time to test in cases where third-party services are involved.

Some advanced developers may have tried to address the lack of a complete test environment by creating their own ad-hoc "stubs" or "mocks." This approach may aid the developer with unit testing, but rarely does it scale to support the entire team, for a few reasons:

- Developing a realistic simulation to support all test cases and test purposes is complicated and can quickly become a major development and maintenance effort, diverting development time away from the application.
- Developers often need to change the underlying application to use the mocks in place of the real components, diverging the application under test from the one being readied for production.
- No infrastructure exists to support sharing these stubs across the team.
- Problems can easily be missed when testing relies on the same developer mindset that created the code.

Service Virtualization and Complex Test Environments

Service virtualization is a technology that can help your organization become more efficient and quality focused in the face of ever increasing complexity. With service virtualization, developers and testers create virtual components that can be shared, enabling parallel development across the team. And because virtual components emulate real-world services, applications, or entire systems, they can help to remove delays in the testing process. These components also run on commodity hardware and decrease the cost of supporting multiple test environments, which can decrease the concerns of operations related to capacity, scalability, and security.

IN THIS CHAPTER

- Identifying services to virtualize
- Automating your tests
- Virtualization for all testing purposes and phases

Chapter 4
Finding Your Way to Service Virtualization

If you've been reading up to this point, you may now understand the benefits of service virtualization and how it fits into the big picture of software quality. You may also be wondering where you should get started. On the surface, it may seem overwhelming. You may be tempted to randomly select one element to virtualize just to see how service virtualization works. However, you can definitely expect a better return on your investment if you take a measured approach to evaluating your testing challenges and let that guide you to make prioritized decisions on what should be virtualized first.

In this chapter, I provide you with a way to analyze the services you should virtualize. I give you more information about how service virtualization and test automation are complementary, and I describe how service virtualization helps throughout all the phases of testing, from the unit test to the user acceptance test and even the performance test.

Identifying Services to Virtualize

To maximize success with service virtualization, you need to identify the right services to virtualize. How do you do this? To begin, bring together the key development stakeholders involved in the application life cycle and start thinking about where your organization experiences the most testing pain(s). Then, ask yourself the following questions:

- Do you have all the environments you need for integration testing?
- Are these test environments available to teams throughout the development cycle?
- Do you experience downtime due to unavailable test environments?
- How often does downtime occur? How long do your teams usually have to wait?
- What's the impact on time and cost due to testing downtime?
- Does your application interface with third-party services?
- Do you need to pay for and schedule access to these third-party interfaces prior to scheduling your tests? How much does this cost?
- Who controls the information needed for creating test environments?
- Do individuals or teams conflict with each other when scheduling the sharing of test environments (or parts of a test environment)?

Your responses to these questions help you prioritize a specific area or areas where service virtualization could help. In fact, I recommend several areas to start:

- **Start with the "low hanging fruit" where you get the most benefit with the least amount of work:** For example, a web service defined in a web service description language (WSDL) that returns data but doesn't allow side effects is an easy service to virtualize. There are no complex state changes to model — just map the input arguments to the data returned in the response. While this service is simple and straightforward to create, it may actually remove

a test environment scheduling issue that could result in project delays.

- **Focus on the conditions that are contributing to the overall cost of testing:** Middleware services are often good candidates for virtualization. If the endpoint is an ERP system, for example, you'll need substantial hardware and labor not to mention a reasonable data set to deploy a copy of the system under test in the test environment. Alternatively, you can simulate that endpoint.
- **Address dependencies on third-party services:** Are you subject to per-use charges when you access certain third-party services? The same principle applies. It might be costly to emulate because it's complex, but it may be critical to do so. This is where doing your homework with a cost benefit analysis pays off.

The cost benefit analysis

REMEMBER

The decision about what to virtualize often boils down to performing a cost benefit analysis. The things that contribute to cost are

- **Impact of unavailability:** Lost team productivity and project delays because dependent services/software weren't available for testing
- **Cost of the skilled resources:** Acquiring and maintaining staff with the necessary expertise required to set up and maintain test environments
- **Underutilized test environments:** Inefficient use of expensive physical hardware, causing unnecessarily high infrastructure costs
- **Cost of licenses:** Software (operating systems, database management systems, and so on) deployed in the physical test lab environment or on a virtual machine
- **Cost of third-party service access fees:** Charges applied when an externally provided service is executed

Adopting service virtualization mitigates these costs and delivers benefits, including infrastructure savings, increased productivity, and faster time to market — each of which contributes to

the overall ROI. You also want to consider more advanced system dynamics when trying to identify what to virtualize.

Service volatility

Service volatility is about how often either the interface or behavior of an application, system, or component changes. Upstream components, the test environment, and tests all need to react to these changes. You need to ensure that the change is handled in a way that doesn't disrupt quality or time to delivery. Consider the following questions:

- Is the dependent service still being developed and subject to ongoing specification changes? You may choose to create a virtual component that offers less effort to modify when change is required but still allows integration testing to proceed.
- Is the dependency on a legacy application that seldom changes? If so, you may create a virtual component to simulate the entire application/system at the service boundary.
- Do your teams require slightly different implementations of the same service? Virtual components allow you to model and simulate specific behaviors to meet the needs of multiple teams and assist with testing during parallel development.

Impact of unavailability

What is the overall impact on the testing effort if a system, application, or component isn't available in the test environment? Some missing components may only impact a few use cases, but others could bring the whole application down and block all testing.

You have several things to consider when evaluating the impact of an unavailable component:

- How many testers will be idle, and for how long?
- What test cases must be de-scoped (or delayed) if all the components aren't available?

All these factors have an associated cost. Virtualize those things that unblock critical testing to reduce these costs.

Cost to deploy or use

Make sure to calculate the cost of having to deploy the system, application, or component. Determine how much it costs to physically create a test environment. Some technologies are certainly more expensive to deploy than others — even in test environments. Or, perhaps you pay each time a third-party service is exercised. Putting a dollar value on the effort to create test environments or execute pay-per-use services could make your decision on what to virtualize very easy.

Complexity of the technology

While there's support for virtualizing a large number of technologies, some technologies may be easier to virtualize than others. In other cases, the technology itself may be easy to virtualize, but the effort to emulate the business behavior of the service may be more complicated. Rate the various technologies in the environment by using scaling factor. The rating process will help you compare the relative complexity and effort required to simulate each dependency.

