COVER STORY

Getting Closer to Customers: Trends and Strategies in IT Services

by Sid Fuchs
Editor's Notes

All over the world, from one culture to the next, those of us working in the software industry are connected by the individual contributions we make every day. What's more, whether we work as independent contractors, salaried employees, or entrepreneurs, most of us share a fundamental belief that "IT" is more than simply a way to make a living. It's a way to express our capabilities, both personal and professional.

Though we probably don't think about it every day, we are in the process of creating a great global society through powerful communication tools. And while it's important for communities around the globe to retain their unique cultural identities and heritage, we are increasingly capable of understanding each other -- as persons, professionals, as cultures, and as nations -- far beyond the façade of slogans and stereotypes.

However our lives may have changed following the tragedies of last week, we should, as IT professionals, continue our contributions with a greater understanding of the role this industry plays in bringing the world together.

We dedicate this issue to the many friends, including our fellow IT workers, that the world has lost, and to the future they would have us continue to build.

Mike Perrow
Editor-in-Chief
Getting Closer to Customers: Trends and Strategies in IT Services

by Sid Fuchs
Director of Professional Services
Strategic Services Organization
Rational Software

Over the past decade, the IT industry has seen dramatic transformations in business models, technologies, customer demands, and skill sets. As these changes continue, many companies are evaluating their ability to offer their customers consulting, training, and technical support services required to meet new business and market demands. This scrutiny is necessary to ensure that the assumptions and methodologies applied in the services business area enable these companies to properly adapt and deliver value and results. In addition, a major trend today is that customers are viewing widely applicable technologies as commodities. They see little differentiation among different vendors' products for certain hardware (e.g., servers, PCs), software (desktop, database), and networking gear (bridges, routers), and price often becomes the main negotiating point and drives the buying decision. Vendors recognize that their true value add, however, is focused IT services, and they are leveraging these to gain a strong competitive advantage in the crowded software marketplace.

I begin this article by taking a look at how industry changes during the past ten years have spurred the growth and evolution of IT services. I will then offer survival strategies that can help both young and established IT services organizations evaluate their services group, define a new vision, and set new goals. Businesses that are searching for new products and service providers can also use these suggestions to understand and evaluate different companies' approaches.

What is driving the dramatic changes in our industry? There are many factors to consider, but here are a few of the major ones:
Customers are under constant pressure to deliver more complex systems of higher quality and greater functionality -- in less time and using fewer resources.

Technology keeps evolving, and the skills required to deliver software systems continue to become more specialized.

The need to solve business problems -- as opposed to the need to keep up with technological advances -- is now the main driver behind system implementations.

On the customer side, IT has assumed greater importance within most organizations. Whereas they once thought of IT as a support function, businesses are now realizing that they can obtain significant advantages with competitive, high-performing IT departments and projects.

On the services provider's side, the goal is now to deliver solutions that not only solve specific problems, but also provide customers with the highest possible Return on Investment (ROI) for the entire initiative, which includes both products and services. This ROI-centric approach is providing quantifiable proof that coupling product technologies with a mix of services provides a greater level of return and value for the customer. It is also providing services vendors with a competitive advantage in certain markets or industries. Of course, for product companies that have an IT services business unit (as opposed to a pure IT services provider), the situation is more complex, as they must figure out how best to distribute resources, investments, and revenue between their product and services sides.

While strengthening customer relationships has advantages for the customer, the benefits for the vendors are also significant. Repeat sales (and therefore a lower cost of sale), increased market share within specific verticals, creating higher entry barriers for vendor competitors, customer referrals, and a deeper understanding of the customer's business, market, and competition are just a few of the reasons why spending more time with your customers is well worth the investment.

The Evolution of IT Products and Services

Over the past ten years, the explosive growth in IT offerings has added enormous gains to many bottom lines. And as technology has evolved, IT services organizations have expanded with an array of new solutions to solve new business problems. We can understand more about this parallel evolution by looking at the Focus/Value diagram in Figure 1.
Figure 1. Focus and Value of IT Products and Services

**Technology Advancement Solutions**

As Figure 1 shows, a decade ago, IT companies led with offerings driven by advances in technology -- high-performance servers, increased bandwidth, increased complexity and functionality in software. These advances were general in scope and applicable to many domains; they did not focus on a particular business problem or function. In this environment, the role of services was to deliver the technology across many domains, without necessarily understanding the customer's business or key issues. As time marched on, however, solutions became more focused on specific business problems or areas. The solutions' applicability to these areas was also better defined. Then, services organizations had to gain a fundamental understanding and appreciation of the customer's business and how technology could improve performance and address business drivers.

**Large Customizations**

As vendors increased their understanding and awareness of the challenges in their customers' evolving environments, they began to deliver large, customized, single-application suite solutions such as monolithic ERP or MRP implementations. These solutions were more specific than those earlier solutions that focused on technology advancements; instead, they focused on specific market verticals (e.g., automotive, financial services, manufacturing). They were also highly complex and required a huge investment in both capital outlay and personnel time spent by the customer. Customers often found that implementations took longer and were much more expensive than they had bargained for. To get the most from these complex and highly proprietary systems, customers were dependent on services organizations to do the customization, supply integration services, provide training, and maintain the system after it was deployed. In addition, it was often the case that customers had to adopt
new business processes that were aligned with this implementation and tool suite, as opposed to having the solutions adapt to the customer's existing business practices.

**Function-Specific Solutions**

In the next evolutionary stage, vendors began to target IT products and services to specific functional segments: eBusiness, CRM (Customer Relationship Management), SCM (Supply Chain Management), HR (Human Resources), etc. Instead of lumping these functional areas into one huge, customized implementation, this approach treated each one as a set of distinct and independent business processes. On the services side, many new companies were created, and many existing organizations shifted direction to take advantage of this new, multi-billion dollar market.

**Industry-Specific Solutions**

The last evolutionary stage depicted in Figure 1 shows where IT product companies are going today: toward industry-specific solutions that address particular problems within both businesses and functions. These packaged solutions (products and services) and applications generally require less customization by the vendor or customer. With these implementations, the balance of products and services shifts more to the services side as deployments focus on the customers' business processes, environments, training, and specific market constraints and challenges. These solutions include such things as human resources applications for healthcare markets, transaction processing for the gaming industry, supply chain management for military logistics and troop support, and call center routing and management for telecom service providers.

Packaged solutions do, however, require more *integration services* than pure development efforts, as customers work to implement them within their existing architectures and systems. This brings a whole new level of focus to IT services, allowing vendors to get closer to their customers and solve very specific and complex business issues.

**Business Models: Services Operations in IT Product Companies**

In today's business climate, many vertical industries are realizing that services are a leverage point for customer success and satisfaction, repeat business, and market share. In the auto industry, for example, new cars are now sold with maintenance included during the warranty period. This bundling of services helps the dealers build customer relationships and enhances their ability to understand what interests consumers and why they buy certain cars.

IT product companies use a similar approach, typically adopting one of three basic business models to deliver a combination of products and services:

- A separate IT services business unit that provides personnel on a
contract basis to perform functions in areas such as outsourcing, call center support, and multi-vendor services. This services unit also supports the company's own products, but it operates independently of product sales.

- A semi-integrated consulting or services unit that supports the company's product sales and places consultants on site as "residents" to provide long-term support. Services can also be independent of product sales.

- A tightly integrated services unit that supports only the company's own product sales and emphasizes knowledge transfer -- so that customers can quickly increase their own development capacity. These services organizations do not provide long-term support but can leverage a partner pool of IT services providers for that purpose. Services are delivered only in the context of product sales. We employ this model at Rational, helping customers adopt our technology and that of our partners, and enabling them to operate efficiently and leverage the abundant functionality and value in our integrated tool suites and the Rational Unified Process® (RUP®). Customer success (not just customer satisfaction) is a key driver; the emphasis is on accelerating the customer's ability to develop and deploy high-quality software in less time.

For any company, the pros and cons of each model are highly dependent on the company's technology product set, field business and revenue model, company culture, customer profiles, and targeted markets. Each model also supports different strategies (e.g., increasing customer capability vs. adding resource capacity), customer sets, and company cultures.

Staying on Top in a Changing Environment

How do successful IT product companies with services operations (or pure services companies) continue to meet the ever-changing demands of the industry, stay competitive, and deliver the most value to customers? They continually fine-tune their business models. They evaluate not only their product and service offerings, but also their internal business and field models to assess whether they can adapt and scale to new demands. Then, they make incremental changes to minimize disruption to both their own organization and their customers' organizations. Companies that avoid incremental change, or any change at all, either die a quick death or wither on the vine.

The key is to evaluate constantly and be willing to act on the conclusions you come up with. Below we will discuss strategies for adjusting business models in response to current trends.

Develop Domain Expertise

In the past, vendor/customer relationships worked as follows: Vendors brought technology expertise (hardware, software, networking, etc.) to the effort or project; the customer brought domain experience; and together, they arrived at a solution. Today, however, as IT vendors
become more entrenched in their customers' business processes and attempt to deliver more focused solutions, they must create teams that have experience in their customers' domains. If a software company wants to penetrate the financial services market, for example, it should hire application developers and support people from the financial industry. With this domain expertise they will not only increase their ability to understand the nuances and specifics of that industry; they will also be able to deliver and support solutions that have a much greater value than those that take a more generic approach.

**Organize by Vertical Markets**

Companies that build industry-specific solutions generally organize themselves to reflect the vertical markets they pursue. For example, many large, and sometimes small- to mid-size companies, have a federal government business unit dedicated to sales and service for that vertical. Why does this make sense? Here are a few reasons, based on my years of experience working within a government agency:

- The federal government has the largest IT budget in the world.
- The government's motivation is not to turn a profit, but to produce efficient programs. This means that vendors must understand and create unique solution strategies that focus on the same goal as the customer in this situation -- focusing the goals of commercial customers (e.g., profitability, faster time to market, competitive advantage), may not work effectively in the government space.
- The buying cycle for government is generally longer, more involved, and requires more vendor investment than the commercial buying cycle (the government's big IT budget supports this).
- Security clearances and special access certifications are sometimes required to work in this vertical (you can't get close enough to make a difference if you don't have access to their world, their problems, and their strategies). Therefore, vendors who invest in obtaining these clearances and certifications can gain a competitive advantage.
- The technical and business challenges of the government are usually leading edge and unique from those of their commercial counterparts -- although this is changing as the commercial world is catching up.

Working with federal government customers requires a level of experience and skill that typical account managers or consultants may not have. The same is true for other verticals. By focusing on what is unique about a particular industry or customer, IT services organizations can accomplish several things:

- Get very close to the customer and become a part of their planning process.
- Have a deeper understanding of the customer's issues -- technical, political, financial, and cultural.
Create and deliver solutions that address a specific problem; this means adapting the vendor's approach, practices, and values to fit the customer's world instead of vice versa.

Increase chances of securing repeat business and long-term relationships, which reduce the cost of sales and provide a more predictable revenue stream.

Enable the vendor's own teams to work together on targeting multiple customers, in multiple geographic locations, within that vertical. A team in Washington, DC, that is working on an opportunity with a government customer, for example, can communicate and work closely with another team in Los Angeles working on a similar opportunity with a different government customer or contractor. This kind of information sharing and coordination also reduces conflict over credit splits and resource allocation.

Create a knowledge base of information on the history, buying habits and preferences, priorities, and relationships of key customers within that vertical. This gives the vendor and the customers significant leverage and allows the vendors to approach these customers at the enterprise level (economies of scale for the customer), and work with both ongoing and new customers in a more efficient and effective manner. Also, the vendor can manage and use ROI data for each customer as a sales tool -- to consistently demonstrate value in the customer's environment.

A cautionary note: As customer organizations grow, bureaucracy tends to set in, impacting efficiency and execution. Often, vendors can help management monitor for these behaviors and take corrective action.

**Broaden Perspectives and Develop New Skills**

As the market grows but also becomes more competitive for IT vendors, having people in your organization that can do many things is a definite advantage. While it will always be critical to have a cadre of talented experts with a narrow technology focus on your consulting team, it is just as important to have teammates able not only to do consulting, but also work with customers. I have found that it takes a combination of people skills, business acumen, management ability, salesmanship, and depth of understanding to be effective in today's environment. As a manager, one thing you can do is expose your technical teams to the customer from different perspectives. Discuss sales strategies, business drivers, budget concerns, and other issues outside of the technical realm to help them understand what is driving the customer a certain way. Also, you should consider training as an integral part of the team's development and find ways to increase their competency in new technical areas.

**Link Solutions to Specific Business Areas**

As we saw earlier, IT vendor offerings used to be driven, typically, by advances in technology. They described their value in phrases such as "price/performance" (hardware), "transactions per second" (databases), or
"bandwidth" and "capacity" (network devices). These value propositions could be applied to many industries, from government to financial services; the offerings were rarely specific to any vertical. In today's world, however, IT vendors must develop and deliver solutions that address specific business areas. If a company has the world's fastest database or most efficient development process, how can it be tailored to or configured for a specific industry? What works for the auto manufacturers probably wouldn't add the same value for the hotel industry. A software company that has excellent technology must send the message that its technology and services are not just best-of-breed across a wide spectrum, but that they can also help customers address and resolve specific needs. To use a medical analogy, you wouldn't go to a general practitioner for a heart problem; you would go to a specialist, a cardiologist in this case, for help.

A "one solution fits all" approach dilutes the value proposition for everyone and also signals the customer that the vendor doesn't have a true understanding and appreciation for industry issues. If vendors are going to be true "partners," then they must share the risks as well as the rewards; they need to demonstrate a deep understanding of the customer's problems as well as a commitment to helping the customer solve them.

Also, as return on investment (ROI) takes on greater importance in most customers' buying decisions, vendors will need to demonstrate ROI in the context of the customer's environment to make meaningful and relevant competitive comparisons. Showing a customer in the telecom world that a customer in the healthcare industry got a 200 percent ROI is much less compelling than a comparison within the telecom world. When you focus a solution on a specific industry, you can collect and use metrics that carry much more weight with customers.

**Leverage Partner Relationships and Programs**

Obviously, one company cannot be all things to all markets. In order to improve coverage across several markets, vendors must leverage the skills and experience of IT service partners. Establishing a well-managed and well-structured partner program that focuses on complementary skills enables vendors to pursue a wider range of business and reduce channel conflict.

Almost every company today has some type of strategic partnership with another vendor or service provider. Unfortunately, more often than not, these relationships result in "marketecture": marketing relationships rather than real working relationships. In order to avoid falling into this trap, it is essential that partner programs contain the following:

- A plan to cross-train each company's employees on the partner's technology; the plan should also identify resources, timelines, and costs.
- A plan to cross-sell each other's products and services. This should include sales credit, product support, and channel agreements.
- A certification plan that ensures the partner companies have some
level of competency in both partners' technologies. This should also include a mechanism for updating these skills as product releases occur.

- An agreement that outlines what products and services will be required to implement these plans, as well as resources and timelines.

As a side note, when discussions are taking place to establish a partner program, it is important that field personnel be involved. They know what it takes to work with customers, partners, and their own teams, and have a good sense of what is needed to make the partnerships effective. Too many times I have seen agreements in place that did not take these things into account. The results are almost always poor in these situations.

**Make Incremental Changes**

As a services provider, you will often find yourself dealing with companies that invested heavily during one evolutionary stage and now have a desire to move forward. It is not always clear what path to follow, and the decision making can be tough. If the organization spent an enormous amount of time and money implementing a corporate-wide Enterprise Resource Planning (ERP) solution a few years ago, for example, it's important for them to assess whether the system has really solved the business problems it was designed to address. If not, then they should begin making incremental changes as soon as possible. The longer a problem exists, the harder it is to change; this is due to many factors, including feelings of ownership that develop, business processes and systems that become entrenched, specialized skill sets that people acquire, and the substantial investment of resources and money that are made as time goes on. Also, an organization's current system may be wasting precious resources, time, and money that may not be available when the time comes to implement a more effective system.

In cases like these, I begin by helping the organization identify and prioritize business issues according to severity. Then, we take things one step at a time. We assess whether an integrated solution (generally less costly) or a single, best-of-breed solution (requiring more time and money to integrate) makes sense for their business. We define a resolution plan and the results we want, then begin executing changes. After implementation, we carefully assess the impact of each change before proceeding with the next one. This ensures that we won't get too far down the wrong path and often protects the company's existing technology investments.

**Understanding the Past, Looking Toward the Future**

In today's business environment, IT services is now a key differentiator when it comes to delivering value. While general IT solutions will always have a role in the industry, advances in technology have allowed us to create solutions for more complex and specific issues. And if the demand is, increasingly, for industry-specific solutions, then vendor companies must get rid of the "bump and run" approach to selling product or
services, which means moving on to the next deal before they deliver proven value. Getting close to a customer and becoming an integral part of their solution can only be accomplished by spending time with that customer, taking a true interest in their world, and working collaboratively to create high value solutions.

This approach can also give IT vendors a competitive advantage. If you develop a service or methodology to help a pharmaceutical company obtain FDA approval for its software development process, then you can package that solution and leverage it to many similar companies within that vertical. In this way, you can transform a specific solution into a force multiplier and offset the development and selling costs for this solution with each sale and delivery.

One thing we know for sure about the future: Software will be the prevailing technology. Other technologies will be important and necessary, but the creativity, functionality, and innovation required to meet new demands and solve increasingly complex problems will reside in the software technologies and methodologies of the future. In this environment, IT product vendors will have unlimited opportunities to make a real difference in the way we live and work, and IT services organizations will continue to thrive as they support those vendors' efforts.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
Web Services: A New Paradigm for Distributed Computing

by Jason Bloomberg
Director of E-Services
Ashton Services

With the downturn in today’s Internet technology marketplace, Web Services shine like a beacon of hope. Fueled by Microsoft’s .Net initiative, Sun’s SunOne architecture, and IBM’s WebSphere platform, Web Services promise a new level of compatibility across multiple technology platforms. Many companies who still have R&D budgets are pouring time and money into Web Services, and bringing forth an alphabet soup of new technologies: WSDL, SOAP, and UDDI, to name a few.

The ideas behind the Web Services movement are reasonably straightforward. Start with XML, the movement’s backbone. This straightforward language provides a "least common denominator" method for different systems to talk to each other. Next, use XML to create a set of industry standard protocols for finding companies, describing and exchanging information, and handling transactions between companies. Add water, and voila! You have a brave new world of e-commerce.

Or do you? Many are skeptical about the Web Services movement. Maybe what you really have, they say, is nothing more than a new spin on some old technologies. Web Services look an awful lot like the distributed component-based technologies CORBA and DCOM.

So, how significant is the Web Services movement, really? Are these new technologies the key to e-commerce compatibility nirvana? Or are they nothing more than another incremental development in the never-ending saga of technology companies looking for new stuff to sell?

