# The IDE as a Development Platform

Rational. software

IBM

# Contents

# Empowering the A in SOA

*IBM Rational Software Delivery Platform v7 products bring an unprecedented level of governance to the SOA development process with built-in process guidance and architectural management and quality management designed into the tools you use for delivering service-oriented applications*

**By Rikki Kirzner**

**IBM Resources**

***Rational Software Delivery Platform - desktop products v7***
***SWG TV***
***Webcast: Architecture Management***
***IBM Downloads***
***Webcast: Quality Management***
***Competitive Trade-up***

By now there should be no doubt in your mind that service-oriented architecture (SOA) is the type of disruptive technology that will influence dramatic changes in how software is created, reused, recombined, managed, deployed, and sold. Accordingly, experts have been talking about the advantages of services in terms of reusing existing software assets by encapsulating them into business services that are used over and over again to create new virtual applications. There is a growing desire among software industry professionals to learn how to make SOA a reality and obtain these benefits. Perhaps that is why there has been very little actual discussion of the key component that enables SOA-the A of SOA-architecture and its management.

Architecture management plays an essential role in keeping software aligned with the changing requirements of business and the reality of imperfect implementations. The full benefits of reuse and other advantages of SOA can't be realized until you have a real services-oriented architecture in place in your organization. Wrappers using SOAP and other Web services technology are a useful-albeit somewhat less than ideal-approach to creating services. Unfortunately, they don't replace the necessary steps you have to take toward building the architecture and obtaining the tools, processes, and best practices that ensure your architecture remains intact and unassailable as requirements change and business conditions and evolve.

Furthermore, most organizations development efforts are adversely impacted by geographically distributed development teams often arising from one or more of the following conditions:

- Increasing practice of outsourcing portions of the development effort to achieve the economic benefits of a skilled labor force with lower salary and benefit requirements
- Large globally distributed organizations with multiple development sites in different cities or countries
- Mergers and acquisitions where development teams from the original companies are not relocated to one central location thereby allowing each group to work on different aspects of development projects at different locations

Monitoring the progress of development becomes difficult and compromised when you are working with geographically dispersed development sites. Multiple groups of developers working on the same project in various locations require coordinated processes and tools so they can work together efficiently and synchronize their development efforts. Without these-chaos ensues.

## Architectural Management is Critical for Success

Architecture management delivers the stable architecture, tools, methodologies, and processes that let you accomplish the following:

- Manage the relationship between design models, actual coding, and implementation done at different locations
- Synchronize the code to the models and architecture design to reduce bugs and software defects before the code is tested and deployed
- Analyze the completed code to make sure it accurately implements the original design and requirements
- Facilitate the reconciliation of those changes that are still required for successful deployment
- Implement better quality management throughout the lifecycle

To function at peak optimization, SOA services must be independent of the context or state of any other services. They must be self contained and able to function seamlessly across geographically distributed development environments. When SOA is implemented properly it both simplifies the functional complexity of software systems and increases the agility and flexibility of the enterprise. In this way, SOA enables companies to combine their own components or components created by their partners, suppliers, customers, etc., to create business services that respond readily to changing business conditions, government or business regulations, and competitive requirements.

Consequently, SOA initiatives require strong architectural management and coordination of development efforts for optimal implementation and maximum operational efficiency. SOA development environments and the supporting architecture must enable engineers, designers, business architects and analysts, testers, and QA teams to communicate and coordinate their development efforts seamlessly around the clock and around the globe.

Good architecture management, therefore, becomes a logical extension of conventional design and construction shifting your focus to managing architectural changes and requirements that are aligned with geographically disparate development activities. It enables organizational-wide adherence to standards and business and government governance, best practices, and policies for creating and reusing services that are crucial to the success of any SOA project. It also promotes risk management procedures while controlling and coordinating the management of Web services and other software assets.

## New IBM Rational Products Enhance SOA Delivery

The SOA siren call for better tools, best practices, and architecture management has been answered with the latest release of IBM Rational Software Delivery Platform v7 products. New capabilities and improved lifecycle quality put you on the fast track to implementing your SOA initiatives while managing your architecture. New functionality empowers you to implement, coordinate, and manage the delivery of SOA projects-even across different geographic locations. These products' technologies and frameworks (listed at the end of this article) eliminate the silos between design, development, testing, quality assurance, and deployment. Improved lifecycle automation and traceability, easily configurable workflows, and extended globalization support give you a more flexible way to manage value and change while reducing the control and risk exposures inherent in globally distributed development and deployment.

## RSA turns SOA into a Holistic Development Effort

One of the IBM Rational Software Delivery Platform v7 products is Rational Software Architect (RSA). RSA unifies all aspects of software design and devel¬opment, architectural modeling and specifications, and J2EE development into one easy to use tool. Architecture management is achieved by pulling together and integrating essential prototypical and disconnected tools and best practices-effectively turning SOA into a holistic development effort.