Doing the math

Planning is the name of the game for service virtualization, and you want to map out your analysis. You may want to capture your details in a table. Table 4-1 contains the set of components for the sample application developed by Whiz Bang International (introduced in Chapter 1). The technology used for each component is listed under the Technology Used column. For instance, the PurchaseStuff component uses SOAP over HTTPS. The impact column describes the impact that an availability constraint or another cost like paying a third-party vendor has on testing the component. For example, it would cost $100,000 to physically stand up an environment in order to test the TurnMeOn Provisioning component. Or perhaps the environment already exists but is unavailable due to scheduling conflicts. The cost of delays in not being able to test is $100,000. The last column is the complexity score for service virtualization. Here's where you rate the complexity of virtualizing a service. You can use a scale of 1–10 or high, medium, and low; whatever you're comfortable with is fine. In this case we're using a scale of 1–10 where 1 is low complexity and 10 is very high complexity.

TABLE 4-1 Components of Sample Whiz Bang International Application

Technology Used	Component/ Functionality	Impact of Unavailability + Cost to Deploy or Use	Complexity Score for Service Virtualization (1–10)
SOAP over HTTPS	URGoodForIt Credit Check Service	$300,000	1
SOAP over HTTPS	PurchaseStuff International Shopping Cart Service	$300,000	4
COBOL Copybook over MQ, 4 different Copybooks	SendMeStuff Order Handling Service	$500,000	3
Custom protocol based on TCP/IP	TurnMeOn Provisioning	$100,000	7

In this case it looks like the URGoodForIt service would be a good candidate to virtualize. Why this one as opposed to the PurchaseStuff component? Both use the same technology: SOAP over HTTPS. Both have an unavailability impact cost of $300,000. However, the PurchaseStuff service is four times more complicated and will take longer to create. If you virtualize the URGoodForIt service, you get the same amount of value ($300,000) with a fraction of the effort. Of course, your numbers and implementations may differ from Table 4-1, but in considering this approach and sample data, I think you'll get the idea of how to go through the decision-making process in terms of what to virtualize.

In general, the more standard a technology or communication protocol is, the more likely the virtualization tools are to support it out of the box. Technologies such as SOAP and XML are almost universal (although there will still be nuances), but sector-specific standards such as SWIFT and FIX are less common. The key point here is that the ROI for each tool will be different depending on whether the protocol is supported out of the box, requires some configuration, or requires an extension to be written to the tool

using an API. Some companies, such as IBM, are investing extensively in technology to allow any protocol to be modeled without the need for custom coding.

Looking into Test Automation Strategies

Many test automation projects focus on automating the user interface (UI). However, this can be problematic, as discussed in Chapter 3.

To remove testing delays associated with recording and exercising fragile user interfaces, consider automating tests at the service or application programming interface (API) layer instead of relying solely on testing at the UI layer. It isn't a coincidence that this is the same layer discussed in the context of virtualization. While service virtualization supports any kind of testing — manual or automated, UI or API, functional or performance — it's also important to recognize the synergies with service testing and the unique advantages of that approach:

- Critical integration points are tested early, even before the UI is available.
- Tests that are less brittle as service interfaces are more stable than user interfaces, especially after they've been deployed into production.
- Tests can act as part of the service contract, increasing clarity across team or organization boundaries.
- The same tests can verify the correctness of virtual components as well as the real implementation.
- Tests generating requests and validating responses are using the same protocols as the virtual components. Therefore, some of the work performed to create virtual components and service tests is the same, and good tools allow you to reuse that work.
- It's easier to isolate the cause of issues found at the service layer because defects are discovered closer to the source of the issue.

Implementing Service Virtualization for All Testing Purposes and Phases

There's no way around it: You're going to find defects when you test. However, when you test later in the application life cycle, odds are these defects may be more expensive to fix. Service virtualization can help you to find errors and issues in all testing phases.

Testing phases

There are various kinds of tests that are part of the application development life cycle. The testing process generally looks something like Figure 4-1.

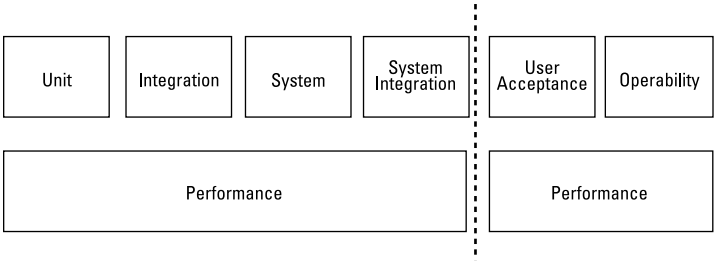

FIGURE 4-1: Progression of testing phases during a project.

Traditionally, testing is done sequentially. Various kinds of tests are deployed in succession, including the following:

- **Unit tests:** Performed by developers to test small pieces of code
- **Integration tests:** Modules of code tested together
- **System tests:** Testing the whole software per system requirements
- **System integration tests:** Testing interactions between systems
- **User acceptance tests:** Testing the system by clients against requirements
- **Operability testing:** Testing the availability of the components that allow the application to run

Quality professionals often wait until System Integration Testing (SIT) or User Acceptance Testing (UAT) to test the whole system. And the reality is that defects will be discovered late in the development process where they're expensive to remediate. Costs associated with finding defects in later test phases can rise by orders of magnitude between each phase. Figure 4-2 illustrates this concept.

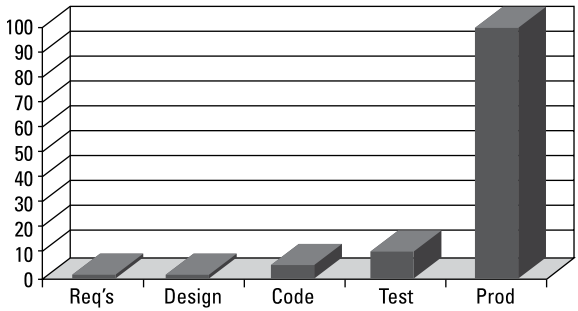

FIGURE 4-2: The high costs of fixing defects during production.