More Than Old Wine in a New Bottle
Just as object-oriented (OO) programming, when it first appeared, looked a lot like procedural programming, programming with Web Services looks a lot like programming the OO way. Early OO programs were little more than procedural programs shoehorned into classes. Eventually, programmers learned to take advantage of the power of OO programming, and a new way of thinking about coding became established. The same thing is happening with Web Services.

The Web Services programs that people are writing today are straightforward applications that leverage the technology's encapsulation and message transmission capabilities -- precisely those capabilities that Web Services inherited from the OO world. What's missing is a broad understanding of the new potential for Web Services, which goes beyond existing models. There's no way to grasp this big picture simply by learning the nuts and bolts of Web Services technologies. What's needed is a top-down architectural vision for Web Services.

**Limitations of Remote Procedure Call (RPC) Architectures**

Until quite recently, we have always thought of computer programs as discrete chunks of code that reside on one computer. Even OO programming began in a single-computer environment. This isolated computer mindset has been around so long that it pervades all thinking about software.

Then along came networks. Only then did people realize that computer programs do not necessarily have to reside on individual computers. Early communication protocols -- such as the Network File System for Unix and Microsoft's Distributed Computing Environment -- focused on the network layer. These protocols, in turn, led to the development of wire protocols for distributed computing, most notably Microsoft's DCOM and the OMG's Internet Inter-ORB Protocol (IIOP) that underlies CORBA.

Remote Procedure Call (RPC) architectures like DCOM and CORBA/IIOP enabled programs to be broken up into pieces, with different pieces running on different computers. OO techniques were particularly suited to this distributed environment, for a couple of important reasons. First, objects maintained their own discrete identities, and could easily be located in different places. Second, the nuts and bolts of communication between objects could be encapsulated into its own set of classes; programmers working in a distributed environment didn't need to worry about how this communication worked.

Programmers, however, continued to work within an isolated computer mindset. Both DCOM and CORBA took basically the same approach: Write your programs so that the remote computer appears to be part of your own computer. If you need code that lives on someone else's computer, you can make a simple request, and the hidden machinery of the distributed platform will marshal the remote code and ship it to you via the preferred wire protocol. Of course, the process of marshaling executable code and shipping it over the Internet opened up a Pandora's box of security concerns. And DCOM and CORBA handle key attributes
In addition, both of these technologies had other serious limitations. DCOM is a Microsoft-only architecture, and CORBA, although intended to provide cross-platform interoperability, in reality is too complex and semantically ambiguous to provide real interoperability without considerable manual integration work.

**Web Services: A Better RPC Architecture**

It was in this environment that Web Services came to life, and it offers major advantages over these prior technologies. Web Services are defined as encapsulated, loosely coupled, contracted objects offered via standard protocols. Let's break this definition down to make it more understandable:

- **Encapsulated** means that the implementation of each Web service is invisible from outside the Web service. Its functionality is known only by the interface it exposes. Encapsulation is a fundamental requirement of OO architectures.

- **Loosely coupled** means that changing a Web service does not require a change in the program that invokes the Web service. In addition, Web Services provide dynamic interface descriptions, allowing for Service invocation at runtime. DCOM and CORBA are both tightly coupled, which limits their flexibility.

- **Contracted** means that the Web Service’s behavior, as well as how to bind to it, and its input and output parameters, are publicly available.

- **Standard protocols** are open and freely available. Web Services are built upon XML and HTTP, which are both standard protocols.

In addition, Web Services can interact via the familiar synchronous request/response pattern of RPC architectures, as well as the asynchronous messages familiar in messaging architectures like IBM's MQ. However, this article focuses on Web Services as an RPC architecture.

<table>
<thead>
<tr>
<th>RPC Architecture</th>
<th>Payload Parameter Value Format</th>
<th>Endpoint Naming</th>
<th>Wire Protocol</th>
<th>Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA</td>
<td>Common Data Representation (CDR)</td>
<td>Interoperable Object Reference (IOR)</td>
<td>IIOP (Binary)</td>
<td>Interface Definition Language (IDL)</td>
</tr>
<tr>
<td>DCOM</td>
<td>Network Data Representation (NDR)</td>
<td>OBJREF</td>
<td>DCOM (Binary)</td>
<td>Inherited from COM</td>
</tr>
</tbody>
</table>
Table 1: RPC Architectures Compared

Table 1 compares important features of CORBA, DCOM, Java's Remote Method Invocation (RMI), and Web Services. Because XML is a text-based, standard language, Web Services, as you can see, solve many of the problems inherent in CORBA, DCOM, and RMI:

- DCOM's format for payload parameter values is the Network Data Representation (DR), while CORBA's corresponding format is the incompatible Common Data Representation (CDR) format. Web Services use XML as the payload parameter value format, providing a simple, open language for tagging payload parameter values.

- Web Services provide a better method for RPC protocols to name communication endpoints. DCOM handles endpoint naming with OBJREF representations, while CORBA uses an Interoperable Object Reference (IOR). OBJREFs and IORs do not correlate, and translations between the two are problematic. Web Services and RMI, in contrast, use URLs, which are universally understood and accepted.

- Web Services use the Simple Object Access Protocol (SOAP) as their wire protocol. SOAP is built on top of XML, which makes it text-based and human readable. As a result, it is firewall-friendly and easy to debug, as opposed to IIOP, DCOM, and JRMP, which are binary protocols. On the downside, the binary protocols are faster than SOAP messages.

- Once RPC architectures have defined their wire protocols, they must provide a way to define their interfaces. DCOM relies on COM's standard memory layout, which it specifies on the binary level, allowing access with multiple languages at the expense of robustness and ease of debugging. CORBA provides its Interface Definition Language (IDL), which, along with its APIs, allow objects to interact with Object Request Brokers (ORBs). RMI provides for Java-only interfaces. Web Services, on the other hand, provide the Web Services Description Language (WSDL), which is an XML grammar for specifying the properties of a Web service, including what it does, where it resides, and how to invoke it. Like SOAP, WSDL is text-based, giving it the same firewall-friendly and easy-to-debug advantages that SOAP enjoys.

- A programmer can use WSDL to create service proxies for remote Web Services. These proxies can be thought of as client stubs (analogous to stubs in CORBA or RMI or proxies in DCOM), which are the local representation of a remote program that characterizes
all RPC architectures. Therefore, a Web Services architecture can be thought of as an RPC architecture, following in the footsteps of DCOM, CORBA, and RMI, but with substantial improvements in interoperability and ease of development.

**A New Software Paradigm**

In my opinion, these advantages alone are enough to justify the investment in tools and training necessary to leverage Web Services. In fact, there's a good chance that many of the Web Services applications developed over the next few years will be nothing more than new and improved ways of handling distributed computing, because of that persistent, single computer mindset we talked about earlier.

In truth, however, Web Services also offer a whole new way of thinking about software -- a new paradigm, if you will -- that goes far beyond the current approach to distributed computing. The key to this new paradigm is the technology used to contract Web Services' functionality: It is this contracting that puts the "services" into Web Services.

The technology that provides publicly available descriptions of Web Services' behavior, binding, and communication protocols is the Universal Description, Discovery, and Integration (UDDI) specification. UDDI provides an XML-based, platform-independent way for Web Services to describe themselves, discover other Web Services, and integrate with those services. UDDI provides both an API for programmers and a system of business registries through which companies can list their Web Services. The API is split into a publication API, for putting Web Services into registries, and an inquiry API, for discovering, binding to, and integrating with Web Services in a registry.

On the surface, UDDI looks like little more than an automated Yellow Pages, which would make you wonder whether enough companies will want to participate in UDDI to make it useful. However, a global eCommerce Yellow Pages is only one vision for UDDI -- and in many ways, it misses the point of the initiative.
In reality, UDDI provides a spectrum of options for discovering and binding with remote Web Services, as shown in Figure 1. At its simplest level, UDDI allows programs to bind to Web Services at design time, essentially hardwiring the connection to a Web service. UDDI also allows for dynamic binding to a static service, which enables a program to discover the specifics of a Web service's interface at runtime. Finally, it is also possible for a program to both discover a Web service as well as bind to it at runtime, essentially enabling a Just In Time (JIT) interface to a set of Web Services. It is this JIT integration to remote services that has never been feasible with existing RPC architectures.

Available Web Services may or may not be registered in a global public registry. The UDDI specification also allows companies to build their own specialty registries, which opens up the possibility of entirely new "Web Services broker" business models. UDDI registries might either belong to private e-marketplaces or be internal to large enterprises. As a result, enterprise software packages, including ERP, CRM, and enterprise portal applications, can be rearchitected into a loose collection of Web Services, tied together with a UDDI-compliant registry located on the enterprise's intranet.

**Architecture for Web Services**

How, then, must architects shift their thinking to move from object-oriented architectures to the new "Service Oriented Architecture" (SOA)
enabled by Web Services technologies? A useful approach is to look at the SOA from multiple views. The 4+1 View Model of Software Architecture, introduced by Philippe Kruchten and detailed in the Rational Unified Process®(RUP®), provides a broad and useful technique for thinking about the SOA.

![Figure 2: The 4+1 View Model of Architecture](image)

As shown in Figure 2, the 4+1 View Model of Architecture suggests a five-view approach, with the four main views taking the perspective of key stakeholders in the development project. On top of the four views is a fifth view, the **Use-Case View**, which overlaps the other views and plays a special role with regard to the architecture.

The **Implementation View**, also called the **Development or Component View**, describes the organization of the software code, and addresses issues of software management. In the SOA, the Implementation View focuses on the internal workings of individual Web Services, and on the Web Services technology stack (HTTP, XML, SOAP, WSDL, and UDDI). Since Web Services are encapsulated, managing their construction is very similar to managing the construction of traditional components.

The **Logical View**, also described as the **Design View**, starts with the end user's functional requirements of the system, that is, what the system is supposed to do, and uses them to create a top-down abstraction of the design model for the system. Much of the work of an architect approaching the Logical View of the SOA will involve user context issues of Web Services, including user privacy and security, service contracts, billing, and management, as well as personalization issues. Such user context capabilities are currently being built into Microsoft's Hailstorm environment as well as Sun's Open Net Environment (ONE). As Hailstorm and SunONE become more established, SOA architects will need to concern themselves with these environments. The Logical View will help them with these concerns.

The **Deployment View**, or **Physical View**, maps the software to the underlying platforms, including the hardware. This view is where software engineering meets system engineering. Deployment view concerns for the
SOA architect will include discussions of n-tier, application server based platforms versus peer-to-peer (P2P) platforms (popularized by the Napster music service), as well as whether to use the request-response model of RPCs or go with a messaging model. Web Services should ideally be agnostic with respect to these decisions, but in reality, such platform decisions will continue to be important. In addition, building Web Services on P2P platforms opens up new worlds of bleeding-edge possibilities sure to interest many die-hard techies.

The bulk of the work in an SOA, however, will take place in the Process View. The Process View addresses runtime issues, including processes, concurrency, and scalability. As the work with Web Services moves into JIT integration, the Process View will become increasingly important. In fact, the service aspect of Web Services is fundamentally a set of processes.

In moving from OO architectures to the SOA, most of the real architecture work will occur in the shift from the Implementation View to the Process View. This shift is the new paradigm for distributed computing. In object-oriented architectures, structuring class hierarchies is the focal point of an architect's work. In the SOA, however, structuring the process flows among interacting Web Services in a JIT environment will be at the core of the architect's life.

The Fifth View

I saved our discussion of the fifth view -- the Use-Case View -- for the conclusion to make a point. The Use-Case View is intended to drive the discovery and design of the architecture, and then to validate the different views. The existence of use cases, however, presupposes that there are business users who know what they want a software system to do. The truth is, today's business users don't know what Web Services might be capable of, so they don't know what to ask for. Web Services are just too new.

Today, people express the functionality of Web Services in the context of existing solutions, because that's what they know. It is up to the early adopters to understand and discover how Web Services and the SOA enable a new way of thinking about component technology. Only then will the new capabilities of this paradigm filter down to the business users. So what will the use cases of tomorrow look like? Only time will tell, but they will most likely be framed in the context of global, business-to-business e-commerce.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
Using the Rational Unified Process (RUP) Successfully for Small Development Projects

An Interview with Rational's "RUP Curmudgeon," Gary Pollice

by Liz Augustine
Technical Writer
Rational Software

At the Rational User Conference 2001, I was pleased to hear many customers speak of their success using the Rational Unified Process® on large projects. But after hearing Grady Booch's keynote address in which he said that the average project team size is only about seven people, I started to wonder if you can use RUP successfully on small projects, too. This week, I caught up with Gary Pollice, one of Rational's RUP experts, to talk with him about this topic.

Liz: Hi Gary. Before we start talking about RUP, perhaps you can give us an idea of what a "small project" is.

Gary: I think that most people intuitively recognize a small project. Some people characterize small projects by the number of project members -- fewer than ten would constitute a small project for most people. Others talk about the length of the project. Especially when people are developing for the Web, projects are measured in increasingly shorter time spans. And finally, some people think about the amount of software they're building and measure by number of lines of code, number of functions, or some other metric.

A key characteristic of most small projects is the level of formality, which usually correlates with the number of people on the project and the level of complexity. (You can probably think of exceptions to this general rule -- small projects with high levels of formality, and large projects that are very informal.) For example, if you work on a geographically distributed team of 100 people, your communications need to be much more formal than if you're working on a five-person team, with everyone sitting in the
same office. If you're working on a simple reporting system, you need fewer formal artifacts than you do if you're working on a missile guidance system.

In other words, as the size and complexity grow, you need to develop more project rules. And you need to make sure that people follow those rules, to ensure that you all continue working toward the same goal.

**Liz:** When you talk about "project rules," it sounds like you're talking about process. Is there a place for process on a small project?

**Gary:** You're right -- this is about process. In fact, many of our customers use the RUP successfully on small projects. On one customer visit, I found out about a five-person team that was using the RUP on a pilot project. They were really excited about it then, and as it turned out, the RUP was a great success for them. The project manager actually went on vacation during delivery week with absolutely no doubts about whether the delivery would go smoothly -- it did.

**Liz:** How do small projects benefit from using the RUP?

**Gary:** In a couple of ways:

- The RUP provides guidelines about engineering best practices.
- The RUP helps you understand the activities, roles, and artifacts you may need on your project and when it's best to use each.
- The RUP provides detailed information that helps you effectively apply techniques to your project, if they're appropriate. For example, if you're creating a design model in UML, the RUP can tell you which diagrams are appropriate and give guidelines about how to structure the model. It can even help you use Rational tools effectively as part of the overall process.
- The RUP can help you tailor the process to address specific project issues, for example, in the area of requirements management.

**Liz:** So if you've never used the RUP before, then how do you get started? Say I'm a project leader, and I'm at the beginning of my project.

**Gary:** RUP is very flexible. You start with the most important concepts, then add what you believe your organization needs to be successful. I approach RUP by thinking about the artifacts (requirements, tests, code, and so on) that my project needs, then determining what activities to perform in order to create those artifacts. But you've got to remember that the goal is to build software, not to create artifacts.

Here are the basic artifacts I think every team needs:

- A *vision* statement. This helps the project team understand what to build and later helps them know when they're done building it.
- A *statement of risks*. What are the actual risks you face and how
will you mitigate them? When you think about risk, consider these elements of your project: people, process, and tools. Don't worry too much about possible risks -- if a problem arises later, or even if a risk starts to look more likely, you can always change your risk statement.

- **A development case.** This describes how you will adapt RUP to your needs. One important part of a development case is that it explains responsibilities of each different role on the project. With a small team, of course, each person typically plays more than one role, so it's important to define all the responsibilities carefully.

- **A set of use cases.** These define a series of interactions between the system and an actor (usually a user) that yield observable results of value. When your team practices iterative development (an industry-proven best practice that is described in RUP), use cases help ensure that you're delivering something of value with each iteration. In addition, use cases serve as the foundation for your requirements. (You can also add non-functional requirements that are not based on use cases, such as requirements that focus on response time.) The use cases also serve as a basis for your test plan, your documentation plan, and so on. You need to decide whether to express your use cases in UML, in text, or both. This is a cultural issue, and there's no right answer. Do what's right for your group. Keep in mind that if you skip doing use cases, then you may very well find yourself in the unfortunate position of being 80 percent finished with the project, but having absolutely nothing of value that you can demonstrate or ship. You might not need use cases if you were developing a compiler, for example, but I'd use them for most other projects I can imagine.

- **A good set of tests.** If you're using RUP, then you can begin generating tests as soon as you complete your first use cases. In fact, if you write your tests first based on your requirements, and code to the tests, you can ensure that the software you produce meets your requirements.

- **An architecture.** This may be extremely informal. Some groups release their first version without a formal architecture, then (assuming success) when they're planning the second version, they start by documenting the architecture so far and how to extend it. Other projects develop an architecture from the beginning.

- **A project plan.** This should outline the iterations and schedule. Design the iterations so that you implement the riskiest items first.

- **A glossary.** This should contain definitions to keep your team's language consistent, project-wide.

**Liz:** I can see why these items would be important to just about any project. But how formal do they need to be?

**Gary:** As formal or informal as you want, depending on the needs of your business. You might write your vision on a Post-it note and the project plan on a whiteboard. You can keep your requirements on 3x5 cards. And you might have an oral agreement about your development case but not
write it down.

As the complexity of a project increases, of course, you need to be more formal with these and other documents. The RUP provides templates for many of them, and Rational tools support the process when you're ready for more formality.

And keep in mind that this is not a black or white proposition: You can mix levels of formality. For example, if you're on a three-person team developing the security system for a Web site, communications for that part of the project can be very informal. But when it comes to communicating with other small projects that have an interface with yours, then you may need to be more formal. As a project leader, you can create your own guidelines and adapt elements of the process to meet the needs of your project.

Liz: That's a helpful reminder. But say that I'm a project leader, and I've worked out a process that I think will work for my team. Now, I want to ensure that my team members will follow the process that I've configured. What's the best way to keep us all on track?

Gary: There are a couple of things you can do. I recommend that you build a project Web site -- RUP has examples of these (look in the "Process Engineer Toolkit" area). On the Web site you can post responsibilities for each team member and status reports, and provide guidelines about how to follow the process. Your developers can then explore the process at the level they need, by clicking on links to get more information.

In addition, you can hold a workshop to "kick-start" the project, as described in the RUP guidelines under "Development Case Workshop." This is an opportunity to discuss how your team will use RUP, to show the team how to use the development environment, and to get feedback from the team about how to improve the development process. This is also a great opportunity for your group to start gelling as a team - by working together toward a common goal.

Holding mini-workshops on an as-needed basis throughout the project is also a good idea. For example, during the Elaboration phase of RUP, as you work on developing the architecture, you might do a workshop on domain analysis. Workshops can help ensure that team members are working with the same basic set of assumptions; they can also help strengthen the team-building effects of the kick-start workshop.