REMEMBER

Service virtualization unblocks end-to-end testing by removing dependencies and enables finding defects earlier through the following ways:

- Developers can begin validating integrations much earlier in the application life cycle, expanding beyond unit testing and increasing the level of testing performed in the development process.
- Testers can begin integration testing earlier and isolate defects to specific areas of the application, decreasing the remediation effort and avoiding the "big bang" integration issues projects are often challenged with.
- The entire development team can benefit by including service virtualization and integration testing as part of the continuous delivery process and getting immediate feedback on the quality of automated builds.

TIP

If a blocking defect is discovered but can't be fixed immediately, consider using service virtualization to simulate the correct functionality, allowing your team to proceed with testing. When the defect is resolved, easily switch back to new source code and continue testing with the actual implementation.

In testing today's composite applications, you're typically validating a larger process made up of many pieces of functionality, which is possible through an increasing level of application interconnectivity and interdependence. As a result, not all of the components are ready when needed, and they're typically brought together for system integration testing or user acceptance testing — near the end of the development effort — where end-to-end testing can really begin.

Performance testing

Performance testing is a key part of a comprehensive testing process. The reality is that teams typically defer performance testing until later testing phases or ignore it all together — a practice that can result in problems in production and damage to a company's reputation. Many performance issues can be associated with flaws in the architecture or application design. Service virtualization can help you discover architectural or design flaws earlier in the process when it's less costly than if you'd discovered them during UAT. Why? Because you can performance test much earlier and simulate conditions that are very difficult to create in normal deployments.

For example, when developing and deploying applications in the cloud, you have little control over network latency. Imagine the insight that could be gained by having the ability to test for conditions requiring a delayed or immediate response. Service virtualization gives you this ability.

I introduce an example application in Chapter 1. The company is Whiz Bang International. If you read that chapter, Whiz Bang International was testing a new application, and the requirement was that 95 percent of all responses needed to be completed in four seconds or less. An illustration of this concept is found in Figure 4-3.

The team already decided to virtualize the URGoodForIt Credit Check service while this service was still under development. The response time of that virtual component was set at one second to reflect the service level agreement (SLA) with the provider. Service virtualization allows performance testing to proceed earlier because the other two components are ready for testing, and the third can be simulated. During this testing, the team discovered that another service (PurchaseMyStuff) had an issue. The team responsible for PurchaseMyStuff had time to improve performance, even before the URGoodForIt team had delivered its implementation.

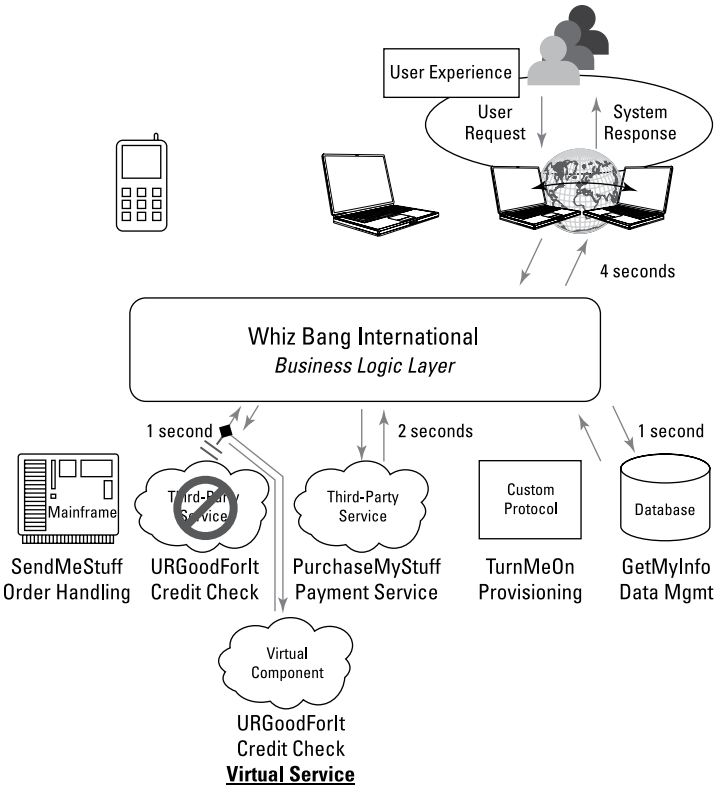

FIGURE 4-3: Measuring the cumulative response time simulating a third-party service.

REMEMBER

Many interesting ways exist to leverage service virtualization in performance testing. These include

- **Model the worst case scenario.** Here you may want to virtualize some back-end services and limit their throughput to see how the overall user experience of your application is impacted under load.
- **Find the next bottleneck.** In this scenario, you may virtualize some services with near unlimited throughput to see which components become the next constraints on overall system performance. This can help evaluate the impact of improving the implementation of a particular service before investing in development.

To ensure that actual systems operate as expected, the real components should be validated. You may turn off all the virtual components in late stage SIT or UAT, but after all the earlier integration testing enabled by service virtualization, there should be few surprises.

Negative testing

Negative testing focuses on how an application or component handles unexpected and error conditions. Examples include supplying invalid inputs (for example, what happens when you try to deposit a negative sum into a bank account) or a service unexpectedly fails (a server goes down). In all cases, the application should handle the error gracefully. However, it can be very difficult to create some of these conditions when testing complex applications. Sometimes it takes a complicated sequence of events and set of test data to produce a particular error. In other cases, you may not be able to put the system into a particular state — such as shutting down a key service — because it would disrupt others who are sharing the same test environment. In addition, you may not know how to reproduce an error — it's an intermittent defect in a particular component — but you want to test that your application can work around it.

Service virtualization makes it easier to support negative testing because you can change the behavior of a virtual component to produce the error condition you want. Different testers or even different test cases can see different behaviors without impacting others by reconfiguring their test environment with different mixes of virtual components.

IN THIS CHAPTER

- Getting a grip on your current architecture
- Understanding the different services to virtualize
- Provisioning virtual components

Chapter 5
Putting Service Virtualization to Work

Service virtualization can enhance and support your company's quality management strategy with all types of testing. While your testing processes are likely to evolve over time, you may begin using service virtualization on an incremental basis so you can realize immediate benefits from this approach. In this chapter, I first describe the steps involved in creating your initial virtual components. Second, because the services you're emulating (and the tests you run) vary in complexity, I provide a range of behavioral models for virtual components. I also share some requirements that a service virtualization solution should meet. Finally, I discuss how you go about deploying and provisioning virtual components and the best practices for running tests. By the end of the chapter, if you read the entire thing, you should have a pretty good idea of what's involved in creating and executing virtual components.