And finally, don't forget the benefit of relying on process and technical mentors. Cultivate an informal consulting relationship with an experienced coworker, or hire a consultant who can provide expertise to your team in a particular area.

Liz: Okay, so let's assume that my team has gotten off to a great start and now we're in the middle of our project. How do we know if we're really on track?

Gary: One real benefit of iterative development is that you always have
current data to assess how you're doing. Especially on a small project, I recommend that you schedule short, frequent iterations. RUP tells you that at the end of each iteration, you need to do an assessment, even if it's very brief:

- Start with your vision. Are you still pursuing it?
- Assess what worked and what didn't.
- Ask if you have working software.
- Are there new risks or requirements, and if so, how do you mitigate or include them?
- Check your project plan. Are you still on schedule? Do you need to adjust the project plan or the process to ensure that you'll deliver on time?
- And finally, if something's not working, don't keep doing it!

The main idea is to continuously verify quality, both of your project and of the process you're following.

**Liz:** Any suggestions on where to learn more?

**Gary:** Two good starting points in the RUP are the "Process Essentials" topic and the "Developing Small Projects" roadmap. Both are available from the "Getting Started" page in RUP. If you don't have a copy of RUP, you can evaluate it online from http://www.rational.com/tryit/rup. Once you get going, Rational offers plenty of helpful support at http://www.rational.com/products/rup, including Webinars, reading lists, and even a discussion forum. So even if you're working on a small project, you still have access to a large development community.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
Achieving Quality by Design
Part II: Using UML

by Ed Adams with Sam Guckenheimer
Testing Methodologists
Rational Software

In Part I of this article, published in last month's issue of The Rational Edge, I talked a lot about modeling and its importance to Rational's "Quality by Design" initiative. But so far I have not discussed how modeling is actually done. This article will focus first on the diagrams used for modeling with the UML (Unified Modeling Language), then elaborate further on how using UML models can help software teams achieve Quality by Design.

UML: The "L" Is for Language

The "L" in UML is for language. Language is all about communication. In Figure 1, we cannot decipher "Bow wow ruff woof..." because we do not speak the same language as our canine pals. This is similar to the problem faced today by developers and testers -- and communication problems between these groups are notorious for delaying and derailing projects.

Figure 1: Translating for a Wider Audience

UML, which has become the de facto language of software development, manages a layer of abstraction to improve communication. What does that mean? It means that UML is to software code what CAD drawings and CAE
models are to the underlying engineering equations they represent. You can describe a circle with the formula \( x^2 + y^2 = r^2 \); however, it is easier for most people to understand what you're talking about if you draw a picture of a circle. UML provides both pictures and a common language that everyone can understand. In Figure 1, the translation of "Bow wow..." consists of both a diagram -- the no dogs symbol -- and words expressed in a common language, in this case English. That is what UML is all about: translating concepts expressed at a low level that is decipherable only to a few into higher-level, abstract concepts that the whole team can readily understand and use.

A very specific benefit of using UML is that it provides a powerful communication vehicle. It is an industry standard, managed by a third party organization, and with minimal introduction, testers can work from UML diagrams and derive benefit from them. The use of UML also allows testers to participate early in the software development process. If you use an XP (Extreme Programming) approach to development, for example, there's a requirement to specify and construct tests before the code is written. You can use UML to do this! Even better, test teams can use UML diagrams and extensions to describe their requirements. Yes, the requirements of the test team can and should be considered. Test teams can help architects and developers design test points (a.k.a. design for testability) into their applications, which will trim the test cycle bottleneck and speed up the overall project. A secondary, but significant, benefit of using UML is that it gives you the ability to improve test coverage by mapping typical UML assets like use cases, class diagrams, and sequence diagrams to test activities like test cases, test designs, and test implementations.

**So Let's Talk a Little UML...**

The basic, and most intuitive, UML diagram is a use case, which specifies interactions between users and the system. I am a big fan of use cases because in describing interactions between the user and the system, they also convey a set of test cases that need validation. That conveyance can be inferred; however, I argue that it should be explicit and part of the design process. The UML also uses two other types of diagrams to detail use cases:

- Dynamic diagrams, such as sequence diagrams and state charts, to specify behavior.
- Static diagrams, such as object diagrams and component diagrams, to specify organizational structure.

Let's take a closer look at how these diagrams work together to create an effective system model.

**Use Cases**

What is a use case anyway? The formal definition in the OMG (Object Management Group) Unified Modeling Language Specification, V1.3 is: "The specification of a sequence of actions, including variants, that a system (or
other entity) can perform, interacting with actors of the system." That's fancy talk for a description of one or more actions that a user would take with respect to a software application. It is a specific way of using the system from the user's perspective. Use cases can be used to verify that all requirements have been captured and that the development team understands those requirements.

A use case can be graphical or textual, but ideally it is both. The graphical part of a use case is represented in a use-case diagram or use-case model like the one in Figure 2. This use-case diagram shows that a professor can login, select different courses to teach, and submit grades. The use case can be detailed further with textual information or additional diagrams. Each line in this diagram represents a dialog or a sequence of actions, which is documented by a flow of events, pre- and post-conditions, and optionally non-functional requirements for that use case. So use cases are graphical but mostly textual, because we typically rely heavily on text and written work for communication. Use-case diagrams also create opportunities to automate the management and versioning of use cases, using automated tools.

![Use-Case Diagram](image)

**Figure 2: A Simple Use-Case Diagram**

The real power of use cases is that they save us time and help reduce errors early on, because they define how real people use the system. And multiple people within an organization can leverage them to work simultaneously on separate tasks. A tester, for example, can easily translate a use case directly into a test case at the same time a developer is building code for that use case. (For more on this topic, see Jim Heumann's article in the June 2001 issue, "Generating Test Cases from Use Cases."

Essentially, use cases provide a common understanding of the system that is to be designed, built, and tested. With use cases, test teams can validate that the system became what the designers expected; they provide a medium through which to check and ensure that what got built is what you expected. "User" cases have been around for a long time, but God bless Ivar Jacobson for elevating their visibility and shining a light on the true power and usefulness of use cases with his Objectory Method[^1], which is now legendary in software planning and development.
When a student registers for a course, a lot of details go on behind the scenes. You can realize, or implement, a use case with sequence diagrams, behavioral diagrams, and collaboration diagrams, which incorporate some of these details. Collectively, elements of a use-case realization describe how a particular use case is carried out within the design model, with respect to collaborating objects. A use-case realization ties together the use cases and the classes of the design by specifying what classes must be built to implement each use case. For our discussion, the term class can be used interchangeably with terms like object, component, code-block, or function. The number and types of supporting use-case realizations will vary, depending on what you need to provide a complete picture of the use case and the project's guidelines.

Sequence Diagrams

Sequence diagrams are one way to realize a use case. A sequence diagram represents a time-based flow of a use case. In Figure 3, the Actor (or user, in this case called Ed) interacts with the Authenticator object by invoking the logon method and passing the UserId and Password arguments (which happen to be strings). The user then goes to his shopping cart by calling the retrieveCartSession method from the ShoppingCart object. And finally, the user goes to check out by calling the checkout method from that same object. This sequence of events is a very common use-case realization for Web-based e-commerce applications. And doesn't it specify, quite precisely, the same sequence of events a tester would have to go through to validate that the application can correctly log users on, retrieve existing shopping carts, and proceed to checkout? Of course it does...but it also provides another layer of information usually hidden to the tester: It shows which objects are performing each step! Let's assume the tester had this diagram at his disposal. If he finds a defect when he hits the checkout page, he could augment the defect report by pointing to the object and function call that caused the error. Wouldn't that make for faster and easier reproducibility of bugs!

A sequence diagram shows what the actor is doing, what components he is interacting with, and what parameters are getting passed. Note that the user (actor) is an important part of the diagram to explicitly model what elements communicate with the "outside world."
Sequence diagrams are a great asset for testers. They provide exactly the kind of information that testers typically never get to see, because they detail a layer that is rarely accessible by testers. Fundamentally, these diagrams provide a picture of what the testers must test and validate.

A tester going through a use case at the GUI level can see what is going on underneath the covers. Because sequence diagrams are time-based, if the system generates an error at any point, the tester can correlate that with the underlying code. He never sees the code, but he doesn't have to. UML is handling that layer of abstraction between the low-level code and the high-level GUI. This allows him to annotate the defect that he submits and reference the sequence diagram. He can say, for example, "The error occurred in step 3, so the problem might be with the checkout function in the ShoppingCart object." This is something that the tester could not do before he had access to the kind of information provided in sequence diagrams. Reusing assets in this way is a fundamental tenet of Quality by Design, and UML facilitates this quite nicely with sequence diagrams. I know this piece of advice may seem too "Twentieth Century," but even if you don't share the electronic diagram, print it. Passing on a reusable asset in paper form is better than not reusing the asset at all. And your testers will thank you for it (we hope).

**Collaboration Diagrams**

A collaboration diagram is just an alternate view of a sequence diagram. You can use one, you can use the other, or you can use both. The decision is up to you because they are just different views of the same thing. A collaboration diagram organizes objects based on interactions instead of
time. Figure 4 shows the collaboration of objects to support the Create Schedule sub-flow in the Register for Courses use case. The student is interacting with this course form, and the form interacts with the RegistrationController, which in turn interacts with three other objects. This kind of diagram provides an organizational snapshot of a use-case realization.

![Collaboration Diagram](image)

**Figure 4: A Collaboration Diagram**

**Activity Diagrams**

An activity diagram (Figure 5) is another good behavioral diagram that can augment a use case. I like activity diagrams for two reasons. First, because I came from a procedural programming world (Pascal, Fortran), and activity diagrams are similar to flow charts, which work well in those environments. Actually, activity diagrams are not much more than flowcharts that can handle parallel processes. Activity diagrams illustrate a specific flow through a use case to explain what is happening. Second, activity diagrams include something called swimlanes, which define activities in concurrent threads and who is doing them. Swimlanes provide a view of what the actors are doing versus what the system is doing throughout a use case.
In Figure 5, the activity diagram tells us that once an order is placed by a sales rep, the system (application) and shipping department can act in parallel. The shipping department gets the goods and packs them in a box, then makes a decision to ship overnight or ground. At the same time, the application can e-mail the credit card receipt and begin processing an EDI (electronic data interchange) transaction with the credit company to start the funds moving. The activity diagram itself visually explains what is happening in this use-case realization, while the swimlanes show which entity performs which actions. These diagrams can be generated very early in the project, because they generally model behavior or business logic. What a tester can take from such a diagram is information that can be used right away to start test planning, because this activity will soon be a test-case realization. The tester can begin to design tests to verify this behavior, including expected results, acceptance criteria, etc.

Using UML Diagrams for Testing

Now, how do these UML diagrams benefit testers and lead to Quality by Design? Well, we have seen that they are designed to help developers and analysts understand both the problem space and the solution space. In the problem space are things like use cases, requirements -- anything that is outside the actual application but still has an impact on it. On the solution side, UML diagrams handle the design and realization of an application: with things such as logical views, organizational views of an application's structure, and deployment views. All along the way -- from requirements to
design to realization -- UML diagrams manage levels of abstraction for system definition. As I wrote above, they can give non-programmers a view of the code that is understandable and concise; they interact to provide a comprehensive understanding of the entire system.

But UML diagrams can also manage levels of abstraction for system test. By linking test assets directly to a system's architecture, you can use UML to build in quality from the start. With UML diagrams at their fingertips, testers can create test cases while developers are still working in the problem space, define test designs in the design phase, and manage test scripts in the realization phase.

What exactly are these testing elements I just listed? For the sake of discussion, let’s use these definitions:

- A **test design** is a description of how to test a system. It includes what and where to test, which data to use, expected results, and the steps needed to implement the design. Test designs can be driven from logical views, which represent the organizational structure of an application. Designs convey intent. Thus, questions like "What is this test supposed to do?" can be answered with test design. So driving designs from organizational UML assets like class diagrams is a logical way to proceed.

- A **test case** is a description of a test, *independent of the way a given test is designed*. Test cases can be mapped directly to, and derived from, use cases. You can also derive test cases from system requirements. For example, the requirement, "All transactions must process in eight seconds or less" is easily translated into a system performance test case. And much of this work can occur early and iteratively -- at system design and requirements gathering (which happens throughout the project).

- **Test implementations** or **test scripts** are specific instances of a test design that can be executed against your system. Test implementations can be mapped directly to system implementations and driven from test cases and designs.

An important concept to retain here is that a test design implies intent. A test design describes how you intend to validate a specific use-case realization.

**Test Reuse**

A co-worker of mine uses this quote in a presentation he delivers, and I love it for its applicability to testers.

> The fox knows many things, but the hedgehog knows only one thing. —Archilocus, Fifth Century BCE

Testers are foxes, as testing consultant Brian Marick observed in his keynote speech at a Software Testing Analysis and Review (STAR) conference a couple of years ago. This was a shrewd observation. Typically, testers are not specialists (hedgehogs). They have to know a little bit about
a lot of things: configurations, network protocols, the development language, and so on. Software development organizations can and should take advantage of this fact. How? By trying to identify and reuse patterns and fault models in their test cycles and then capturing that information for reuse. And testers are generally very good at identifying fault patterns and breaking things.

A pattern, of course, is a proven, reusable solution to a recurring problem. Test patterns can be applied in a number of ways, including

- **Asset reuse across projects** and various stages of test.
- **Context matching** and finding fault models, etc.

**Reuse Across Projects.** Let's visit the mechanical design world for a minute. When I was a mechanical engineer, we used a system called "Ideas." Whenever I started a new design, there was a little piece of software running in the background that did a compare every time I made a physical change to my model. It did not do anything too complicated, just simple compares of aspect ratios. If I were designing a bracket, for example, it would compare the height and width and angles with earlier, existing designs. If the system recognized that a similar design had been used or built before in this project, it would interrupt to tell me, "You're designing something that looks very much like this other thing. Do you want to take a look at it to see if you can reuse it?"

It was the most awesome piece of software I ever used. Everything was stored in a central repository that was constantly scoured as I worked with my model. Imagine being able to tell when similar tests have been applied to a particular context before, and then being able to retrieve those assets immediately, making them available (and useful) to your current staff. When software developers figure out how to do that with software designs or tests, they will take the IT world a quantum leap forward. In the meantime, you can use UML to decorate or extend use cases to include details about certain test cases or implementations -- details that will help a tester assess whether it can be re-used or applied to the particular job at hand.

**Reuse Across Stages of Test.** There are ways to leverage tests for reuse today, and we can benefit from them right away. Developers often create unit tests to validate specific components before integration testing. Why not leverage these unit tests across similar components, or reuse them during the integration testing phase? If you have a test script that was used in the unit test phase, go ahead and run it again as you are doing your sequence testing or scenario testing. If you have validated that your component performs a specific function just fine in isolation, then you will want to verify that it will continue to perform that function when it has to pass data to other components and receive results back from them.

The same idea applies to system testing. A tester does not necessarily have to know how to interpret information that a unit test tool will deliver. But why not use tools that allow the tester to acquire metrics like code coverage while they are performing their normal GUI testing? Then they can simply attach those results to their regular reports as they submit
them. The testers do not have to be able to interpret what the code coverage results mean; but if they find a defect, they can provide more information so that a developer can understand and debug that problem as quickly as possible.

Things like memory leaks, system crashes, and missed lines of code happen all the time. And testers find them -- sometimes by accident -- but they typically cannot capture good metrics on these problems or easily reproduce them. Even more significant, testers usually cannot point to the offending function call or line of code that caused the problem. Using "white box" testing tools in conjunction with manual or automated regression testing can yield tremendous benefits. What if you found a crash and could detail your bug submission with the exact line of code that caused the crash? Or what if you were able to attach a code coverage report to your test results that showed the checkCreditCard function got missed during your "place an order" test case? That is powerful stuff! Imagine the ease with which a developer could reproduce those defects. Achieving this level of test quality is easy to do. There are plenty of tools on the market for this kind of testing. Don't be afraid of them -- Use them! You will be happier, and so will your development team.

Context Matching. Context is one of many test-specific elements of pattern templates as defined by Robert V. Binder in his book Testing Object-Oriented Systems: Models, Patterns, and Tools. Binder suggests thinking about context by asking ourselves these questions:

In what circumstances does this pattern apply? To what kind of software entities? At what scope(s)? This...corresponds to the first problem to be solved in test design: given some implementation, what is an effective test design and execution strategy? This section corresponds to the "motivation," "forces," and "applicability" subjects of design patterns.3

This is a topic too deep for this article; however, I strongly suggest reading Binder's book. His content on context matching and patterns is terrific.

Fault Models. Fault models represent another pattern element. Almost everyone uses fault models at one time or another, though they might not even realize it. Here is an example. Let's say you go to a doctor and tell her you have a pain in your lower back; it goes all the way to your heel, and you cannot sit or lie in bed without pain. When the doctor says, "You must have a sciatic nerve problem," how does she know that?

Because she has studied fault models. She has studied documentation that identifies patterns, so she knows where to look for problems, and she recognizes this particular pattern of symptoms.

Similarly, testers know where to look for problems in software. They have a sense of where things can break. If a tester opens a browser to test a Web application and immediately gets a JavaScript error, the first thing he might do is check to see if JavaScript is disabled in his browser. If it is, then it might not be an application problem at all, just a configuration problem. The tester knows where to look because he has encountered the problem
before. He recognizes the pattern and knows the fault model. Another simple example of this is when a tester checks boundary conditions everywhere because he knows these areas often contain errors.

Fault models are great; but if a tester keeps his knowledge about them to himself, then it is a waste. Testers need to capture that knowledge by taking the time to document fault models. Sharing that knowledge will increase team efficiency in the long run. Return, for a moment, to our doctor/patient example: If you walk in to see a doctor who is just out of medical school, you wouldn't expect to spend five days going through a battery of tests for the doctor to diagnose strep throat, because the doctor has (hopefully) learned from well-documented fault models. Good doctors can easily recognize patterns that lead to accurate diagnoses, and good testers can do the same for your software.

Test Case Generation. Just as with photographic film, test cases can be imaged from the "negative" of a design or model. If the design is clear, then patterns can be recognized and reused. Patterns allow you to automate the test case generation process.

A good example of this comes from the EJB (Enterprise JavaBeans) world. Imagine you are testing an EJB application, and there is a certain bit of logic that has to happen every time you try to instantiate a bean on a certain Web server. The model shows you that it does happen every time, so you create a test once in a header file or in a callable routine and then simply reference it in every test script. You can recognize a pattern in the design and use that knowledge for test case generation: This is yet another example of leveraging your design model to improve quality.

Some Predictions

With such clear advantages, why aren't more development teams taking advantage of these opportunities to leverage the UML and other tools for early testing and reuse and to achieve Quality by Design? In most cases, it is either because their process does not recognize these opportunities or provide the mechanisms to exploit them, or because they do not really use any process at all. A recent Gartner Group summit identified four states of software development, mapped by "speed of development" vs. "quality." Their "Retro Time" quadrant showed where the software industry was ten years ago: comparatively slow development and lower quality. Their "Chaos" or low-quality, high-speed quadrant showed where a lot of e-business companies are right now -- putting out a lot of bad software really fast. Their high-quality, low-speed quadrant might be military applications or airlines flight tracking systems; these have to be extremely accurate, yet it might take years to develop just one release. Gartner's fourth quadrant, Utopia, is where we all want to be: putting out high-quality software quickly. The question is, "Where are you, really?"

Even if you are not living in Utopia, there is still hope. As one industry analyst group tells it:

There are some bright spots on the horizon. Vendors are beginning to address software quality issues with methodologies
and tools targeted both at higher levels of design abstraction and for use earlier in the design cycle. As a prime example ... Rational Unified Process (RUP) provides process support for incorporating testing information into Unified Modeling Language (UML) models.

If you were lazy and skimmed through the whole article just to get to this one quote, don't worry -- it's a good summary of the previous content. I think this quote is very important because it highlights the need to *use a process that has support for incorporating test information during the design phase*. Whether you subscribe to the Rational Unified Process or some other process, it is vitally important that the process ties quality and testing to the design activity.

**Twenty Years Behind, but Catching Up**

The 1960s saw the advent of integrated circuits (ICs) and assembly language programming. In the 1970s we progressed to large-scale integrated circuits (LSICs) and third and fourth generation languages (3GL and 4GL). By the 1980s the hardware industry had developed design for testability standards, but the software industry was far behind -- about twenty years behind, as it turned out. But I believe that design for testability standards in the software industry are really going to take off in this decade.

As another industry analyst sees it:

> As applications become more complex, spreading over multitiers, across varying networks, and including different client configurations, application architecture testing will become increasingly *mandatory*.
> - Uttam Narsu, GIGA Group

I really love this quote, too -- especially the last line. In fact, I like it so much that I want to leave you with this thought. Mr. Narsu does not say that application architecture testing will be "nice" or "cool"; he says it will be mandatory. That is the essence of Quality by Design. Validate your design! Test your model!

Listen to me; I used to shoot people for a living.

---

**Notes**

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
Top Ten Trends in Streaming Media Application Development

by Frank Days and Mark Halliday
MediaPeals, Inc.

In the late 1800s, during the early days of cinema, pioneer filmmakers such as Thomas Edison and the Lumiere Brothers developed the core technology that made it possible to capture images and project them before an audience. These early filmmakers filmed scenes such as "Record of a Sneeze," made in early January 1894 using Edison's Kinetoscope, which showed a man, Fred Ott, sneezing. Another popular early film was created with the Lumiere Brothers' Cinematograph, showing a train entering a station. This clip was famous for causing many in the audience to rush from their seats to the exit doors, terrified that the train was about to run them over.

Although streaming media content today is not evoking similar reactions from viewers, video and audio are becoming increasingly popular Web site components. This article concentrates on the network, software, and hardware systems that are changing the way we experience video and audio in the enterprise.

The reality is that streaming media is still in the very early stages of development. The state of the art today is a small video image, going out of focus, stopping and starting again a few seconds later in response to network congestion. It is not much better than the Kinetoscope peep shows people were watching over a century ago. Fortunately, the technology continues to improve. Eventually, a majority of viewers will watch streaming video that looks more like television. Additionally, features such as high-resolution graphics, interactivity, and personalization will ultimately make the delivery of streaming video over digital networks and the investment in high-powered computers and high-capacity networks worth the wait.
The question remains: When?

Although we can't accurately predict when that day will arrive, we can expect to see analog video creation and viewing systems gradually replaced by enhanced digital versions. We will wake up one morning to realize that the way we watch video has no connection to the way our parents used to watch video, and whether we are watching a movie on television in the living room or a CEO address on the computer in the home office, it will look and sound great.

As application developers, we don't have the luxury of waiting for what might be. We can only consider available technologies and plan for what is coming in the future. We have identified the top ten trends that will affect application developers over the next few years, as streaming media becomes more prevalent in the enterprise.

1. Dynamic Content Will Replace Static Content

Today, streaming media content creation is where Web authoring was in the mid-nineties. Most of us remember the early days of the Web, when sites were built by hand. Skilled artisans dutifully churned out HTML, page by page, to build a site. The resulting Web sites looked good at first glance. Very quickly, however, companies realized that these sites were difficult to maintain without a great deal of manual intervention. Furthermore, presenting unique pages to individual visitors based on backend databases or user preferences was impossible. Although GUI tools emerged to help Webmasters quickly create individual pages, they alone were not enough. Companies realized that a data-driven approach to building Web sites was needed. Shortly thereafter, application servers that built individual Web pages from data stored in a database and an explosion of personalization technologies began to hit the market.

As people begin to scale up their efforts with streaming content, the need for better ways to create and manage content will become critical. Your marketing department, for instance, is going to ask for ways to target streaming messages to specific customers, and current authoring technologies are not going to cut it. More products will emerge that support the dynamic creation of streaming content and work in concert with application servers, Customer Relationship Management (CRM), and Knowledge Management (KM) systems.

2. Streaming Media Will Become More a Web-based Technology Than a Video-based Technology

Currently, most streaming content is created in the same way that corporate marketing videos are produced. A small crew -- perhaps a producer, cameraman, and soundman -- show up with a Betacam to capture the event on tape. After editing, digitizing, and encoding, the company ends up with a stream ready for the Web. The trouble with this approach, however, is threefold. It is too costly, it doesn't scale, and it isn't optimized for the Web.
One emerging strategy for creating better Web content is known as "guerilla streaming." A single person with a "prosumer" (PROfessional quality conSUMER electronics) digital video camera (using the MiniDV format) quickly captures the event in one or two takes. Post-production takes place on a PC with standard software like Adobe Premiere. The result is content that is developed specifically for the Web in a fraction of the time that it would take a traditional filming crew. We have seen Webmasters create complete video presentations with PowerPoint slides in less than one day. These new technologies allow existing Web team staff with minimal training in video production to create high-quality streaming content.

3. The Current Standards Battle Will Produce One or Two Winners

Real Networks, Windows Media Player, Apple QuickTime, MPEG4, MPEG7, SMIL, SMIL 2.... OK, so who is going to win the standards battle? Today, most sites use either Real only or both the Real and Windows Media formats. It appears that, for the foreseeable future, Real Networks and Microsoft will control streaming on the desktop. Like the browser wars, this one will take a while to work itself out. Even if one standard "wins," it is clear that companies will still have to support more than one media format. Right now it looks as if we are heading for an Internet Explorer-Netscape type of battle -- approximately an 80-20 split. Both companies are pushing their own proprietary formats. What matters to the application developer is that companies will need to support both standards and think about technologies that work seamlessly with both.

What about all the new media players that provide "near TV experiences at dial-up speeds"? While the benefits of these products are compelling, they face major hurdles in the form of getting a critical mass of content encoded and desktop players installed.

4. Streaming Content Will Become More Synchronized with Other Media

Web users are accustomed to the rich experience of browsing through pages and drilling down into information. Sadly, most streaming content on the Web today consists of standalone video or audio files delivered through a media player.

More and more, however, companies are creating "synchronized content." For example, presentations that include a talking head video with PowerPoint slides that change along with the speaker are becoming more popular. As streaming synchronization technologies improve, people will see more content that fulfills the Web's promise of interactivity, such as video programs that respond directly to individual user interaction and educational courses that build in real-time feedback. Today, Web developers lack the development tools to create these rich experiences and link them with existing backend systems to create these types of data driven, dynamically assembled experiences.
5. Live Media Will Make Way for On-Demand (Pre-recorded) Content

If you have ever priced the cost of a live streaming event, you know how expensive it can be when all costs are included. The question is which communications must be live. Time-sensitive events like financial earnings calls or critical corporate information delivery require live streaming. Beyond that one must weigh the benefits of real-time information with the price and performance pressures of a live event. Today, many companies are realizing that they can get many of the same benefits of live streaming by using on-demand content, and they can do so at a fraction of the cost of doing it live. By adding interactive elements like chat, polling, and phone-in teleconferencing, you can create the illusion of a live event without the on-the-spot performance pressures and expenses.

Think back to the early days of television when most shows were live performances. Contrast that with today when few programs, other than news and sporting events, are delivered live. The Web is inherently an asynchronous medium, and on-demand content will ultimately become prevalent.

6. Video Search and Mark-up Technology Will Continue to Improve

As corporations increase their use of streaming, applications that can search (find the appropriate clip) and mark up (identify keywords, concepts, and scenes) video will become more important to managing the knowledge embedded in the content. Video search and markup could be the topic of a long article; there are a range of technologies you can implement today. Most current search applications are based on two basic technologies: image searching and voice recognition. Image searching packages do a passable job of identifying scene changes and patterns in a video. Speech recognition continues to improve as CPUs get faster and data buses wider. Although most companies are thinking only about how to get streaming media online, tomorrow's challenge lies in managing that streaming online content as another valuable piece of corporate data. Expect to see XML as a way to facilitate the exchange and re-purposing of content.

7. The Last Mile Will Continue to Be a Barrier for the Consumer

Now and for the foreseeable future, the ability to view high-quality streaming requires a broadband connection. Companies can still stream audio to the average AOL user, but streaming video to 56K users is limited as smaller video sizes, lower frame-rates, and buffering all lead to a low-quality experience. Obviously, audio streaming is one solution that is attractive for lower bandwidths audiences. We are, however, starting to see some organizations (see NFLFilms.com) putting disclaimers on their homepages that say the site is designed for broadband users. Some developers are starting to segregate their audiences rather than develop to the lowest common denominator.
Given the recent telecommunications meltdown, it also appears that the last mile will still be a barrier for the consumer market. Fortunately, most if not all enterprises have broadband connections to the desktop, making streaming even more attractive to the business user.

8. True Convergence Will Continue to Be a Mystery

While people are still smarting from failed efforts in the early 1990s to develop interactive television, many talented teams continue to experiment with television/Internet simulcast events. No one is sure, however, where these efforts are heading. We recently visited a major ad agency in New York City, which handles the branding for many of the biggest household brands in the US. They had assembled a team to experiment with a wide range of events for "Enhanced TV"-like programming during football games and synchronized Web content during educational shows. They freely admitted, however, that they didn't really know where convergence was heading. They only knew that they needed to be a part of it.

What does this mean to the enterprise application developer? Expect the marketing team to dream up new ways to use Web content alongside traditional media. Although existing video and audio assets will eventually find their way to the corporate Web site, it remains unclear what business case will drive the mass adoption of "converged" technologies. Nevertheless, the consensus is that Internet and television technologies are on a crash course, and we can expect to see new applications that test the limits of current technology and capture our imagination. As businesses test these technologies, new business cases are likely to emerge and drive their adoption.

9. Corporations Will Continue to Improve Network Infrastructure

Despite the recent high-technology slowdown, the network has become the circulatory system for corporate information. Dependence on information technology is only increasing, and network infrastructure will grow along with it. The greatest challenge with streaming in the enterprise is bandwidth. New products are emerging to overcome problems associated with running bandwidth-intensive applications such as streaming media. Low-cost stream caching devices can reduce the network load and move the content to the edge of the network. Advances in switching coupled with smart caching can only help improve bandwidth-intensive applications.

10. Entertainment Will Be Out, Positive ROI Projects Will Be In

While the entertainment industry is still licking its wounds from numerous failed, ad-driven business ventures, business managers have found that streaming media is a cost effective way to train staff and partners, qualify sales leads faster, and enhance existing relationships. For example, one of
our clients estimates that it costs them between three and five thousand dollars to send a single person on one trip to Europe to train sales partners. So as an alternative, they are creating an on-demand streaming presentation that can get the message out for less than the cost of that single trip. Measuring return on investment is simple in a case like this; not only are you reducing travel costs, but you are also capturing an event and archiving it for reuse. With a clearer business case in hand, expect businesses to increase the amount of streaming.

Conclusion

Today, businesses and consumers face a major bottleneck in terms of how to create and integrate content with existing infrastructure, which will inhibit widespread use of streaming media for the immediate future. Java and XML are becoming essential to building streaming media applications; yet most streaming media is developed with desktop tools by skilled artisans and "creative" teams as opposed to software engineers. The next generation of streaming applications will demand better development environments as well as tools that treat rich media content as another business critical data type.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
**Book Review**

*Learning XML: (Guide to) Creating Self-Describing Data* by Erik T. Ray

O'Reilly, 2001  
ISBN: 0-596-00046-4  
Cover Price: US$34.95  
(368 Pages)

XML is the acronym for "Extensible Mark-Up Language," a technology that has opened a whole new realm in the software world and completely re-defined the way we look at and manipulate Web page elements. Originally conceived as a way to focus more importance on data and its structure than is possible through HTML (which focuses primarily on the way data is presented), XML is rapidly growing into something with a much broader potential.

XML's great strength is that it's a vendor-independent, open standard; it has rapidly acclimatized to different environments and is currently being applied to solve problems once thought unsolvable. Quite honestly, the dizzying pace of XML's growth can be intimidating to a Web guy like me, because, already, the development of many standards to solve different problems in different arenas has led to numerous XML "off springs." A couple of years ago, all you had to understand in order to decipher an XML document were DTD and cascading style sheets. Today, so many new terms and technologies have sprung up around XML -- Xpath, Xref, Xpointer, XQL, etc. -- that it's tough to keep up.

**Pitched to Experienced Web Developers**

This new book from O'Reilly is a help in this regard for *experienced* programmers. A more accurate title for it would be something like *Know Everything About XML* or *XML Inside Out*. It is exhaustive. In the preface, the author states his intention to provide a "birds-eye view" of the XML landscape for beginners. But no novice could really learn XML from this book without a considerable level of sophistication based on "real world" Web experience. To cover his bases, the author says that "...some experience with HTML or TEX and the World Wide Web is needed," but the truth is, you need a considerable amount of experience with these technologies to get much from this book's detailed coverage of nearly every facet of XML.
Fortunately, the coverage is clear and lucid, and the examples are both easy to understand and memorable. For example, to explain how the XML processor picks up only those processing instructions that it can understand and leaves the others for other programs to process later, he asks us to imagine a sign: "The party has moved to the green house." People who are interested in attending the party, he explains, will go to the green house, but those who are not interested are free to ignore the sign.

Another plus is that the book doesn't delve into long discussions about aspects of XML that are still under design. The discussion about XML Schema, for example, is fairly sketchy, as the specification was not finalized when the book was written.

A couple of warnings. This became my favorite bedtime book; every time I opened it, I fell asleep within five minutes. The author and publishers could certainly use a few lessons in "reader-friendliness"! My major complaint with the book, though, is that there are not enough code examples. True, every chapter has a three-to-four page working example with numbered sections, followed by two or three pages of explanation corresponding to those numbers. But I found it annoying that I had to look back and forth across a span of several pages to match up the code with the explanations. Although the examples are complete, this is definitely not an effective way to present instruction. And then, guess what? There is no explanation of how to make that example work. But maybe that's expecting too much from a "birds-eye view."

Another problem is the volume of unnecessary information. When I was in school, the course I really hoped I'd never have to take was economics. It always seemed to me that economists could never get to the point: They'd have to haul in historical tidbits, a few math equations, some geography, and a little philosophy along the way. When I read their writing, I found myself skimming carefully through a paragraph to pick out the point, which often did not show up until the last two lines. I felt exactly the same way while reading this book. If, like me, you're the kind of person who frequently says, "Let's get to the point," you might find the going a little tough.

Recently, I started reading Java and XML, a complementary O'Reilly book, which provides working details for most of the technologies discussed in Learning XML. So if you treat Learning XML as your main textbook and Java and XML as your lab book, then you can actually do some programming. I recommend that Web developers working with XML and Java have both of these books in their arsenal to help them build smart, extensible XML applications.

- Anil Tatineni
Web Engineer
Rational Software

Read a review of Adaptive Software Development by James A. Highsmith
Book Review


Dorset House Publishing, 2000
ISBN 0-932633-40-4
Cover Price: $44.95 (softcover)
(392 Pages)

Out of this very rich book, I would like to extract a few simple ideas, just to whet your appetite:

- Speculate, collaborate, learn.
- Results over process.

Instead of the widely accepted loop to express iterative development -- plan, build, revise -- Highsmith proposes speculate, collaborate, learn.

Speculate

We know little, either about the product we want to build (the requirements), or how we are building (the design), or how we are getting there (the plan and the process). We can only speculate. Put forth a few hypotheses. Do some fuzzy planning. But beware: Blindly "following a plan produces the product you intended, just not the product you need." It is actually "deviations to the plan that will guide toward the correct solution."

Collaborate

We cannot manage a software development project the way we do a construction project; there is no fixed, deterministic process for reaching the goal. Much of software development is problem solving, which is hard to describe as a workflow. The actual process we follow includes more emerging behavior than planned, imposed behavior.

Highsmith proposes that the speculations are explored by a set of virtual teams who work together to produce a shared result. The book contains much valuable advice on how to set up an organization for collaboration.

Learn
Highsmith suggests that we learn best when we look at the mistakes, revisit the speculations, and change the assumptions, the models, and anything that got in the way. Then we can go back to "Speculate," this time with more knowledge. The planning will become less and less fuzzy as you iterate. Adapt! The emphasis in the title is on adaptive -- as opposed to predictive -- development. And this adaptation depends more on leadership and collaboration than on command and control.

**Results**

Important in this loop is to stay focused on the results, and not on the process. This sounds obvious, but it is easy to forget, especially as projects grow larger and more complex. Often, managers try to find comfort in focusing on predictive process, to forget about the anxiety of the unknown, the risks, and the changes ahead. "The key to scaling up to extreme projects is to apply increasing rigor to the results, that is, to the workstate rather than to the workflow." Note that Highsmith is not opposed to processes and methods, to rigor, as long as it is imposed "slightly less than just enough." "The emerging solution is to neither abandon process nor force-feed process, but to steer process -- adapting methods to fit the situation using careful analysis and relevant feedback measures."

**Highly recommended!**

There are many more gems in this book. It is rooted in experience, much of it matches my own, and I thoroughly enjoyed it. It is very well written, and it should provide some useful companion reading for managers and team leaders using our very own Rational Unified Process (RUP), especially for those who try to apply the RUP in an overly predictive, bureaucratic way, rather than as a framework and a repository of advice out of which an adequate process can "emerge." This book offers one of the best reasons why, over the past two years, Rational has made the RUP more applicable, universal, and flexible.