Understanding Your Architecture

Traditionally, when quality professionals test today's complex applications, they treat everything behind the user interface (UI) as a black box. This approach allows the tester to focus on the

end-user experience, which is certainly an important element of your company's quality management strategy. However, certain limitations exist with traditional UI testing given the increasing complexity of application environments. In order to provide a more complete understanding of any problems that could impact the user experience, the quality professional needs to follow a white box approach — looking inside the box. Knowledge of the underlying architecture is required if you want to virtualize services or automate integration tests.

The different boxes include the following:

- **Black box:** What's a black box? The application code is considered a black box to UI testers because they only need to know the inputs (to the application) and the outputs (from the application). The data transformations or analytic computations that generate the outputs that the users see aren't relevant.
- **White box:** What's a white box? Testers follow a white box approach when they examine and test the internal structure or workings of an application.

To test at the service layer, you need to follow a white box approach. Getting your application working correctly requires that you ensure the interconnected and interdependent elements all work together. In other words, you need to understand the components of your system and the connections between components. It's going to be important to understand aspects of the architecture of the system that work below the user interface level, either to virtualize services or automate integration tests. To do this, you need to answer the following kinds of questions:

- What are the components?
- How do the components talk to each other?
- What's the technology for getting messages from one point to another?
- What are the protocols (for example, the details of these messages)?
- Where are the endpoints?

For example, I gave you an ecommerce example in Chapter 1. Feel free to flip back there now and take a look at Figure 1-1. This example contains a business logic layer, a presentation tier (mobile devices, laptops), and the back-end services (order handling, credit check, and so on). Some of the back-end services are developed internally and others are provided by third parties. The middle layer uses JMS to publish messages. Those messages are formatted as XML and described by a schema (expressed in .xsd files).


TIP

The components I describe in this section are a few examples of what you may find in your own system. Hundreds of different types of transports and protocols exist, but virtualizing these components will be easier if the software you select for service virtualization includes support for your specific technologies right out of the box.

Communicating between components

In order to virtualize a component, you need to understand how that component communicates with other components in the overall application. These include web services, middleware, and databases. Details of the communication will go into a model of your application environment. Messages — units of information — are sent from or received by components according to various communication patterns:

- **Request/Response:** This type of communication is defined by a behavior pattern where one request generates one response. Request/Response is the foundation of data communication for the Internet. Web services and Hypertext Transfer Protocol (HTTP) use this type of communication. For example, when you type a Uniform Resource Locator (URL) into a web browser, the browser makes an HTTP GET request to a web server based on the URL. The server responds to that request with the contents of the page (often HyperText Markup Language or HTML), which the browser renders. Each request generates one synchronous response.
- **Publish/Subscribe:** This pattern is a little bit different. A component publishes a request for processing to a message queue. Another component subscribes to the message queue, watches for incoming messages, and processes requests from the queue. If a response to the

original requestor is required, the same technique is used on a different message queue. In this situation, the processing component publishes a response message to a queue subscribed to by the originating component. Requests and responses are asynchronous. IBM WebSphere MQ is an example of a message-oriented middleware that uses this type of communication.

For example, consider the business process for approving an insurance claim. The approval process may require the gathering of additional data to support the claim. This may take more time than the requesting service can reasonably wait, so the request is published and the requestor can go on to do other things. When the approval processing is complete, the verdict is published for the requestor to see.

- **Query/Result:** This communication pattern is characterized by a behavior pattern where one query request generates one synchronous response in the form of a result set. Programming models for relational databases, such as JDBC, use this pattern to execute queries. It's worth distinguishing from Request/Response because, unlike HTTP, it abstracts away any network communication, which may or may not be required.

Transporting messages

Communication protocols use various types of transports for sending messages and receiving responses. These transports — methods of communication between components — describe how the messages get from one component to another. Several examples include

- **Hypertext Transfer Protocol (HTTP/https):** This protocol is foundational for data communication for the web. It is Request/Response protocol based on Transmission Control Protocol/Internet Protocol (TCP/IP).
- **Simple Object Access Protocol (SOAP):** SOAP is a simple Extensible Markup Language (XML)–based protocol to enable applications to exchange information over HTTP.
- **Enterprise Service Bus (ESB):** The ESB is an architectural component designed to monitor and control the communication between business services. IBM's WebSphere MQ, Software AG's WebMethods Integration Server, and the open

source Mule are examples of ESBs. A standard Application Programming Interface (API) such as the Java Message Service (JMS) API is often used to access these systems.

- **Java Database Connectivity (JDBC):** JDBC is an API that uses Structured Query Language (SQL) to connect relational databases and other data sources.

Messaging standards

Many industries have created standards for the details of the messages used in their industry. These standards include specifications for the message *schema*, which are rules regarding the format of the message (such as structure of fields, types, and values). Standards help to ensure that messages pass accurately and quickly between components. These messaging standards are a good fit for service virtualization. Some examples include

- **A web service defined with a Web Services Description Language (WSDL):** Web services are defined with WSDL. Regardless of the messaging schema deployed, the WSDL can be used to describe endpoints and their messages.
- **Society for the Worldwide Interbank Financial Telecommunication (SWIFT) and Financial Information eXchange (FIX):** SWIFT and FIX protocols are specific to the financial services industry. They provide a standardized and secure way to transfer and communicate financial information.
- **Health Level 7 (HL 7):** HL 7 is a standard used for messaging in healthcare environments.

Finding the endpoints

Applications need to know how to reach the services they depend on (the endpoints). Examples of endpoints include a web service URL, a Java Messaging Service (JMS) endpoint, and a JDBC Connection String. Some service virtualization technology has the capability to observe and manage communication between components without requiring your team to make any changes to the endpoints in the application. This really simplifies the process of configuring a test environment to take advantage of virtual components.

Defining Virtual Components

The first step in defining virtual components is to model the architecture with enough detail to expose the boundaries where virtual components can be introduced. It's great if you have a tool that can display the services and components in the system under test and the dependencies between them visually. It's even better if the tool separates the logical view (components, services, and protocols) from the physical view (specific endpoint URLs, hostnames, IP addresses, and so on). This enables the tool to support multiple environments running the same components. They share a logical view, but each environment has different physical characteristics. For example, the IP addresses of servers in production are different from each test/pre-production environment.

Synchronizing with external sources

Much of the information needed to paint this picture may be available in the environment and development assets you already have. You may be able to synchronize with those external sources to populate the architectural views. Some examples include WSDL files and middleware environments.

Recording existing services

You need to define the behavior of your virtual component. Recording an existing service is a great way to capture a lot of the information — including behavior and data — that you need to create test cases and virtual components very quickly. It can also help you decide what needs to be simulated.

For example, some protocols are big, with dozens of messages in the protocol. In practice, your application may not use them all. You can learn what really happens under the covers by recording a session in a production or pre-production environment. Then use this new information to scope your service virtualization effort. The same goes for data.

Here's the value of recording:

- You can discover what messages are actually being shared and the format of those messages (for example, the credit check service may offer a method to request an appeal, but

your application may never do this, and therefore, your virtual component doesn't need to model its behavior).

- You can discover the range of data actually used (for example, the credit check service may return a credit score, and your virtual service needs to know a reasonable range of responses for that value).
- You can understand message exchange patterns. These are sequences of messages. Often you make a request, get a response, and something in that response becomes input to the next request (for example, you use a web service to place an item in a shopping cart and get back the ID of the shopping cart; you then need to use that ID to make another request to view the contents of the cart).

So, what are the important considerations to think about as you're driving the system during recording? I give them to you in this section.


TECHNICAL STUFF

What if you can't record? You don't need to record in order to get started with service virtualization. Virtual components can also be created from the design specifications before the service has been fully developed.

Bootstrapping virtual component behavior

Service virtualization tools look at the inputs to a component and the outputs from a component. Here is an example to illustrate how you can bootstrap a virtual component.

Consider a web service for looking up the zip code for locations in the United States. The request has two parameters: city and state. The response is a string containing five digits. A recording of the traffic to this service may look like Figure 5-1.

Inputs		Outputs
City	State	ZIP Code
Port Jefferson	NY	11777
Austin	TX	78701
Littleton	MA	01460

FIGURE 5-1: A sample recording of the traffic for a ZIP code service.

A recording of the inputs (city and state) and the outputs (ZIP code) provides adequate information to bootstrap or quickly capture virtual component behavior.

Understanding the mechanics of recording

Make sure that the recording technology doesn't interfere with your application's deployment. In addition, virtual components should be easy to introduce into your environment. For example, your testing process may be slowed down if you need to make manual adjustments to allow for communication between the virtual component and the rest of the system. You don't want to take the time to manually reconfigure your application's deployment prior to using a virtual component. Also, if you need to change the application's code in order to pick up an accurate recording, it may interfere with your productivity and success with service virtualization.

Behavior of virtual components

Virtual components exhibit a range of behaviors including the following:

- **Simple:** A simple behavior is deterministic — the virtual component emulates a web service by returning the same response for a certain input every time. For example, assume you have an enterprise travel app called Hotel Finder that communicates with multiple services to determine hotel availability. It performs a request to find a hotel. To test this app, each time you request a hotel, you receive the same response — ABC Hotel. That's enough for some test cases.
- **Non-deterministic:** Here there's a little bit more variability. The service may get the stock quote for Big Company, for example. You want the virtual component to generate a different number each time you request the service. For the purposes of testing, you don't care what the number is as long as it's within a reasonable range. In this case, there's no business logic, but the variation enables more realistic testing.
- **Data driven:** This behavior expands the richness of data available from the virtual component. Input and output data are specified in an external data source such as a spreadsheet or database. The service inputs are used to look up the corresponding outputs. An example of this type of behavior may be a hotel finder service where you need to include

12 cities and three to four hotels in each city. It will take approximately 42 (12 x 3½) rows of data to build a table of responses, but this variety could be necessary to make sure the service communicates with other services in the right way.

- **Model driven/stateful:** In a *stateful* service, one request may change state on the server that must be maintained and factored into responses to subsequent requests. These changes can be modeled with a state transition diagram. For example, say you have an ecommerce application and you want to add an item to a cart. You make a request against a service to add the particular item to the shopping cart. The virtual component needs to remember the contents of the cart to correctly respond to requests that examine the cart, check out, and so on. You want to have a tool that makes it easy to specify this behavior in the tool.
- **Behavioral:** This category applies to components that don't follow the other more typical behavior patterns. You want an easy way to add an arbitrary behavior to your virtual components through scripting or coding.


REMEMBER

Coding adds complexity. Sometimes you can avoid simulating complex behaviors by creating separate virtual components, with different behaviors, for different test goals. Service virtualization makes it easy to switch between components as testing needs change, and you can avoid coding a lot of conditional behavior.


TIP

For managing virtual component test data, here is a checklist of important capabilities:

- Data extracted from production environments must be presented in an easy-to-use way.
- Data must be captured during recording in a way that allows for easy creation of a virtualized service.
- Data should be able to be privatized or masked when needed (for example, you virtualize a service that retrieves medical records — you need realistic data for your test, but you need to privatize/redact it for security and compliance purposes).
- Multiple types of boundary conditions should be tested (for instance, will an empty shopping cart be sufficient for your test environment, or do you need to include items in the shopping cart?).

» Externalize data from virtual components to allow for easy updating. It will be easy for the author of a new test scenario to add relevant data to a spreadsheet.

DETERMINING THE RIGHT TOOL FOR THE JOB

When selecting a service virtualization solution, recognize the needs of different roles within your organization. Not everyone needs to define virtual components, but many testers need to access virtual components in their test environments. To ensure that you can easily and successfully deploy service virtualization, your virtualization solution needs to provide functionality or support for several key elements:

- A method for observing and recording messaging conversations
- A tool for creating and maintaining virtual components
- A hosting environment for virtual components
- An easy way for testers to configure their environment with virtual components

You want a virtual service solution that's flexible enough to allow you to switch back and forth between the real component and virtual components when testing. For example, 90 percent of your testing is supported by a virtual component (say the Hotel Finder example from this section). If that testing passes, the last 10 percent requires the real service (say Expedia). You want both sets to use the same build to avoid risk that something changed between builds. In other solutions, such as *ad hoc* developer-written mocks, the application must be changed, rebuilt, and redeployed. With tools such as IBM Rational Test Workbench and IBM Rational Test Virtualization Server, you don't need to change your application code to toggle service invocations between the real component and a virtual component; simply go to the control panel and click a button. You can even define rules in your virtual component to selectively respond to some incoming requests while letting others pass through to a real system. This is particularly useful when you're trying to limit access to the real system for cost or performance reasons, but there are a few behaviors that are difficult to simulate.