With this, I would like to offer a few keys for RUP-literate readers: What Highsmith calls a cycle is what we call an iteration; his project initiation phase is similar to our inception phase; his project mission combines RUP's vision, business case, risk list, and project plan.

This book will join the prized collection on my shelf of books such as Brook's *Mythical Man-Month*, Gilb's *Principles of Software Engineering Management*, and Royce's *Software Project Management*.

- Philippe Kruchten
  Rational Fellow

---

Read a review of [*Learning XML*](#) by Erik T. Ray!
Going Over the Waterfall with the RUP

by Philippe Kruchten
Rational Fellow

When discussing the software development lifecycle, there is a temptation to quickly label the waterfall model as "bad." Instead of making hasty conclusions, we should ask questions. What is the right development lifecycle for my project? Is the waterfall lifecycle bad for this project? Or what about aligning the waterfall lifecycle with the Rational Unified Process?

This last idea may sound like an oxymoron, as the Rational Unified Process® (RUP®) is touted as an iterative development process, and it holds "develop iteratively" as its number one best practice. However, as many organizations are using an overall waterfall approach to system development, it makes sense to try to see how the two apparently conflicting lifecycle models might pair up, and how the RUP can be effectively used within an encompassing waterfall framework.

There are many reasons why you may need to insert the RUP into a waterfall lifecycle:

- To match your software development approach to the development approach of a bigger system.
- To comply with externally imposed standards, especially in a bidding and contracting situation, where intermediate milestones are linked to the delivery of specific artifacts in sequential order.
- To deal with simple situations, like maintenance cycles, for which iterating might be overkill.
- Or, more simply, to introduce the RUP into an organization in which the waterfall mindset is very entrenched, and to avoid making too many changes at once. First you introduce some aspect(s) of the RUP, and then gradually shift to iterative development.

Waterfall Lifecycle vs. Iterative Lifecycle

In software engineering, we inherited the waterfall lifecycle from other
engineering disciplines, where it has proven very effective. It was first formally described for software by Winston Royce in 1970, in rather cautious terms. Since then it has been a bit abused by organizations that used it blindly in circumstances for which it was not suitable.

The waterfall lifecycle goes through a series of phases (see Figure 1):

1. requirements (system requirements, then software requirements)
2. requirements analysis
3. software design
4. coding and unit testing
5. integration
6. system test
7. operations

There is minimal feedback from one phase to another. Also, there is often only a small set of artifacts (also called "workproducts," which can include documents, models, or code) that is produced in each phase, validated at the end of the phase, and then used as input for the next phase. These artifacts are considered complete, almost frozen, and revisited only to fix a major issue.

Of paramount importance for certain projects is the issue of freezing the requirements specifications (together with some high-level design) in a contractual arrangement very early in the lifecycle, prior to engaging in more thorough design and implementation work. This is the case when an organization has to bid a firm, fixed price for a project.

In contrast, an iterative lifecycle exploits the "soft" nature of software, and proceeds by developing in iterations that encompass the activities of
requirements analysis, design, implementation, integration, and test. One of the best descriptions is in Professor Barry Boehm's paper on the "spiral" model. You can summarize it with the catch phrase, "Analyze a little, design a little, test a little, and loop back." See Figure 2.

In the iterative lifecycle model, artifacts are in some ways "grown" or "refined," from one cycle of the spiral to another. They are not thrown away or frozen, but rather expanded. The early iterations produce a very small system, which gradually expands over a series of iterations to become the complete system. Feedback takes place from one iteration to the next.

Figure 2: An Iterative Development Lifecycle

Chapter 1 of Walker Royce's book on software project management has a longer discussion of the contrast between the two approaches.

**The Rational Unified Process**

The RUP offers two major components to a software development organization:

1. A flexible, iterative lifecycle that must be tailored by the project manager to the exact project circumstances.
2. A set of recipes: techniques, templates, "job" descriptions, and tools that will guide the software developers in their daily work.

The iterative lifecycle is of importance mostly for the project manager in setting up the project plan.

The rest of the process, which constitutes about 95 percent of the RUP guidance, is described in the form of:

- **Roles**: competencies and responsibilities. Examples: Designer, Tester
- **Activities**: what people do (the recipe). These are associated with Guidelines, which describe techniques and heuristics; and with Tool Mentors, which explain how to use certain tools to perform the activity. Examples: Integrate Subsystem, Code a Class
- **Artifacts**: things (documents, models, code, and so on) that are created or evolved, together with Templates for these artifacts, and Examples, for clarification and inspiration. Examples: Design Model, Test Plan
The two major components of the RUP -- the lifecycle and the "how to" or guidance -- are not completely tied together. If you wish, you can adopt the RUP lifecycle and some of the techniques, or you can adopt most of the techniques but not the lifecycle.

**Iterative Lifecycle: Iterations, Builds, and Phases**

We saw in the first section what happens over time in a waterfall lifecycle. Let's now look at what happens in *one iteration* of an iterative lifecycle:

- The objectives of the iteration are defined, and the iteration is planned.
- Requirements are further refined and analyzed.
- The design is expanded.
- Code is written, then tested.
- The (partial) system is then integrated and tested (note that this may occur over several builds, which are like mini-iterations inside the iteration).
- The iteration is assessed, and lessons learned are folded into the plan for the next iteration.

What characterizes an iteration is that at the end there is a *release* -- internal or external -- of the software product, which is used to objectively assess the outcome of the iteration, and therefore the progress of the project.

In an iterative lifecycle, you have a succession of such iterations, each ending in a partially implemented product. In practice, iterations will vary slightly in focus and content across the iterative lifecycle: Early iterations focus primarily on exploring ideas, consolidating requirements, putting the architecture in place, and getting feedback from stakeholder(s), whereas later iterations focus on making the software product complete and robust.

Therefore, the iterative lifecycle of the RUP is partitioned in a different set of phases than the waterfall lifecycle. These new phases cannot be named after a type of activity (such as "design"), since these activities will be repeated, or a kind of artifact (such as "requirement"), since the artifacts will be evolving over several iterations and phases.

The phases of the RUP are organized and named primarily for what is achieved.

1. **Inception phase:** Scope the project, define the business case.
2. **Elaboration phase:** Refine the requirements, establish an architecture, mitigate the most technical risk.
3. **Construction phase:** Complete the system up to a point where it can be deployed in limited context ("beta version").
4. **Transition phase:** Finish the product and reach product final release.

Each phase has zero, one, or more iterations, but generally at least one. The
RUP has guidelines and heuristics on how to define phases, iterations, number and duration of iterations, and iteration objectives.

Figure 3 shows a "typical" RUP lifecycle, with 1+2+3+2 = 8 iterations.

Note the key difference in terminology (which tends to confuse people): *What the waterfall lifecycle calls phases (requirements, design, and so on) are only types of activities (or disciplines) in the RUP. If it bothers you that the RUP calls Inception, Elaboration, Construction, and Transition phases, then you can refer to them as stages, for example, and steer clear of the word phase. In the RUP, a cohesive set of activities is called a discipline, not a phase: Analysis and Design, for example, is a discipline.*

**Iterative Lifecycle: A Flexible Template**

This approach with phases and iterations may look a bit complex at first, but it offers the project manager a very flexible template for organizing a project. Let us look at some variations on the basic "theme."

If there is no need to explore feasibility, and if the scoping is easy to do, then maybe there is no real iteration in Inception; we have now 0+2+2+2 = 6 iterations (see Figure 4).

If the architecture already exists, and the requirements are stable and well understood, then there may be few or no iterations in Elaboration; so we are down to five iterations (see Figure 5).
If the product is stable early on and has a solid foundation (architecture), an incremental delivery scheme can be introduced (as originally described by Tom Gilb\(^5\)) with several deliveries in the Transition phase that provide increasing functionality. We now have five iterations but with a different configuration (see Figure 6).

And ultimately, the waterfall lifecycle can be described as an iterative lifecycle with only one iteration (see Figure 7).

**Adjusting the Waterfall Lifecycle**

In practice, most waterfall projects have some limited form of iterations for several different reasons.

- Unless you have a really small project, or a team of super-heroes, you rarely have a beta version so perfect that it does not require some fixes after the product release. So most waterfall projects I have seen have some form of transition phase with one or even two iterations leading to releases to end users.

- Unless you are working in a well-known domain, with some existing architecture in place, and an experienced team, you may need some feasibility studies, some exploratory prototyping, which, if well done, amounts to an early iteration. As Fred Brooks wrote thirty years ago: "Plan to throw one away; you will anyhow."

- Finally, if your waterfall project is big (in size) and long (a year or more), you will want to break it into two "steps" to assess progress, get some milestone payment, and so on.
In short, nowadays, the waterfall is rarely 100 percent waterfall, and any non-trivial project needs some form of internal feedback mechanism to do a good job, or to allow some tactical changes of course.

The most likely case looks like this: $0+0+1+1 = 2$ iterations. See Figure 8.

There is one iteration to do the job up to the beta release or field trial, and a small iteration to clean up whatever needs to be fixed, completed, or changed. There are only two releases, and both are going to the end users.

**Mapping the Rest of the RUP**

Once you have defined your lifecycle, which probably looks like Figure 8, mapping the rest of the RUP elements -- activities, artifacts, roles -- is rather trivial.

In the *Inception phase* (or stage) you will:

- Develop the *Vision*, the *Business Case*.
- Develop a *Project Plan*.
- Compile a *Software Development Plan*.
- Schedule and assign work.
- Develop the requirements (use case, nonfunctional requirements).

Optionally, you can add activities from the business modeling discipline to feed into the requirements work.

In the *Elaboration phase* you will:

- Further refine the requirements.
- Design the architecture.
- Design each subsystem.
- Design the tests.

In the *Construction phase* you will:

- Implement the various components and test them.
- Integrate each subsystem.
- Test the subsystems.
- Integrate the systems.
- Test the overall system, for different classes of test.
- Plan the deployment.
- Develop collateral (user documentation, and so on).
- Release the system.

In the *Transition phase*, you will:

- Fix whatever needs to be fixed.
- Repeat the activities in the Construction phase.

*Note: This is just a sketch; there are many more activities described in the full RUP.*

So, in a sense, the planning and scheduling job is easier here than for the full iterative case. You do not have to worry about iteration planning, end of iteration reviews and assessment, and rescheduling the same activity multiple times across several iterations. In a tailored combination of waterfall and RUP, a handful of project management activities and guidelines are merged. Some artifacts specific to iteration disappear.

Most supporting activities of the RUP are not affected: You will still use configuration and change management, as well as project monitoring. While running (enacting) this WUP (or single-iteration RUP), the team members following guidance of the RUP will hardly notice a difference, since each artifact or activity is not described in terms of a given iteration. Only a handful of activities in project management explicitly mention iterations: iteration review and iteration plan, for example. And if there is the occasional instruction to "...do X in the next iteration," the team can simply remember that it is "as if" they were in the last iteration of the Construction phase. (They are, in fact.)

**RUP Best Practices in the Waterfall Lifecycle**

In aligning the waterfall lifecycle with the RUP, it seems that we have thrown away the RUP number one best practice: Develop iteratively. What is happening with the other best practices?

**Develop Software Iteratively**

We may have limited the concept of formal iterations, but we can keep some of the underlying principles to mitigate risks. In particular, start performing regular *builds* (weekly, daily, depending on the size of the team) as early as you can, to stay on top of events, provide something concrete with which to start the testing effort, and provide feedback to everyone involved. Don’t wait until very late in the development process to start integrating. Dedicate some staff to drive the regular build process. Also, as noted above, develop some prototypes early, while elaborating the requirements, such as user-interface prototypes.

**Manage Requirements**
You may have finalized them very early, but you still need to trace them to the design and the tests, to refine or clarify them, and to manage the scope when things get tough.

**Use Component-based Architectures**

It is hard to develop and validate a new architecture; many projects either just use an existing architecture from another project or acquire one externally.

**Visually Model Software**

There is nothing specific to iterative development in using the Unified Modeling Language (UML) to document your design.

**Continuously Verify Software Quality**

You may have a test phase late in the waterfall lifecycle, but this should not stop you from starting test planning and development, and executing a growing set of tests on partial builds as early as you possibly can.

**Control Changes to Software**

Change control is not specific to iterative development. Although artifacts are not modified as frequently in the waterfall approach, you need a consistent strategy and tools to keep track of artifacts.

**Align the Team Structure to the Product Architecture**

To the original six best practices, I will add this one about organization. One of the mistakes I've seen in "waterfall strongholds" is that they often align the teams with the waterfall phases (that is, the RUP disciplines). As you can see in Figure 9, they have:

- A **requirements team**, which throws the finished Requirements Specs over the wall to...
- A **design team**, which, when "done," throws a complete and reviewed design to...
- A **programming team**, which, after unit testing, throws the code to...
- An **integration team** and a **testing team** -- which will take all the blame for being late to deliver.
We have found again and again that having smaller, more capable teams responsible for subsystems or functional areas of the product produces better results. The organization of these teams matches the architecture of the software product itself, and is complemented by: a project management team to drive, facilitate, and control progress; an architecture team to provide technical coordination; and an integration team to drive the builds. Such a team structure requires a wider range of competence and skills, but will reduce miscommunication, boost morale, allow increased learning, and reduce the numerous errors typically introduced by the "handing off" approach of the more traditional waterfall organization.

**Frequently Asked Questions**

Investigating whether or not to map the RUP to a waterfall lifecycle often raises one or more of the following questions:

- In the RUP, when is the right time to sign a firm, fixed-price contract?
- Are there cases for which the waterfall approach would be recommended?
- I want to use an iterative approach, but "they" (customer, management, other project team) impose a waterfall approach. Any suggestions?
- I have to use a waterfall approach because of the hardware guys. What can I do?

Let's look at some possible responses.

**When Do I Sign?**

When is the time to commit to a firm fixed-price contract? In the waterfall process, there seems to be an ideal point in time: when the requirements are all captured, reviewed, approved, and frozen, and top-level design is mostly done.

In the iterative lifecycle, the ideal point is at the end of the Elaboration phase; this is when requirements are fully understood, technical risks are mitigated, and an architecture is in place. But often this is considered too late: Too much has to be done prior to having a contract. So the alternative is at the end of the Inception phase.

From a risk management perspective, we would recommend for an unprecedented project (or a project with a large amount of risks) to do this in two steps, if the customer is amenable to this variation:

1. A *limited first contract* at the end of Inception, to cover all the activities of Elaboration, and
2. A *second firm contract* at the In terms of the mapping the RUP to a waterfall approach, this means that the contract is signed based on either:
1. A Vision Document: This corresponds to a limited first contract.

OR

2. A full set of requirements that have not benefited from validation through prototyping in one or more Elaboration iterations. This corresponds to a second firm contract.

See Figure 10.

![Figure 10: When To Sign a Contract](image)

But contrary to what I have often heard, you are not better off in a waterfall lifecycle, because you really have no way to realistically validate your hypothesis or the requirements, or to sketch an architecture and try it out.

**When Is a Waterfall Approach Suitable?**

In order to emphasize the major merits and all the side benefits of an iterative approach, some of its proponents have vilified the waterfall approach. But it does have its merits and applicability, and I have personally been involved in successful projects using this approach.

The waterfall approach as described here is most likely to succeed if the project:

- Has a small number of unknowns and risks -- i.e., if
  - It has a known domain.
  - The team is experienced in current process and technology.
  - There is no new technology.
  - There is a preexisting architecture baseline.
- Is of short duration (two to three months).
- Is an evolution of an existing system.

**I Want to Use an Iterative Approach...**

Back in 1986, David Parnas offered some very useful advice in his paper, "A Rational Design Process." He noted that, often, a waterfall lifecycle is imposed by external entities: the customer or purchaser, a system engineering organization, or compliance to a standard. It is often feasible for the software development organization to "fake" a waterfall process. That is, you can offer to the outside world the appearance of following a waterfall
process, while actually running an iterative process for risk-mitigation purposes.

Suppose you have to deliver and review in turn:

1. The requirements
2. The top-level design
3. The detailed design
4. The code
5. The system

Nothing prevents you from developing some of these elements iteratively and in advance. At the beginning, you may appear to be behind schedule, but you rapidly catch up as you progress in the cycle. Also, you have the peace of mind of knowing that you have validated your artifacts by going further into the implementation than would have been the case in a waterfall approach, complementing formal inspection of paper artifacts with demonstration of partial implementations. In the sidebar for this article, Aligning the Traditional Waterfall Review Sequence with the Iterative Approach, an extract from the RUP shows how some typical milestones of traditional waterfall processes match the RUP lifecycle.

The risk in using an iterative approach in a waterfall-imposed project is that there are many opportunities for miscommunication, wrong expectations, and so on, between the various stakeholders. Consequently, project management must be very vigilant and educate the other parties about what they are doing, and the benefits of doing things iteratively.

Aligning with the Hardware People

One delicate case -- one for which, it seems, it is hard to "fake it" -- is when you develop software to run on a platform that does not yet exist and is delivered only toward the end of the software development cycle.

Rather than just accepting this situation, software, system, hardware, and VLSI (Very Large Scale Integration) people must put their heads together to find solutions that allow some software to be run at earlier stages of development on a platform that is reasonably similar to the final hardware. Some solutions are:

- Develop a crude prototype.
- Use part of a previous generation.
- Use discrete components instead of the final VLSI.
- Build a software emulator.

It is often the case that the hardware people themselves need some form of iteration in their process. In such instances the two organizations can support each other rather than accept the worst possible scenario: late integration of untested software onto "finished" hardware, with high risk of failure or long delays if anything goes wrong.
Conclusion

The RUP iterative lifecycle and the waterfall lifecycle are not incompatible approaches, and an organization can adopt the practices of the RUP within a waterfall lifecycle. The RUP activities, artifacts, and roles apply to both iterative and waterfall project processes. A true "hard-nosed" waterfall lifecycle introduces a number of risks to software projects, risks that could be mitigated by adopting an iterative lifecycle. There are halfway solutions for the adoption of an iterative lifecycle in which the internal process differs from the external and visible process. Such approaches help adopting organizations improve their internal development processes while keeping the customers and external stakeholders happy.

Although it is possible to use the RUP in a purely waterfall approach, some words of warning should be given: "You never get it right first time"; "Requirements will change"; and "Give people a chance to learn." A waterfall approach ignores these factors and assumes that it is possible to get everything right the first time. Practitioners accept that this is not the case and therefore often describe their project as waterfall, but within it provide for a number of iterations (beta programs, phase two, prototyping, and so on).

With the advent of e-development, software is required to be both higher quality and delivered faster. These two opposing forces, coupled with the inherent complexity of an "e-solution," require improved ways of working. Often these improved processes cannot be adopted instantly, and thus less bold solutions are required in the interim. The RUP provides a solid framework for a sensible adoption process. It allows an organization to implement certain aspects of the process in a controlled manner, and at a pace that is appropriate. Combined with that flexibility is a set of standards. These standards help ensure consistency -- consistency that is crucial to successful enterprise process adoption.

Let me leave you with these parting thoughts:

- The RUP is a very flexible process framework that can be customized to a wide variety of situations.
- The waterfall lifecycle is never really fully waterfall, especially when there are risks and unknowns to address.
- To help map RUP process guidance, the waterfall lifecycle can be seen as a very simplified iterative lifecycle with only one or two iterations.
- With a bit of creative thinking, a moderately iterative approach can be made to "appear" as a waterfall framework, if that is the externally imposed development approach.

Acknowledgments

Thank you to my Rational colleagues for contributing the waterfall and iterative experience to this paper, in particular Dave West, John Smith, Jas Madhur, Bill Grant, and Catherine Southwood for the careful editing.
Notes

8 Ibid.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
Aligning the Traditional Waterfall Review Sequence with the Iterative Approach

(Sidebar for Going Over the Waterfall with the RUP)

The following text is extracted from the RUP. It matches the RUP iterations with some typical reviews found in traditional waterfall processes, such as DOD-STD-2167A and MIL-STD-498.

The default review sequence for a waterfall lifecycle project is a major review upon completion of each important artifact. For example:

- **System Requirements Review (SRR)**, at the completion of the system specification.
- **Software Specification Review (SSR)**, at the completion of the software requirements specification.
- **Preliminary Design Review (PDR)**, at the completion of the architectural design sections of the software design description.
- **Critical Design Review (CDR)**, at the completion of the detailed design sections of the software design description.

In the RUP, parts of the equivalent artifacts are reviewed as they are completed in each iteration, but the major milestones (and therefore reviews) are aligned with the completion of the phases: Inception, Elaboration, Construction, and Transition. Because of contractual obligations, a project manager wanting to adopt the RUP may have to find a way to reconcile this apparent conflict. Ideally, the project manager should convince the customer that the phase and iteration-based approach in fact gives greater true visibility into project progress, and also reduces risk, so that there is no need for an SRR, an SSR, and so forth. However, this is not always possible, and the project manager has to schedule these reviews at appropriate points. It is possible, in the RUP, to locate the points at which these important artifacts (actually, their RUP equivalents) are essentially complete, although this does not always neatly align with phases or iterations.

Matching traditional waterfall reviews with RUP milestones (see Figure A-1) is done here by assuming that the relative effort spent on requirements, design, and the like, will be approximately the same in the
RUP as in the (ideal) waterfall lifecycle, but that the effort will be distributed differently. The result is the following:

- The SRR (concerned mainly with the Vision) can be scheduled at the end of the Inception phase.
- The SSR (concerned mainly with the Software Requirements Specification) will occur about a third of the way through the Elaboration phase.
- The PDR (concerned mainly with the Software Architecture Document) will occur at the end of the Elaboration phase.
- The CDR (concerned mainly with the Design Model) will occur about a third of the way through the Construction phase.

![RUP Milestones with Traditional Waterfall Reviews](image)

**Figure A-1: Matching RUP Milestones with Traditional Waterfall Reviews**

For efficiency, the project manager, in consultation with the customer, should attempt to combine these reviews with the prescribed RUP reviews. This is clearly possible for the SRR and the PDR; they can be combined with Lifecycle Objectives Milestone Review and the Lifecycle Architecture Milestone Review, respectively. It is not so obvious for the SSR and CDR. However, observing that almost all projects will have at least two iterations in Elaboration and at least two in Construction, it is recommended that SSR be combined with the Iteration Acceptance Review for the first iteration in the Elaboration phase, and CDR be combined with the Iteration Acceptance Review for the first iteration in the Construction phase. In both cases, there is then good visibility of mature artifacts, with enough time remaining for any required correction -- although the iterative approach should cope with this as a matter of course.

---

*For more information on the products or services discussed in this article, please click [here](link) and follow the instructions provided. Thank you!*
Testing J2EE Applications with Rational PurifyPlus

by Goran Begic
Technical Marketing Engineer
Rational Software

Lately, a lot of customers have been asking me how to use Rational PurifyPlus® for a server side Java application. Some general workarounds are available in the online Help section for PurifyPlus tools as well as in the Rational Technical Support database. These instructions, however, may not be sufficient for every available deployment environment -- e.g., a Java application server that hosts and executes Java server side components. Plus, there are many Java application servers on the market now. In this article I will present a proven and efficient way of using PurifyPlus to test JavaServerPages (JSPs) and Java servlet applications running in Apache Jakarta Tomcat, Version 4. Similar workarounds can be applied for the commercial Java application servers; I have successfully "Purify-ed" various J2EE applications running in BEA WebLogic, Versions 5.1, 6.0, and 6.1, as well as in IBM WebSphere, Versions 3.5 and 4.

About Jakarta Tomcat

Jakarta is an open source project supported by Sun. The final goal of this project is to create a free J2EE server side solution of a quality equal to those of commercial solutions. The Tomcat application is the main part of this project, and in many people's minds is synonymous with the whole Jakarta undertaking. Tomcat is not a full size Java application server, but only the Servlet+JSP Engine, which is more then enough to deploy and test JSPs and Java Servlet applications. It can either be run as a standalone or integrated into the Apache Web Server. I decided to use Tomcat because it is very robust, reliable, and available for free. More information about Tomcat and the Jakarta project can be found in the list of references at the end of this article. The page listed in the references also contains a link to the Tomcat installation binaries and source code.

About Rational PurifyPlus
PurifyPlus is a complete solution for testing Java applications. It consists of three applications:

- **Rational Purify** -- A memory profiling tool (collects method and object level memory profiling data and pinpoints application memory hot spots).

- **Rational Quantify** -- An application execution time profiling tool (collects method and line level profiling data and pinpoints application performance bottlenecks).

- **Rational PureCoverage** -- A code coverage tool (collects information about the untested parts of the application by highlighting the unexecuted methods and lines of code).

These three tools provide full support not only for Java, but also for Visual C/C++, Visual Basic, and .NET applications. In this article, when I mention PurifyPlus, I am referring to Rational PurifyPlus, Version 2001A. In order to test Java server side applications with Rational PurifyPlus, you must install it on the server machine.

**About Servlets and JavaServer Pages**

Servlets are Java applications running on the server side; their main purpose is to create content for Web pages upon request from the client. The client side is normally a Web browser and a Web page onto which users can enter data that will be processed by the servlet running on the server side. The results of servlet operations are displayed on a Web page on the client side.

JavaServer Pages (JSPs) are basically HTML pages with special tags that enable them to either embed the Java code in the page or access Java beans and servlets running on the server. Rational PurifyPlus cannot check the HTML syntax, but it can test the Java part of the JSPs by monitoring the events that it collects from the Java Virtual Machine (JVM).

**Preparing Java Servlets and JavaServer Pages for Testing with Rational PurifyPlus**

PurifyPlus offers two basic levels of data collection: the method level and the line level. To test Java applications on the method level, you don't need to recompile the tested Java application to collect all the relevant data about the methods. If you are interested in the line level information, however, then you need to recompile the Java code with the symbolic debugging information. The switch to get the symbols in the Java class files with the Sun Java compiler is `-g`.

```
>javac -g MyServlet.java
```
All the information about Java applications run in the Sun Java 2 compatible virtual machines will be collected through JVMPI (Java Virtual Machine Profiling Interface). It is necessary to use the JVM that is fully compatible with the Sun Java 2 specifications. Rational PurifyPlus also fully supports Microsoft JVM.

You will need to include the servlet library `servlet.jar` in the classpath when compiling servlets. This library is installed with Tomcat and can be found in the directory `<Tomcat home>\common\lib`. Here is an example command line for compiling `MyServlet.java` with the symbolic debugging information:

```
Javac -g -classpath <<Tomcat home>\common\lib\servlet.jar>MyServlet.java
```

### Preparing the Windows Environment for Testing JavaServer Pages and Java Servlets with Rational PurifyPlus

The environmental variable

```
JAVA_HOME
```

specifies the home directory for the default JVM. It is also the default choice for the Java Virtual Machine that will be used by Rational PurifyPlus. To correctly set up the Java run that allows you to use PurifyPlus, however, you must execute the following command line before you start profiling for the first time after a new Java service setup:

```
pstart <or qstart, or cstart> -setup
```

This command will update the Java policy file for the selected Java Run-time Environment (JRE).

Now, let's look at another environmental variable:

```
JAVA_OPTIONS (or IBM_JAVA_OPTIONS for the IBM JVM)
```

Rational PurifyPlus collects all the information about Java applications through the Java Virtual Machine Profiling Interface (JVMPI). Since PurifyPlus and JVM run as two different processes, the Java process needs to load a PurifyPlus shared library called PureJVMPI that will listen to the JVM events, collect information about the run through the JVMPI, and send this information to the PurifyPlus tool. This dynamically linked library is loaded through an additional option for the run of the Java executable: `-Xrun`. Here is an example of how PurifyPlus can be started from the command line:

```
>java -XrunPureJVMPI:Purify Java_App ( OR
>java -XrunPureJVMPI:Quantify Java_App OR
>java -XrunPureJVMPI:Coverage Java_App )
```

To profile server side Java applications and Java services, you need to manually create a special system environmental variable with the `-XRun` option that will launch the selected PurifyPlus tool every time JVM is engaged. For the Sun JVM the name of the variable is `_JAVA_OPTIONS` and the value for the variable should be:
If you use the IBM JVM (necessary for running the IBM WebSphere Java application server), then the name of the environmental variable should be `IBM_JAVA_OPTIONS`.

The above option specified in the `_JAVA_OPTIONS` environment variable will launch Rational Purify (or Quantify, or PureCoverage) every time the Tomcat application is started, and Purify (or Quantify, or PureCoverage) will automatically start collecting data for this Java process. You can keep the report free of data collected from parts of the Java process that are irrelevant for testing servlets and JSPs by using both pre-filters and the PurifyPlus Filter Manager.

**Preparing the Java Application Server Environment for Running with Rational PurifyPlus**

To run Rational PurifyPlus with the Java application server, it is essential to specify the same `JAVA_HOME` variable for both the server and the PurifyPlus tools. For Apache Tomcat, you can set this up directly by executing the command:

```
SET JAVA_HOME=<path to JRE installation directory>
```

For BEA WebLogic Servers and IBM WebSphere, you can modify the batch files you use for setting up the environment to run the server by changing the value of the variable `JAVA_HOME` as shown above.

**Preparing PurifyPlus for Collecting Data from Java Applications Running in Apache Tomcat**

To enable Rational PurifyPlus to collect information from your Java servlets and JSPs running in Apache Tomcat, you must create a custom set of prefilters prior to testing.

**PurifyPlus Filters**

In PurifyPlus there are two ways of filtering data that is not relevant for testing. The first way is by setting prefilters for the tool you plan to use (Rational PurifyPlus, Quantify, or PureCoverage). A pre-filter is a list of Java packages from which no data should be collected during the run. This list is defined in the [Prefilter] section of the `Profile.ini` file that can be found in the main directory of each of the PurifyPlus tools.

Each of the Java packages on the list should be in the new line. Let's look at an example of how prefiltering works in PurifyPlus functions.
Let's say we would like to exclude the Java package `com.sun`. If we specify `com.sun.` (please note the dot at the end of the string) as a line in the prefiltering section of `Profile.ini`, then the file and all its sub-packages will be excluded from data collection by the selected PurifyPlus tool during testing (see Figure 1). To prefilter individual classes you can use fully qualified names (package first) without the terminating dot. For example, `com.rational.MyClass` would prefilter `MyClass` in the `com.rational` package. If you specify just the class name (e.g. `MyClass`), then this class will be prefiltered in all packages. PurifyPlus tools do not prefilter individual methods of a class.

**The PurifyPlus Filter Manager**

The second way to filter data is through the Filter Manager feature available through the Graphical User Interface of each of the tools (see Figures 2A and 2B). The Filter Manager creates a special binary filter file for the tested Java application. It can be used only after the profiling or the coverage data has been collected.
The class files checked on the list will be excluded from the reports, but the data collected for them will be kept in the overall results (i.e., the "Time" box in the lower right corner of the Filter Manager window shown in Figure 2A).

To run Tomcat in PurifyPlus, I recommend the following list of prefilters to the Profile.ini file for each of the PurifyPlus tools (Purify, Quantify, PureCoverage):

org.omg.
javax.servlet.
com.sun.
org.apache.
org.xml.
org.w3c.
sun.
For other Java server applications, you will need to create other prefilters. See my Tech Tips on how to run Rational PurifyPlus with the BEA WebLogic and IBM WebSphere Java application servers.

The Demo Application

The demo application used in this article is an online bookstore called "Duke's Bookstore." This Web application can be downloaded from the Sun Web site. It was originally created as part of a tutorial for creating and running Java servlets. It consists of a series of Web pages and servlets that create the content for these pages. Rational PurifyPlus can be engaged in testing either the whole Web application or only parts of it (individual servlets, for example).

Run-Time Testing of Java Servlets and JavaServer Pages in Rational PurifyPlus

After compiling the Java components with the debugging information and setting up the necessary filters for the tools, you will be ready to collect line level profiling and code coverage information from the servlets and JavaServer Pages.

As a first step, I suggest engaging Rational Quantify and collecting information for an execution time analysis.

Profiling Application Execution Times with Rational Quantify

Assigning the value -XrunPureJVMPI:Quantify to the environmental variable _JAVA_OPTIONS will launch Quantify when you start the Tomcat server application. As you're browsing through the demo Web application, Quantify will record times spent in executing each method and line of code that was triggered through the Web page. After Tomcat is initialized and started, the test application can be reached by specifying the following URL in the browser:

http://localhost:8080/bookstore/bookstore.html

After you finish the test run, close the browser and stop the Tomcat server application. The results of the profiling will be displayed in several different views in Quantify. The first is the CallGraph view, as shown in Figure 3.
The Call Graph highlights the chain of calls that consumed most of the execution time. The thicker line highlights the slowest part of the application and represents the percentage of time spent executing the highlighted method, compared to the overall execution time.

The line level information is included in the Annotated Source view for the selected method, as shown in Figure 4:

Further information can be obtained from other views that Quantify creates, including the Function Detail view shown in Figure 5:
The advantage of using Quantify over some conventional solutions is the way the profiling data is represented. Quantify leads you straight to the heart of the performance bottleneck in the tested application.

**Code Coverage with Rational PureCoverage**

If we now change the value for the \_JAVA\_OPTIONS environmental variable to `-XrunPureJVMPI:Coverage` and repeat the test run of the sample Web application, PureCoverage will record the methods and lines of code that were tested and highlight the untested parts of the application.

As Figures 6A and 6B show, PureCoverage provides both method and line level information about coverage of the tested Java servlet application:
In the method level coverage, PureCoverage provides statistics for the methods of the tested application, sorted by the modules in which the methods reside, or sorted by the source file in which the methods are defined.

The coverage information about the lines of code of the tested application is presented in the annotated source code; different colors indicate that the code is hit, missed, dead, or partially hit.
This type of information can be very useful when determining the steps for run-time testing or creating script files that will automate the tests. PureCoverage also allows you to merge the coverage data for different tests on the same application, thereby providing a clear overview about the quality of the tests that have been made on the developed application.

**Memory Profiling with Rational Purify**

Using Rational Purify to obtain a memory profile of the tested Web application is similar to using PureCoverage and Quantify. Taking "snapshots" of the memory usage for the running application, however, gives you the opportunity to compare the memory footprint of the application at different stages of its execution. It is a very useful method for detecting memory leaks.

Memory leaks in the server applications (which often run 24X7) can easily bring both an application and the system down, because the application continuously uses more and more memory as it runs. The impact of memory leaks on application performance is significant as well. The memory footprint that you can record for Java applications by using Rational Purify enables you to analyze memory usage in fine detail.

When applied against Java servlets and JSPs, Purify produces the same types of reports it produces for Java applications and Java applets. Figure 7 shows a Purify recording for a tested JSP with the invoked JavaBean methods.

![Rational Purify Call Graph for a Demo JSP](image)

The list of methods gives more information about the memory usage for this JSP session, as shown in Figure 8.
Just as Rational Quantify leads users to the portion of the application that consumes the most execution time, Purify leads the user to the application's memory bottleneck. By sorting the methods in the Function List view, it is easy to locate those with excessive memory usage; Purify also provides numerous other views that can help you accomplish this.

**Maximum Yield with Minimum Effort**

It is easy to deploy Rational PurifyPlus on the server machine to obtain extensive profiling and code coverage information for Java server side applications. You can use Rational PurifyPlus not only with commercial Java server applications, but also with a lightweight -- and free -- Java servlet and JSP engine such as Apache Tomcat.

---

**Notes**

1 TIP: I recommend installing Tomcat in the directory without spaces in the names. That makes it easier to include Tomcat directories in the classpath when compiling the servlets, for example.
A full version of Rational PurifyPlus can be downloaded for evaluation purposes from the Rational Web page. The evaluation license expires in fifteen days. Rational Purify, Quantify, and PureCoverage are also available in the following suites of Rational tools: Rational Suite DevelopmentStudio, Rational Suite TestStudio, and Rational Suite Enterprise.

TIP: When installing Rational PurifyPlus, select the custom installation option and then check the option: "Add Purify (and Quantify and PureCoverage) to the system path." This is not a default setting, and it can be helpful when executing PurifyPlus tools from the command line.

For more information on building Java servlets and JSPs, see Khawar Ahmed and Loïc Julien's article in the February 2001 issue of The Rational Edge.

TIP: You can check a large number of classes at once (e.g., all Java.* classes) by marking them on the list of classes with the mouse and holding the CTRL, or SHIFT key on the keyboard. A right-click on the marked set of classes will bring up the pop-up window, and you can then choose "Enable" to create filters for all marked Java classes at once.

References

1. PurifyPlus Homepage
   http://www.rational.com/products/pqc/index.jsp

2. Jakarta Project Homepage
   http://jakarta.apache.org/

3. Java Servlet Specifications
   http://java.sun.com/products/servlet/

4. JavaServer Pages™ (JSP) Specifications
   http://java.sun.com/products/jsp/

   http://www.therationaledge.com/content/feb_01/t_entrose2_ka.html

   http://www.therationaledge.com/content/jan_01/m_memjava_gb.html

   http://www.therationaledge.com/content/jun_01/t_profiling_gb.html

   http://developer.netscape.com/docs/manuals/enterprise/40/servlets/1-intro.htm

9. Servlets Bookstore Example
   http://java.sun.com/docs/books/tutorial/information/download.html

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
Tech tips on how to run Rational PurifyPlus with the BEA WebLogic and IBM WebSphere Java application servers

by Goran Begic
Technical Marketing Engineer
Rational Software

<table>
<thead>
<tr>
<th>DOCUMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>TITLE: How to Test J2EE Applications Running in BAE WebLogic 5.1, 6.0, or 6.1 with Rational PurifyPlus</td>
</tr>
<tr>
<td>PRODUCT: Rational PurifyPlus, Purify, Quantify, PureCoverage</td>
</tr>
<tr>
<td>OS: Windows NT, Windows 2000</td>
</tr>
<tr>
<td>DEFECT #: N/A</td>
</tr>
<tr>
<td>PATCH #: N/A</td>
</tr>
<tr>
<td>REFERENCES: N/A</td>
</tr>
<tr>
<td>CREATED: 10-SEP-2001</td>
</tr>
<tr>
<td>REVISED: N/A</td>
</tr>
</tbody>
</table>

Question:

How do I test J2EE applications running in IBM WebSphere Version 3.5 or Version 4 with Rational PurifyPlus?