Provisioning Virtual Services

In order to provision virtual services, you need two key capabilities:

- A way to develop virtual services in a personal environment on your desktop
- A shared environment for service virtualization to use across your development team

Your developers and testers need a shared infrastructure for hosting virtual components. In addition, you need to maintain multiple environments in parallel with different mixes of real and virtual components. An environment binds a set of variables from the logical view of your system to specific virtual and physical resources (identified by URLs, host addresses, ports, or other connection settings). By creating multiple environments (for example, developer private, SIT, and UAT), you can run tests against different configurations during each phase of the product life cycle. Figure 5-2 shows how the services in the Whiz Bang example from Chapter 1 are bound in three different environments.

Services	Environments		
	Dev Private	SIT	UAT
SendMeStuff	Virtual-SMS	192.168.12.15	192.168.1.10
UR GoodForIt	Virtual-GFI	Virtual-GFI	http://sv4d.co/goodforit
PurchaseMyStuff	Virtual-PMS	http://sv4d/test/purchase	http://sv4d.co/purchase
TurnMeOn	Virtual-TMO	Virtual-TMO-2	Virtual-TMO-2
GetMyInfo	Virtual-DB	jdbc://192.168.12.27/infodb	jdbc://192.168.1.20/infodb

FIGURE 5-2: Provisioning services (virtual and real) across test environments.

At this point you need to be able to control which traffic is routed to a real service and which is routed to virtual services across your environments. After you have your environment established, you should be able to run all your tests without knowing the difference between real and virtual services. Your service virtualization tool should allow this change in routing without requiring any modification of the application code or configuration. You can adjust virtual components throughout.

RUNNING TESTS AND EVALUATING RESULTS

You can easily run tests by taking advantage of service virtualization. With service virtualization, all your tests — such as manual, automated, user interface, and integration — can be run in the same manner as in a non-virtualized test environment. Ideally the tool used to create your virtual service also includes a capability for running tests and reporting on the test execution results. Or better yet: The tool is integrated with a quality management solution to manage and report on execution results in a single centralized repository accessible by all. The same quality management best practices that apply in non-virtualized test environments are also important in virtualized test environments.

IN THIS CHAPTER

- Developing a business case
- Realizing the financial benefits of service virtualization
- Being successful with your service virtualization solution

Chapter 6
Measuring ROI

The first part of this book provides insight into how service virtualization can help you reduce cost, increase quality, and reduce the risk in delivering high-quality, complex applications. Now that you're ready to move forward, you find out about building your business case for service virtualization.

In this chapter, I review some of the key benefits of service virtualization and their potential contribution to your return on investment (ROI). In addition, I provide a checklist to help you choose your tools carefully and invest wisely.

Building Your Business Case

The purpose of a business case is to help management understand the costs and benefits of a new initiative prior to making an investment. You can develop a business case to determine if implementing service virtualization is economically viable for your organization and to assess how it compares to other investment alternatives. You need to measure how service virtualization enhances your testing process and also, more broadly, look at how service virtualization improves your application life cycle management processes. Because the results are measureable, quick, and real, your business case should help convince management

to make service virtualization a priority investment for your organization.

Your business case should include three main components:

- The rationale supporting your decision to implement service virtualization
- The estimated costs for the implementation
- The estimated benefits of the implementation


TIP

Make sure to develop your own list of projected benefits to help measure your ROI. While many companies have a very long list of benefits, to get you started, here are a few important ones:

- Increased team velocity
- Decreased cost of quality
- Reduced project risk
- Improved level of testing
- Defect removal efficiency

Why service virtualization?

How much testing downtime occurs due to your team's inability to access systems? How much money is wasted as development teams wait to begin testing? Service virtualization is important because it helps with the areas in your test process that cause the greatest pain. You need to understand the costs that you can attribute to delays in testing. Refer to your internal company analysis on this topic to provide a business and technical rationale for service virtualization in your organization.

In addition, get an idea of how often you need to access third-party interfaces prior to scheduling your tests. Understand how much you're currently paying third-party service providers to access their test environments. In essence, this component of your business case is the place to begin justifying the recommendation to implement service virtualization. Identify what your testing environment is like now and the biggest problems that you want to fix. Make a convincing case regarding the importance of streamlining the software development process and how it can be accomplished.

Also consider referencing Chapter 4 to help with the first component of your business case. In Chapter 4, you find a number of questions to assess the need for service virtualization.

Estimating the costs of implementing service virtualization

The costs of implementing service virtualization are relatively easy to identify. They include the cost of buying licenses for the software, the hardware cost to host the solution (if hardware isn't already available), and any associated implementation costs. There will also be some time spent training the development team on the new tools. Your estimate for these expenses will vary depending on the software solution you decide to purchase and its ease of use.

Estimating the benefits of implementing service virtualization

You want to use this component of your business case to articulate the costs you can avoid by implementing service virtualization. In addition, you want to provide insight into how you expect to gain efficiency and productivity. The next section, "Quantifying the Benefits" presents some real-world examples of the benefits potential.

Quantifying the Benefits

Testing makes up a significant portion of the cost to develop your application, and if you can eliminate some of these costs, by adopting service virtualization, you stand to realize a large financial benefit. There will also be benefits based on the reduced testing time and improved quality of your applications. These benefits can be significant, but some aren't easily quantified in an ROI calculation. For example, how do you measure the benefit of getting to market with an innovative product before your competition? What's the monetary value of decreasing the duration of a testing cycle?

Regardless of how you test or the complexity of your applications, many testing costs can quickly be eliminated with a mature service virtualization solution implementation. If you are one of the many organizations currently struggling with testing complex applications or you have dependencies on third-party services, you can expect a significant ROI indeed!

Some of the most important benefits you'll want to measure to estimate your ROI include the factors in this section.

Eliminating or lowering costs associated with traditional test environments

The complexity of today's composite applications requires modern test environments to enable multiple servers hosting a variety of application software, including http servers, application servers, middleware, databases, and more, installed on a variety of operating systems. The costs for setting up and maintaining such an environment include everything from the initial hardware and software costs to ongoing costs of server administration and maintenance.