Answer:

1. Make sure to download the latest version of Rational PurifyPlus and install it on the server machine together with the IBM WebSphere 4 Java Application server.

   The newest proto-builds for PurifyPlus and the point products Purify, Quantify, and PureCoverage can be found at the following location:

   ftp://exchange.rational.com/exchange/outgoing/PurifyPlus_Prototype

2. Set the system environmental variable JAVA_HOME and assign the path to the IBM JVM JRE installation directory. For example, on my machine it was:
3. Set the system environmental variable _JAVA_OPTIONS and assign it the following value:

-XrunPureJVMPI:<tool>

The tool can be either quantify, purify, or coverage, depending on what Rational PurifyPlus tool you want to use to test your Java components running in the server. This variable will cause the selected PurifyPlus to launch automatically and start collecting the profiling or coverage data as soon as the Java application server starts.

4. Open the ASCII file Profile.ini from the installation directory of the tool you selected in Step #3 and append the following list of Java packages to the pre-filtering list:

```plaintext
ibm.
dcom.
com.ibm.
db.
```

This instructs your tool not to collect profiling or coverage data for the methods from these packages.

5. Run the following command line to make sure the Java policy for the JVM from the JAVA_HOME is updated for the run with Rational PurifyPlus:

```
pstart <or qstart, or cstart> -setup
```

6. Modify the batch files for setting up the IBM WebSphere environment to use the same JVM as specified in the JAVA_HOME environmental variable. You can modify the original batch file `<WS Home>in\setupCmdLine.bat` or create a new one from it. You can call it `setupMyCmdLine.bat` for example and change the value for the line SET JAVA_HOME.

7. From the command line execute the modified batch file `setupCmdLine`.

8. **For IBM WebSphere 3.5:**
   Launch the admin server application by using the batch file `adminserver.bat` from `<WS Home>\bin\debug directory.
   Start the Administrative console and start the server.

   **For IBM WebSphere 4:**
   Launch the server by executing the batch file `<WS Home>\bin\startServer.bat`

9. When the server application is initialized and running, start the client part of the tested application. If you would like to record line level profiling and coverage information, please make sure that your Java components are built with the debug information. Record the snapshots of the run if appropriate.

10. Stop the client application from running and shutdown the Java application server.
Question:

How do I test J2EE applications running in BEA WebLogic 5.1, 6.0, or 6.1 with Rational PurifyPlus?

Answer:

1. Make sure to download the latest version of Rational PurifyPlus and install it on the server machine together with the BEA WebLogic Java Application server.

2. The newest proto-build for PurifyPlus and the point products Purify, Quantify, and PureCoverage can be found at the following location:


4. Set the system environmental variable `JAVA_HOME` and assign the path to the JRE installation directory to it. For example, on my machine it was:

   D:\Program Files\JavaSoft\JRE\1.3.1.

5. Set the system environmental variable `_JAVA_OPTIONS` and assign it the following value:

   `-XrunPureJVMPI:<tool>`

   The tool can be either `quantify`, `purify`, or `coverage`, depending on what Rational PurifyPlus tool you want to use to test your Java components running in the server. This variable will cause the selected PurifyPlus tool to launch automatically and start collecting the profiling or coverage data as soon as the Java application server starts.
6. Open the ASCII file Profile.ini from the installation directory of the tool you selected in Step #4 and append the following list of Java packages to the pre-filtering list:

   a. weblogic.
   b. com.sun.
   c. COM.
   d. com.beasys
   e. sun.
   f. bea.

   This instructs your tool not to collect profiling or coverage data for the methods from these packages.

7. Run the following command line to make sure the Java policy for the JVM from the JAVA_HOME is updated for the run with Rational PurifyPlus:

   pstart <or qstart, or cstart> -setup

8. Modify the batch files for setting up the BEA WebLogic environment to use the same JVM as specified in the JAVA_HOME environmental variable. Please note that you need to use eight character names in the batch files. The example path from Step #3 would be:

   'JAVA_HOME=D:\Progra~1\JavaSoft\JRE\1.3.1'

   Check to see the condition for checking the JVM used in the batch file for starting the server checks for java.exe and not for javac.exe as specified by default.

9. Start the Java application server. The selected Rational PurifyPlus tool will launch automatically and start collecting data for the run.

10. Start your client and launch the Web application to be tested with Rational Purify. If you would like to record line level profiling and coverage information, please make sure that your Java components are built with the debug information. Record the snapshots of the run if appropriate.

11. Stop the client application from running and shut down the Java application server.

12. Analyze the results recorded during the run of the tested Java application.

For more information, contact Rational Software Technical Support.

For more information on the products or services discussed in this
article, please click here and follow the instructions provided. Thank you!
Getting the Most from Your Automated Testing Tools

by Laura Rose
Quality Assurance Manager
Rational TestManager

In today’s competitive software market, customers expect quality products, and they gauge quality according to the frequency and severity of post-implementation problems. What’s the key to reducing these problems? The answer is deceptively simple: Test the software sufficiently and correct the sources of error before the product is shipped.

As every tester and quality assurance person knows, however, the actual business of testing for, and then correcting, errors is not a simple matter. Here are just a few of the factors that make it complicated:

- Statistics show that even the most carefully constructed code averages one to three defects per hundred statements.¹
- Studies indicate that inspections can uncover about 60 percent of total product defects; the remaining errors show up in application testing.²
- Application testing is a labor-intensive and expensive undertaking. Studies show that it consumes at least 50 percent of total product labor costs.³
- Manual test efforts tend to find the majority of defects at the end of the release effort or during beta testing, where the errors are more expensive to fix.
- Few programmers like testing, and manual tests are often executed inconsistently.

Automated Testing: Filling the Quality Gap

The past five years have seen a rapid proliferation of tools that address some of these issues by testing software automatically. Automatic testing tools can
detect errors early in the development process. The earlier the detection, the
easier and cheaper these errors are to fix, and the less impact the correction
cycle will have on the customer. Figure 1 shows a benchmark comparison of
manual vs. automated effort for various test steps. The testing involved
1,750 test cases and 700 errors.

<table>
<thead>
<tr>
<th>Test Steps</th>
<th>Manual Testing</th>
<th>Automated Testing</th>
<th>Percent Improvement with Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Plan Development</td>
<td>32</td>
<td>40</td>
<td>-25%</td>
</tr>
<tr>
<td>Test Case Development</td>
<td>262</td>
<td>117</td>
<td>55%</td>
</tr>
<tr>
<td>Test Execution</td>
<td>466</td>
<td>23</td>
<td>95%</td>
</tr>
<tr>
<td>Test Result Analyses</td>
<td>117</td>
<td>58</td>
<td>50%</td>
</tr>
<tr>
<td>Error Status/Correction Monitoring</td>
<td>117</td>
<td>23</td>
<td>80%</td>
</tr>
<tr>
<td>Report Creation</td>
<td>96</td>
<td>16</td>
<td>83%</td>
</tr>
<tr>
<td>Total Duration (Hours)</td>
<td>1090</td>
<td>277</td>
<td>75%</td>
</tr>
</tbody>
</table>

1Taken directly from the November 1995 issues of QA Quest, The Newsletter of the Quality Assurance Institute. Note that percent of hours may be a more valuable benchmark for individual organizations than actual hours.

Figure 1: Hours for Manual vs. Automated Testing

As you can see, automation makes a significant difference in all areas of
testing, especially executing tests and producing reports.

But how do you use automated testing to achieve a maximum positive impact
on overall product quality? We'll discuss some approaches below.

Test Early

In a good software development process, test planning and design parallels
the steps needed to produce the working code. Product quality begins with
documenting rules and restrictions that define what the software must do
(functional requirements). Once you complete the first draft of your functional
specification, you should evaluate it using four criteria:

1. Completeness
2. Consistency
3. Feasibility
4. Testability

Pay special attention to item #4. During the functional specification phase,
you should determine your test automation strategies. If you keep test
automation at the forefront of your thinking, then you can easily shape your
As a software development project evolves, it's important for testers to be aware of what is under development so that they can introduce automated tools at strategic junctures. Early iterations and prototypes often undergo only ad hoc, superficial usability testing. But once a product design has been selected and the specification is in place, testing becomes more serious. The focus is on features. Ideally, each feature is tested with various inputs and constraints, and several iterations are required to fully cover the product. On most projects, however, it soon becomes obvious that the testing team cannot possibly examine all interactions between all components manually in a timely and cost effective way. This is where automation really shines. It is a viable solution to these complex testing challenges. In a mature testing environment, automation can help provide quality measurements, useful analyses, and optimization of staff testing time.

Too often, teams wait until the features are complete before hauling out their automated testing tools. At that point, they are racing delivery deadlines, so they restrict their activities to regression testing. This not only leads to serious post-implementation problems for customers; it also robs the test team of an opportunity to get their money's worth from those tools. The earlier you incorporate an automated test approach into your development cycle, the greater the return on your investment.

Understand Your Testing Environment

Many factors affect testing during the software development lifecycle, but let's look at why it's important to have in-depth knowledge about -- and control over -- four factors that can ultimately have a major influence on product quality.

Know your staff. Each staff member has different strengths, and it's worth taking some time up front to assess these before deciding how to assign various testing activities. In addition, because automation changes how work is done, you can expect complementary changes in people's jobs. Some may disappear; other, more technical, jobs may materialize. Still other jobs may change focus. Accept those changes and try to match up each person's skills with these changing roles. If necessary, provide new skills training for your staff to lay the groundwork for success.

Know your customer. If the goal of testing is to reduce the frequency with which your customer will encounter a defect, then your tests should ultimately reflect the customer's environment:

1. Mirror the customer's configuration in the test lab.
2. Match the test data to the customer's environment.
3. Make sure the workload characteristics correspond to the customer's performance and load requirements.

Know your product. The more background you have on the product you are testing, its design emphasis, and its customer base, the closer your test cases
will be to the real world use. In a good software development process, test planning and design should parallel the steps needed to produce the working code. The test plan should start during the generation of the requirements.

**Know your test tool.** Figure 2 illustrates the four basic categories of automated test tools:

1. **Static tools** analyze the application's source code.
2. **Dynamic tools** actually run the application to inspect real output: screens, reports, database records, and input/output (I/O) activity.
3. **Test management tools** assist in managing the test process.
4. "**Other utilities**" is a catchall category for software that does not perform or manage testing, but in some way aids the test process.

![Figure 2: Taxonomy of Automated Test Tools](image)

It's important to understand which category your tools fit into because there are several different test phases in a product cycle, and some tools are useful only for specific phases. Dynamic tools, the most versatile, may be used in all major test phases, as we will discuss below.

**Know your data.** It is very important to control the application data you are testing. To get meaningful test results, you must ensure that you are inputting consistent data to the test cases. If developers and testers are allowed to independently add or remove information from a test database, chaos may result. You won't know which version of the data you are testing, and it will not be clear why you are seeing variations in the results.

**Modularize and Reuse Test Cases**

A significant cost of test automation is keeping up with the inevitable changes associated with normal software development. One method for minimizing the effect of changes to the product and test lab environment is to modularize the test cases. To do this effectively, you must plan and design your test cases ahead of time. Be sure your feature specifications are complete enough to create the test plan and test cases in parallel with feature coding.

Choosing the proper automation strategy for each different test phase can also help you cut costs because you can leverage your automated tests for reuse. There is no one method that fits all test types or test objectives. Although dynamic tools can be used in all test phases, their implementation
will not be the same for each phase. If you carefully tailor your strategy to each phase, then it may be relatively easy to adapt your tests for reuse during the same phase on your next project.

**Use a Configuration Management (CM) System**

Tests, like development code, change throughout the development and maintenance cycles. Therefore, it is wise to use a Configuration Management (CM) system for test code.

- CM dictates the rules for storing and managing the software, hardware, and tools used in the development cycle. These rules give you traceable control of changes during the different phases of development.
- CM also helps you coordinate the efforts of the development and test project teams. It allows the teams to change code and test scripts without creating confusion or hampering each other's work. Additionally, it provides control over baseline and intermediate releases of the software, test scripts, documentation, and development tools.
- CM also allows everyone to share test libraries and test data. Developers and testers can execute the same test cases simultaneously, in different configuration views or working directories, without overwriting each other's test results. This reduces test time in two important ways:
  1. The engineers can run the same tests cases prior to their code submission, allowing them to catch the errors before the final software build.
  2. Testing on different hardware platforms can be done in parallel instead of serially.

**Don't Reinvent the Wheel**

Automated test tools can provide tremendous productivity gains for a software development organization. Unfortunately, many software developers waste time reinventing the testing wheel. On most projects, test tools are the last requirements to be considered, and then they must mesh with already-specified development tools. Plus, the budget and schedule are often stretched, leaving little of either for bringing testing tools online. As Richard Morin explains:

> Although some commercial packages for bug tracking and regression testing are available, most companies write their own. This might have to do with local conditions (assumed or real), deficiencies in the commercial offerings, or cost factors. Regardless, the result is that many programmers currently are busy writing and maintaining private versions of these software systems. What's worse, these systems really aren't solving the problems at hand.5

If you select the proper family of automated test tools, however, you can design your software product, from the beginning, to take advantage of the tool's entire feature set.
Share Unit Tests with Developers

Historically, to produce high-quality software, you had to test each function, module, or class (in object-oriented programming). This practice, called **unit testing** (or module testing), while effective, is extremely time consuming and labor-intensive, and is usually performed by the software developer.

Automated test tools offer a way to make this process easier. Capture/Playback is an essential component of rapid script development, and most Capture/Playback tools can automatically collect all user interactions with the application into an easily edited test script that is later used for playback. So if you use a Capture/Playback tool, a test browser, and a CM system, you are already on your way to creating reusable tests.

In most organizations, development engineers both outnumber test engineers, and know more about problematic areas. Often, testers don't take full advantage of developers' testing knowledge. After all, these people have already performed much of the testing, even though their tests are typically designed to be temporary and disposable.

Testers and developers can coordinate their efforts if developers design both program and unit tests for modularity, and then incorporate these tests into a regression suite or library. They can also write basic utility scripts and configure them into a library. Then, developers can browse through these components and use them to create unit tests. The code for these tests can also be configured into the same library. The unit tests can be combined or reworked into integration tests to be shared throughout the development cycle.

With some test automation tools, you can modify scripts to include external routines, and the software department's shared utilities (error routines, debug routines, message routing, etc.) can be configured directly into the test. These tools are extremely flexible, and the tests they create can often be used in more than one test phase.

Create Playback Scripts and Leverage Comparator Features for Regression Testing

As stated in the IEEE Standards,

> **Regression Testing** is the selective retesting [of software] to detect faults introduced during modifications of a system or system component, to verify that modifications have not caused unintended adverse effects, or to verify that a modified system or system component still meets specified requirements. (IEEE 1990).

In other words, regression testing answers the question, "Does everything still work after my fix?"

Tools can automate the regression testing phase of the development cycle and allow you to utilize your testing resources efficiently (and avoid the monotony and error-prone results that characterize manual quality testing). Tools enable
you to deliver high quality software with tremendous time and cost savings.

Most automated test tools capture the right answer from a previous version of the software and store it. During regression testing, the test scripts are rerun, and their actual results are compared with the previous version's expected results (see Figure 3).

![Capture/Playback Overview](image)

**Figure 3: Capture/Playback Overview**

To perform reliable regression testing, the environment of the system under test must remain the same between the time of the original recording and the playback. One way to achieve this is to create a script that records the state of the original test system. Then, prior to regression testing, play back the script to bring the system to the original state.

Since regression testing implies the use of a comparator (a program that compares the actual with the expected results), the more flexible the comparator, the less rework is required. Some features of screen comparators to look for include:

- Ignore a field or region.
- Include only a field or region.
- Ignore video attributes.
- Ignore color shading or change in color palette.

By programming to automatically ignore information that changes from one release to the next, you are reducing maintenance for the expected results. For example, the release or version number changes with each release, as do the date and time stamps on screens and files. If this information is captured in the expected results file and compared with the next release, then it will fail. If you take advantage of comparator features to mask those fields, however, then the test will not fail.

**Combine Unit Test Scripts for Integration Testing**

Testing a specific feature together with other newly developed features is
known as integration testing. Testing the interface of two components is a way to explore how components interact with each other. Integration testing inspects not only the variables passed between two components, but also the global variables. This test phase assumes that the components and the objects they manipulate have all passed their local unit tests.

Previously captured unit test scripts can be combined, with minimum effort, to create a variety of integration test cases. For instance, a unit test script that tested an ADD function can be scheduled with other unit tests for DELETE and COPY to create an integration test of the entire file maintenance system with little rework.

**Conduct Non-intrusive System Testing**

**System testing** is designed to reveal bugs that cannot be attributed to either individual components or the interaction among components and other objects. System testing studies all implementation aspects of the design that are similar to those in the customer's environment. The issues and problem behaviors it targets can only be exposed by testing either the entire integrated system or a major part of it. System testing includes testing for performance, stress, security, accountability, configuration sensitivity, usability, data integrity, start-up, and recovery.

Verifying these characteristics is especially important for products intended for users on diverse OS and hardware platforms. You can test a variety of platforms using the same test scripts and capture/playback record files, then compare the results. That way, you can readily associate the causes of variations with either a specific hardware platform or a generic problem.

Since **system testing** is aimed at testing the product as the customer will receive it, the test tool you use at this stage of the product development cycle should be non-intrusive. If the test tool requires the software application under test to be built or linked with specific runtime libraries or include files, then you are not testing the end product. A true non-intrusive approach requires no modifications to the test environment or application, and no special hooks into your application or libraries.

In today's marketplace, you can choose between intrusive and non-intrusive technologies for virtually every phase of testing. Rational recommends non-intrusive technologies for all test phases in the development cycle; they help you guarantee that the application you test in the lab is the same application you will deliver to your customer.

**Stress Test for Multiple Users on Different Machines**

**Stress testing** determines whether a program can fulfill its defined requirements and work as it should under extreme conditions. Automated test tools can easily simulate multiple users and emulate several machines and applications. This is critical for both Client/Server and UNIX applications; the application itself might hold up fine with a 500-user load, but some hardware platforms might have lower load limits.

Avoid hard coding the number of emulated users within the script; instead,
pass the size through environment variables, parameters, or test drivers. This allows the same set of test scripts to be used on other machines with different load limits.

**Use Load Testing Tools for Performance Testing**

A software performance problem can cost millions of dollars in downtime, so you need *performance testing* to verify that an application meets specific performance efficiency objectives. Automated load testing tools can help you evaluate performance and response times by delivering a *workload of user activity* to your application. They then measure the application quality and response time that will be achieved in the actual environment. These automated test tools emulate *real* scenarios in order to truly test system performance and software functionality.

It's best to use a flexible test case scheduling mechanism so that you can easily specify and modify the order of the tasks, job mix, and rate at which jobs are submitted. Without this, it is difficult to reproduce the real workload, since the rate of test case submission is a critical component of the workload. This scheduling mechanism should be independent of the test cases, so that the workload dynamics can be modified without changing the actual test cases.

To avoid load dependent results, use the tool's text matching capabilities instead of matching on timing characteristics. For instance, match on the text response "DONE" instead of a response that returns in two seconds. Matching on timing characteristics means that the benchmark execution will be load dependent, and therefore not repeatable (since the load changes as you increase the number of users in a network environment or vary the configuration parameters). Investigate the use of synchronization mechanisms such as "wait on events." These mechanisms can be used to provide mutually exclusive access to critical data regions, as well as user cooperation on concurrent tasks.

**Let Developers Do the Sanity Testing**

*Sanity testing* is used to verify that the software build is ready for System Test (or more extensive testing). The tests are very general, targeting the most frequently used functional areas. If an application does not pass the Sanity Test suite, then further testing activities are suspended.

If testers automate and share these tests with the development staff, then development engineers can quickly execute the tests prior to a software build to ensure that they are not handing off an unsatisfactory release. This eliminates several time-consuming cycles, including:

- Building the product.
- Handing the product to QA.
- Doing formal QA testing to discover and isolate the major error.
- Sending the product back from QA to engineering.
Collect Field Diagnostics

Capture/Playback techniques are invaluable to customer support and field technicians. This technology collects all user interactions with the application (cursor movements, button activity, keystrokes, think time delays, and so on) into an editable test script that can be transmitted back to the development organization. Support staff can play back the scripts (with minimal modifications) in their test lab, emulating the customer's exact steps. This eliminates speculation about what actually transpired. The same customer scripts can be incorporated into the automated regression test suites, making yesterday's problems today's verification tools.

Use Test Scripts for Vendor Calibration and Performance Verification

Another service automated test scripts can provide is vendor calibration and performance verification. When vendors receive software or hardware, they can run the same performance test scripts. These reports can be used as proof that the vendor-supplied items are working at agreed-upon levels.

You can use the same scripts to offer yearly performance maintenance runs on your product. Supply these reports to the customer as proof that the software is working at the same level it was when they purchased it.

Use Test Scripts for Loading and Lab Setup

Automation can assist in several manufacturing procedures, including installation, initialization, and verification. Different scripts can be called to load environments, based upon customer configurations or software versions. These scripts can then be used to perform the final check before the equipment and software are shipped to the customer.

Challenges and Limitations of Automated Testing

Test automation cannot bring order to chaos. Even the most advanced tools will work only as well as your process. If your organization has no active quality policy, no configuration management, no documented requirements, then even the best tools will disappoint. On the other hand, if your process is organized and continually working to improve quality, then automated tools can be extremely successful in reducing test cycle times while increasing test coverage.

Most applications include areas that are especially difficult to test: interrupt handlers, exception handlers, critical sections relating to multitasking and relatively rare conditions (such as February 29th). To thoroughly test these chunks of code, it may be necessary to set up pre-test conditions or use debug techniques to set values, and this can be challenging.

Naive user and usability tests are also hard to automate. Automated tests run the same way each time (or travel the same paths), but different customers can use the product differently. It is impossible to cover all combinations. Normally, automated test scripts follow known paths. Once a new user discovers an unexpected path, however, you can update the
automated tests to include it, thereby improving your regression suite coverage.

The Redesign Challenge

Another limitation of automated testing is that when a new version of the software is released, the test scripts no longer work without redesign. Because test redesign was never built into the release schedule, often entire automation systems (representing thousands of hours of work) collapse, and the projects are completed with inadequate testing. A similar problem can occur with new platforms or operating systems.

When this happens, you may need to re-evaluate several processes:

Test Planning and Test Design. At the start of a project, all potential platforms and features should be incorporated into the test design. Also acknowledge that software features and platforms will change and require test script modifications, and incorporate test maintenance into the development schedule. The task of designing and coding reusable automated tests for the full product cycle is complex, and must go beyond a capture/playback mentality.

Also, remember that automated test tools are not the entire solution. Tests can be executed by an unattended machine, but planning and development must be done by a qualified engineer. Good software test development is sophisticated programming and requires testers who are comfortable in this environment.

Scheduling. The use of automated test scripts requires that even small changes to the software product, anywhere in the development and maintenance phases, be carefully considered. Even the smallest change can impact test scripts or require additional test design and implementation to cover the new features or enhancements. This test case maintenance is an ongoing effort.

Test Tool Selection. Frequently, development teams select test tools either after the product design phase or without even considering the product that is being designed. Those tools might not be right for your product.

The Human Factor

Knowing where each member of your team stands with respect to test automation knowledge and experience is vital to the success of a new implementation. You can easily destroy an employee's morale if you assign him automated testing tasks that don't match his knowledge and abilities.

Also be aware that automation changes an organization. Interactions between testers and the rest of the software organization change. With automation, testers can do more for developers, release engineers, and customer service representatives. Sophisticated testing changes "educated guessing" into engineering. More people can run more tests earlier in the software development cycle. Testers can concentrate on more complex testing and analysis. But be prepared: Expectations change, too. Managers will expect
higher quality code and faster deliveries. If you select the proper test tool at the start and implement it use early in the development cycle, however, you can meet those expectations. You will see tremendous productivity gains and greatly increase the return on your investment.

Notes

4 From QA Quest, *The Newsletter of the Quality Assurance Institute*, November 1995. Note that percent of hours may be a more valuable benchmark for individual organizations than actual hours.

References


*For more information on the products or services discussed in this article, please click [here](http://gdbdoc.gdb.org/dev/qa/instl_qualTOC.html) and follow the instructions provided. Thank*
you!
Build It and They Will Come

by Joe Marasco
Senior Vice President
Rational Software

In the world of software products, there are successes and failures, determined by the free market system. We must, of course, add to the list of failures those projects whose products never see the light of day -- the ones that are worked on for various lengths of time but never "ship." As obvious as it sounds, you cannot be successful unless you meet the precursor of shipping your product.

As you cannot ship what you cannot build, actually putting together the pieces becomes critical. This article talks about the issues involved in creating a repeatable build process for your product.

In the Beginning There Was the Sandbox

Products come out of projects, and projects tend to begin in haphazard ways. Organizations with well-defined processes have developers building their components in local work areas, sometimes called sandboxes. They provide for mechanisms whereby the subproducts of these sandboxes can be assembled, sometimes in ad hoc ways, so that each development team can test its progress in the context of the whole product. Configuration management systems allow for appropriate partitioning such that each developer (or team of developers) has the autonomy and isolation to work on his piece without stepping on the other guy's toes, while at the same time providing for a loose integration context.

This works fine in the early, chaotic days when everything is changing very rapidly, and before architectures are well defined and interfaces are nailed down. However, before too long even modest projects outgrow this framework. At that point, one of two things happens: Either the organization makes the build a priority and adds some structure, or it doesn't. In general, those that do establish a regular "heartbeat" for the project -- a periodic, regular, dependable, build cycle -- improve their
chances for success. Those that don't establish this rhythm find that entropy begins to take over, and that building the product becomes more difficult over time.

Many organizations vastly underestimate the effort it takes to put a good build process in place. Because of this, projects in their latter stages often have a "new" problem to deal with: In addition to having buggy software, incomplete parts, and so on, they also struggle with something that they have taken for granted -- the "simple" assembly of their product. This is a trap for the unwary. In order to not fall into the trap, you need to understand more about the process of assembling a product.

**Why Should the Product Build Be Hard, Anyway?**

First of all, the product you are going to ship to other people has more pieces to it than the prototypes you have been putting together for internal consumption. A classic example: Developers and testers rarely look at the "help system," as they know the product well enough to play with it and test it. Once you are going to have outsiders try to use it, you need a well-elaborated and working help system for people to use. Further, you need instructions for installing the software in different computing environments, as well as various other adjuncts that you can live without when you are only consuming your software internally. So the first problem that comes up is one that might be dismissed as "packaging." You need more pieces to ship a product than to use it internally, and further, you need to document all the little details that the internal team has always "known" or taken for granted. Making the product ready for outside consumers is sometimes called "sanding off the rough edges." Some of these "rough edges" can be very sharp, and because you don't catch them all the first time, your first consumers may cut their fingers on them.

Let's assume, however, that this is just a logistical exercise, and that with enough planning you can avoid the "packaging" trap. In some sense, it can be put in the "annoying detail" category: If you ignore it, it will bite you, but if you are aware of it and plan for it, then it is relatively easy to overcome. So, forewarned is forearmed: Treat packaging as a purely technical problem, and you will be fine.

In fact, there are three much more fundamental obstacles to success that come up over and over again. They are distinct and interrelated, and all three must be worked on to achieve a successful build process.

**Obstacle #1: Organizational Politics**

Many software development managers lose sight of the simple fact that controlling the build process is first and foremost a political problem. To put it simply, he who controls the build has an enormous amount of power. After all, the build cycle itself defines the rhythm of the entire development and test organization. Think of the build cycle as the software equivalent of a factory assembly line. The person who gets to define the characteristics of the line and its speed determines, to a very real extent, the output of the factory. Line workers are very aware of their
subservience to the line. The cardinal sin in the factory is to slow down, or - Heaven forbid! -- shut down the line. The software equivalent is submitting a set of changes that "breaks the build."

Now the build process is something that everyone must participate in but only one group can control. By its very nature it is not a democratic enterprise; it requires a certain amount of hierarchical and structural apparatus to work at all. Everyone agrees on this, more or less. The sticky wicket is determining who gets the responsibility and authority to make it work. For that group will, from that day forward, wield a lot of power and clout.

Because human beings are, in general, reluctant to give up this sort of power, the build process becomes a political football. Myriad discussions ensue as to who will have the right to do what to whom in the interest of the build process. All of the negative political tendencies of your organization will be exposed during these discussions.

The purists among you will cry out that political tendencies should be discouraged or even condemned, pointing out that the job is hard enough from a technical point of view, and it should not be "polluted" by politics. In most organizations, however, wishing politics away will not necessarily make them go away. Politics is a fact of life that must be dealt with. However, you must get through this phase, as unpleasant as it first appears. Else, you will be incapable of dealing with the next two hurdles. Here are some specific suggestions:

1. Try to get the group to agree that someone has to be in charge, that a loose confederation approach is doomed to failure.
2. Try to reach a reasonable compromise between the autonomy of the constituent teams and the centralized authority that will be required.
3. Always make sure that the management team understands the importance of the issue and has the very best people assigned to the build.
4. We will talk below about having a "czar of the build." Make sure it is a person who is technically competent, firm, fair, and respected by everyone. Install him or her early in the process and have this person guide you through the political shoals.
5. Enlist management's support in crushing "bad politics" should it rear its ugly head.

**Obstacle #2: The Process**

Having hacked through all the political jungles that accompany conceding power to the build group, the participants must now agree on the process they will use. Just as form follows function, the "process" will often be shaped to mirror the political compromises that were made to get to this juncture. There is plenty of interaction between the first and second obstacles. In fact, often the process obstacle presents itself early on, in phase one, because it is being used as a surrogate by those who don't
want to openly admit that there are unresolved political issues. In some organizations, we see these two obstacles mushed together into one giant hairball, which in turn gives "process" a bad name. You cannot use "process" to solve what are intrinsically political problems, much in the same way that you cannot "solve" technical problems through political compromise.

The basic tension at this point revolves around the people who want a strict, rigorous process -- sometimes called "lots of rules and no mercy" -- versus the people who want a "looser" set of policies. Acknowledging that there is no single, simple, right answer is usually the best place to start here. Your process will have to be tuned to your organization, because all organizations have their peculiarities.

That does not mean that you need to invent new process. I used the word "tune" in the above paragraph, because I am firmly convinced that the best way to deal with this issue is to start with a base process that has been demonstrated to work before. Unified Change Management (UCM), for example, has a rich legacy of successful application. We know it works across a broad spectrum of domains, applications, and organizations. Why start over? Do you really think you are going to do better?

There are a few traps you don't want to fall into at this point. One is the "religious wars" pitfall. In every organization there are "process gurus" who believe that they, and only they, have the magic formula. And, sure enough, every time, there are others who resist, quite certain of their own convictions. Regardless of who is right or wrong, these crusades are totally unproductive, often revolving around obscure details of little import. The strong manager needs to identify the religious process fanatics and stifle them early. Sometimes the only answer is to tell them to put a cork in it. Remember always that process is not an end in and of itself; it is a means to an end -- shipping product!

Another trap is to think that any process, no matter how good, can substitute for thought or judgment. For every "ironclad rule" there is bound to be an exception. You will have to watch what is going on and make midcourse corrections, no matter what your process is. As called out above, you will need to modify and tune your process in real time as you discover what works for you and what doesn't.

Lastly, get on with it. Perfect is the enemy of good. You will develop your process iteratively, just the way you develop the software. Get to iteration one quickly. Learn. Change. Improve. Repeat until done.

Obstacle #3: Tools

Just as the first obstacle (politics) and the second obstacle (process) are intimately related, so are the second and the third. The third, of course, is the toolset that you will use to implement the process. Needless to say, choosing the tools first is getting it bass-ackwards, but surprisingly enough, that's the way many organizations go about it. They then wind up with the tool determining the process, which can be loads of fun when the process thus derived is inconsistent with the political philosophy of the
Obviously, you need tools that will automate and enforce the process you have chosen to use. If you have a process that admits mistakes, you will be "backing out" changes from time to time. Does the tool support that easily? Are developers going to be checking in their work to a common baseline from multiple remote sites? Then your tool had better support that model. Do you want to build your entire product from top to bottom every night? If so, then I hope your tool has the performance and turnaround characteristics that will permit that. Do you want to automate your regression testing as part of the build? Once again, tool support is crucial.

Even organizations that have done a good job with the first two problems sometimes flounder with the third. And sometimes it is not the tools' fault either. Once again, using our factory analogy, you need someone to monitor the line, and to do quality control for the product coming off the line. Without constant vigilance, it is easy to automate a process that produces a low-quality result. Every successful build process requires a "foreman" or the equivalent thereof; sometimes he or she is called the "czar (czarina?) of the build," or more simply, "the buildmeister." The buildmeister monitors the health of the line and makes sure that a steady stream of good quality product is produced.

One last semi-technical note: Beware of the old saying: "We can always write a script that can do that." It is true that scripts can be written to do almost everything, just as duct tape can be used to stick almost any two things together. The problem is that these scripts always start out small and simple, and then grow in ways that are random and unsupervised. Scripts, unlike programs, are rarely "designed"; they just grow. They become inadequate to the ever-increasing demands of the organization; they are brittle. They are a maintenance nightmare, especially if the original author moves on. And they are very, very difficult to debug. Just as the road to Hell is paved with good intentions, the road to "build Hell" is paved with the out-of-control products of general-purpose scripting languages.

**What About Iterative Development?**

In iterative development, we avoid one of the great pitfalls of the waterfall approach: leaving system integration to the last minute. One of the reasons so many waterfall projects fail is that, very late in the game, developers are trying to assemble their product for the very first time. In addition to finding many bugs, mostly in the interfaces, they grapple with the normal logistical and organizational problems of putting together a build chain for the first time. Often, things that pass for "bugs" are nothing more than the artifacts of broken builds. But the organization is in such chaos at this point -- running out of time, nothing working, people frazzled -- that it is hard to separate the sugar from the salt. It is also a very bad time to be trying to solve political and process problems.

By contrast, iterative development requires that you construct your build chain to accomplish the deliverable for iteration one -- a working program.
So you begin to debug this process early in the project, not at the end. By the time you get to iteration three or four, the build process is actually starting to work pretty well. For the last iteration, the one that will deliver the final bits, the build should be working like a finely lubricated Swiss watch.

**Parting Thoughts**

As with pretty much everything else in software development, there are a small number of ways to get this right, and almost an infinite number of ways to get it wrong. If you view "the build" as a detail that will "just happen," then the odds are against you. Make sure that you attack the build process as a conscious effort that is critical to your success, and devote the time, energy, and resources to it that it demands. To do any less is sheer folly.

---

**Notes**

1. Entropy is the tendency that all systems have to move from an orderly state to a disordered state when left alone. It is a fundamental physical law. One might say that all attempts at progress, by any civilization, fly in the face of entropy. Another way to say this is that to bring order out of chaos takes work, and that once you stop working, entropy will cause the system to spontaneously move to a more disordered state.

2. The standard vehicle for this is called the "release note." The release note documents the limitations of this version of the software, known bugs, and so on. It is an attempt to characterize the state of the deliverable, as it is better to tell your consumers about things you know about rather than have them discover them on their own. Sometimes the release note is called the "readme" file.

3. This subject is so important that next month there will be an article in this space talking about politics in the software workplace. My perspective is that there are "good politics," akin to the notion of "fighting fair," and that a healthy political process can and should work toward making good decisions. Then there are "bad politics," which make organizational objectives subservient to personal agendas and self-aggrandizement; this sort of politics needs to be stamped out wherever it is found. The problem, of course, is the gray zone in between. But more on this next month.

4. I believe the world is indebted to James E. Archer for this characterization.

5. Some people argue at this point that you should endeavor to get your process "right" and then tune your organization to fit the process. While this is a laudable objective and theoretically the right approach, I have rarely found it to be successful in practice. You cannot allow a regressive organization the prerogative of rejecting reasonable process; on the other hand, it is difficult to implement any process that is too far out in front of the organization that must carry it off.

6. To illustrate how far out of control this can become, the wars are often characterized as struggles between the "process Nazis" and the "anarchists." With such value-laden labels, it is difficult to have discussions that will get to the right place.

7. In a like manner, the anarchists will be hard put to demonstrate that they can ship product without any process. As is the case in almost all these debates, neither extreme position is defensible.

8. Old Russian saying, first introduced to me by Mikhail Drabkin of Riga, Latvia.

---

*For more information on the products or services discussed in this article, please click [here](#) and follow the instructions provided. Thank you!*