For the purposes of this discussion, assume a test environment requires five servers: http server, application server, middleware server, database server, and ERP system. Four of the servers have a cost of $5,000 per month, and the ERP system costs $20,000 per month. The team is considering the use of service virtualization to emulate the database and ERP system during early test phases. For every month that testing can proceed without these physical implementations, the savings would be $25,000.

Alternatively, you may decide to avoid some of the high costs of purchasing and maintaining a mainframe environment by leveraging a hosted environment. In this scenario, you pay monthly access fees that may vary depending on the million instructions per second (MIPS) used. Regardless of how you pay for the mainframe, you probably depend on software hosted in such an environment, and there's a cost associated with testing.

What if you could decrease your monthly mainframe testing costs by 50 percent or more? Assume a typical monthly cost of using a mainframe environment for testing is $15,000. Decreasing the

cost of testing on the mainframe by half could deliver $90,000 in annual savings.

Now consider the access fees you may be paying to third-party vendors. These could be charges to access their service in a dedicated testing environment on a cost per transaction. You can decrease these access fees significantly with service virtualization. For example, the access fee for third-party hosted services is $30,000 per month. By virtualizing the services and testing with virtual components, you can decrease this amount by 80 percent. This results in a savings of $24,000 per month and potentially $300,000 per year.

Time spent provisioning test environments

Using expensive resources to set up, tear down, and reset test environments greatly increases your cost of testing. With average full time equivalent (FTE) rates of $104,000 for testers and $135,000 for developers, reducing the effort for managing test environments by 80 percent could result in savings of $80,000 to $110,000 per resource per year. As a result, testers and developers can refocus their efforts from managing test environments to improving the level of testing.

This new efficiency in standing up the testing environment frees up your highly-skilled teams to spend more time on revenue-generating activities. By adopting service virtualization, you can become more efficient and get higher quality software to market faster.

Finding and resolving defects early in the development process

One of the most important metrics to include when measuring your ROI is the reduction in software defects. Defects are expensive to fix and can have an obvious negative impact on customer satisfaction. With regard to software defects, remember these facts:

- Software defects are the biggest man-made problem plaguing software development organizations.

- If you catch defects early in the software development process, it's easier and less costly to resolve them.
- The key to finding defects earlier is to begin testing earlier — moving testing to the left.

For the purposes of an ROI calculation, assume 100 defects are discovered during user acceptance testing (UAT) of a single release for a combined remediation cost of $200,000 or $2,000 per defect. And in production, the cost per defect increases by a factor of 10 or $20,000 per defect. Fifteen defects are discovered in production for a cost to fix of $300,000, bringing the total cost for repairing defects to $500,000. If the number of defects found in UAT and production was reduced by 70 percent, the savings potential in this scenario is at least $350,000.

This figure doesn't account for the potentially significant financial impact of customer dissatisfaction or lost revenue. The benefit of improved customer satisfaction could make the savings even more substantial. A service virtualization solution puts control in the hands of the tester to begin testing earlier and isolate defects for faster remediation at a much lower cost.

When you tally the dollar amounts from the three very quantifiable measures, the savings are significant and continue to increase with each software release where service virtualization is used. Two additional factors to consider as you build the business case aren't as easy to quantify, but can have a big impact. These factors are faster time to market and process improvements.

Faster time to market

If you begin your testing earlier and you test more efficiently, you can innovate faster and get your products to market before your competitors. Consider your revenue potential if you're first to market with an innovative solution.

Process improvements

Inefficiencies in your testing process are very costly. Consider how much you spend on labor and associated costs while your testers wait for dependent software to be deployed or for a shared test environment to become available. Every minute of tester downtime that you can avoid adds to your ROI for service virtualization.

Some companies find that tester downtime is so high that it sharply reduces productivity for the software development and deployment teams. In some situations, this wait time can represent as much as 30 percent of a tester's work hours — or $30,000 spent on unproductive time for a tester with an annual salary of $100,000. Employing service virtualization allows organizations to get their testers testing instead of waiting.

Selecting a Solution

Prior to selecting your service virtualization software, make sure you understand what it takes to be successful with the solution and how it meets your company's specific requirements. Think of the additional costs you may incur if the capabilities you need aren't supported by the software. If the solution you select falls short on the capabilities you need, you won't realize the ROI you expect.

To help you compare solutions, ask yourself the following questions:

- Is the solution easy to use?
- How much and what level of training is required?
- Can you create virtual components from recordings or design specifications?
- Does the solution support manual, automated (integration and functional), and performance test types?
- Can you create virtual components that allow you to test different scenarios including happy path, alternative flow, and negative testing?
- Can you create virtual components that enable you to test what-if conditions?
- Does the solution offer the capability to quickly deploy and manage virtualized services through an administrative console?
- Can you toggle between live systems and virtual services without having to reconfigure your application deployment?

- Can the solution manage the complexity of your application environment (ranging from data-driven and correlated response sequences to full stateful database emulation)?
- Can you automatically schedule and execute tests supported by virtual components upon the availability of a new application build?
- Can you create, modify, and deploy virtual components without requiring your teams to learn new programming skills?
- How easy will it be to share and reuse virtual components across teams?
- Can your teams develop in parallel environments?
- Will your solution scale to accommodate very large teams as you grow?
- Does the solution require a long-term professional services engagement to help you get started and continue to move forward?

Technology is constantly changing, so make sure that the software vendor you select is committed to delivering new functionality regularly to ensure the solution remains current. To help speed your adoption process and keep your productivity levels up, make sure your vendor provides excellent support with access to trained technicians.

IN THIS CHAPTER

- Start off on the right foot
- Put control in the hands of the tester
- Increase productivity by building skills

Chapter 7
Ten Key Points for Success with Service Virtualization

Service virtualization has the capability to dramatically increase efficiency in the way you test software. Service virtualization enables you to rethink your approach to testing and adopt processes that are optimized, setting you free of testing bottlenecks and allowing you to focus on creativity and innovation. You recover time that was typically spent standing up complex test environments and continuously testing through the development life cycle. You explore advanced testing scenarios sooner without having to reinvent the wheel. The return on your service virtualization investment is quick, and the benefits to your development teams and your business are real.

Rethink Your Approach to Testing

Service virtualization supports any type of testing methodology, and your goal should be to accelerate test execution — manual or automated — in a way that's repeatable and offers improved

efficiency. The most appropriate place to start is by understanding your testing methodology and determining where service virtualization can increase team velocity while empowering your team to deliver higher-quality software. After that, you may want to look at testing the exchange of messages and behavior between the integrated components of your composite applications. Why take a black box approach to testing and execute tests from the user interface when you have the opportunity to validate at the integration layer? Service virtualization allows you to emulate dependent, yet unavailable, software and test integrations earlier and isolate defects for faster resolution.

Plan for Flexibility

Begin by identifying your biggest pain points — what challenges are keeping you from performing your tests at the right time and with the right level of detail? Teams can reduce testing bottlenecks by starting with virtualizing the components that are the most stable and the most expensive to stand up in test environments. If some components will be delivered late, use service virtualization to help simulate that missing functionality. However, as priorities shift over time, teams also need to be flexible. You may not be able to set all your service virtualization requirements upfront because priorities may shift over time.


TIP

Start simple and take on more complex service virtualization as your team becomes more comfortable with the improved process. Continuously review your needs and make adjustments to improve efficiency.

Practice Controlled Integration

Too often development teams delay integration testing until the end of development and bring all components together hoping for the best. Unfortunately, things don't always go as planned. New defects, design flaws, architectural issues, or massive failures discovered during late development stages will put a project at risk. A virtual component, simulating the real software's required behavior, offers a new level of control to the test team. Systems can be isolated and subsystems simulated using virtual

components to make the unavailable available. Performance tests can be run earlier to validate design and architectural decisions. And when the new or modified software is finally available to test, development teams can introduce the new source code in a controlled fashion by turning off the virtual component and testing the real implementation.

Test Continuously from Development to Production

You need a straightforward way to continuously test your code so you can avoid introducing errors that become more difficult to find and fix later on. The challenge with this is that testers often lack access to critical elements at the right time to make continuous integration testing a reality. Service virtualization makes this possible. Creating virtual components that operate the same as the real implementation makes it easier for your testers to test in sync with development. Instead of encountering roadblocks to continuous integration testing, your teams will be able to test at a more continuous level in many more environments than were previously possible.

Externalize Your Test Data

Valid test data is an integral part of testing and service virtualization. However, making sure that your testing process incorporates data that's accurate as well as manageable in both size and complexity isn't a simple task. Your data may come from multiple systems and multiple sources. Therefore, you should externalize the data, decoupling it from the virtual component itself. This makes it easier for resources unfamiliar with the virtualization solution implemented to enhance the data. As a result, you can create a data set that's accurate and consistent with the real application without requiring technical knowledge of the service virtualization solution itself. If you extract data from a production environment, you need to make sure you use this data in a responsible way that protects personal information.

Explore Advanced Test Scenarios

Service virtualization provides a way for you to do important system level testing that you always wanted to do but couldn't because it was too costly or time consuming. Or maybe you didn't have access to all the components you needed. Today's complex software applications need more complete and advanced testing across the life cycle.

With the increasing focus on quality, you need to be able to start testing — even negative and performance testing — much sooner. Service virtualization enables you to simulate what-if situations and stress your system to test for errors outside of the production deployment. In addition, you can quickly implement alternative paths in a virtual component, allowing you to create situations that regularly occur in the "real" world but are challenging to reproduce.

Avoid Reinventing the Wheel

Too many times, developers and testers will start from scratch in an attempt to increase testing efficiency, improve defect identification, or fix issues faster. For example, you may start out on a path to solve testing bottleneck issues by manually authoring simulation stubs. With service virtualization, it's much more productive and cost effective to implement a commerical solution that's built on proven best practices to create virtual components. Leveraging technology built on a highly abstracted model is more effective than coding stubs manually. Organizations should look for a supported solution that offers out-of-the-box support for their middleware, protocols, and formats. This allows developers and testers to do what they were hired to do, which is focus on improving the business value of the resulting product and delivering higher quality software.

Service Virtualization Isn't Just for Testers

Application quality doesn't just fall on the tester or a group of testers within an organization. Everyone should contribute to improving the quality of the software delivered. In today's world,

programmers execute tests as part of their development responsibilities, and they're subject to the same constraints as testers. These programmers may test the code they wrote without validating integrations. Because much of the dependent software isn't available for testing, the programmers will leave the integration testing for the test team. However, integration testing is too important to leave for later in the development cycle.

Service virtualization is the enabler allowing development shops to move integration testing farther to the left, simulate the missing dependencies, and get insight into the quality of the deliverable by incorporating automated testing as part of the build cycle.

Share Virtual Components across the Enterprise

Your service virtualization solution should include an easy way to share virtual components across development, operations, and test teams in your organization. Because testing needs to be a continuous process, it's important that all members of your software development and operations teams work in unison with each other. Service virtualization supports cross-disciplinary teams in ways that can change the dynamics of testing in your organization and assist you in being compliant with industry standards. Everyone needs to work together, including analysts and developers and programmers and testers, for maximum efficiency while sharing knowledge and expertise across your entire organization.

Enhance Team Productivity by Building Skills

Adopting new technology is only valuable when team members are well trained in its use. Too often a busy application development team doesn't take the time it needs for training, which can lead to a drop in productivity. Putting in the effort to help team members build the right mix of skills should result in a big payoff. Your developers and testers need to have a good understanding of the technologies used in your application's implementation and how service virtualization will be included. When all team

members work together to solve the problems of testing complex composite applications, you're in a better position to gain value from service virtualization.

Adopting service virtualization is more than just installing a tool. If your teams stay current on the topic of service virtualization and build expert level skills, you'll see a reduction in testing bottlenecks and improvements in productivity and software quality.

Notes

Notes

Discover service virtualization

If you want to begin testing earlier and deliver higher quality software faster, open up *Service Virtualization For Dummies,* 2nd IBM Limited Edition. Discover the secret to increasing the efficiency and effectiveness of your testing processes while reducing testing downtime and testing cost.

Inside…

- What service virtualization is all about
- Testing complex applications more effectively
- Where to start with service virtualization
- The benefits of service virtualization
- Key steps to service virtualization success

Judith Hurwitz is a strategy consultant and thought leader in emerging technologies and is the co-author of eight books.

Go to Dummies.com®
for videos, step-by-step photos, how-to articles, or to shop!

ISBN: 978-1-119-41593-0
Part #: RAM14005-USEN-00
Not for resale

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley’s ebook EULA.