Rational Software Architect leverages model-driven development for creating well-architected applications and services and facilitates your ability to comprehend, design, manage, and create SOA services. Built on top of the open and extensible Eclipse platform, RSA's integrated design and development capabilities manages change between models and code and related artifacts throughout the lifecycle, making it key for implementing architecture management functions discussed here.

Rational Software Architect's integration with the other products in the IBM Rational Software Delivery Platform delivers requirements management, traceability, source code control, and other team management functions throughout the lifecycle to reduce the risk associated with software develop¬ment. For instance, requirements stored and managed in Rational RequisitePro can be accessed, associated to corresponding modeling elements, and synchronized with user-selectable rules. Modeling files can be managed by Rational ClearCase, which ships with Rational Software Architect.

Additionally, there are some new, compelling features in version 7 to enhance your SOA projects. First, there is a set of new extensions to RSA including a UML profile that turns the general purpose modeling language into a domain

specific language that lets you use RSA in a SOA context.

Next, you will find new, common online and integrated process guidance capabilities in a SOA context for distributed development teams. This capability was the result of merging SOMA (Service-Oriented Modeling & Architecture), the specification and realization of business-aligned services that form the SOA foundation with IBM Rational Unified Process (RUP) for creating best practices.

RSA features a new transformation engine, making it easier to write transformations to better control how architecture flows into code and how to keep your code and models in synch. Improvements include model to model transforms, model to code transforms and code to model "reverse transforms, giving you the ability to analyze existing code and extract its design into a model you can compare against the original model.

RSA (and all of the v7 desktop tools) now employ a new installation technology that supports more granularity in the features presented, thus allowing for more options and flexibility to install only those components needed. They also offer an option to install into an existing installation of Eclipse v3.2 or to have the product install Eclipse v3.2 for the user. This means that you don't need to install a redundant copy of Eclipse if you already have Eclipse installed on your system. RSA will install the current version of Eclipse for you if you don't already have Eclipse. It also means you now get to pick and choose what tools you want to install from the many available options.

Finally, if you want to use just the Eclipse framework itself, but are concerned about doing mission-critical production work, relying solely on open community support, then IBM provides a service-only support for Eclipse option that is consistent with that which comes with its commercial products.

Additional IBM Rational Software Delivery Platform v7 products include:

- IBM Rational Software Modeler
- IBM Rational Application Developer
- IBM Rational Systems Developer
- IBM Rational Performance Tester
- IBM Rational Functional Tester
- IBM Rational Manual Tester

## IBM Simplifies Creation and Management of SOA Architecture

Your SOA projects will benefit significantly from the efficiency of version 7's new, combined integrated architecture, modeling, geographic collaborative features, and lifecycle quality management. IBM Rational addresses the fundamental requirements of SOA architecture management by simplifying your ability to link business and IT architectures to service models.

Modular and seamlessly integrated tools give you a faster and more structured approach for SOA development using robust architectural design and automated service delivery. Improved hardware and software modeling along with support for standards, open systems, and packaged applications make it easier than ever to create and manage large numbers of high quality services while still maintaining your SOA architectural integrity.

***Rikki Kirzner*** *is a freelance writer and veteran computer industry professional with experience as an analyst and former Research Director for IDC, Gartner Group, and Meta Group and as a Senior Editor with Open Computing Magazine. Rikki covers software, development, open source, SOA, and mobile computing.*

# Quality Management Is a Strategic Delivery Advantage

***Managing quality throughout the development life cycle ensures your applications satisfy all business specifications, customer needs, and governance requirements, producing fewer software defects, shorter release cycles, and lower development and support costs.***

**By Rikki Kirzner**

**IBM Resources**

***Rational Software Delivery Platform - desktop products v7***
***SWG TV***
***Webcast: Architecture Management***
***IBM Downloads***
***Webcast: Quality Management***
***Competitive Trade-up***

## Why Testing Needs to Morph Into Quality Management

For too long, testing and quality management has been more of an afterthought-a task usually handled by QA and test engineers toward the end of a software project. Using this approach is no longer feasible if you expect to deliver high quality software. The National Institute of Standards and Technology estimates developers spend about 80 percent of development costs on identifying and correcting defects. The Cutter Consortium reports that 32 percent of organizations release software with too many defects and 38 percent of organizations believe they lack an adequate software quality assurance program

In all probability, your job requires on-time delivery of high quality applications while juggling the pressures of shorter release cycles, increased technology complexities, the logistics and communications challenges of working with geographically dispersed and outsourced development teams and service providers-all while needing to comply with internal and external business and regulatory requirements. Trying to meet these criteria exacerbates the pressures to deliver quality applications demanded by your end users.

It is no wonder the scope and value of testing has expanded and evolved. It has become a requirement for automated processes throughout the release stages to validate quality, identify regression, and assess the state and value of software against the changing requirements of evolving business conditions. Only this level of quality management can ensure delivery of high-quality software on time and under budget. It also enables better risk management and governance compliance.

Quality management must be a strategic initiative embraced by everyone throughout your organization, across the entire lifecycle-from requirements and design, to development and deployment-to be most effective. However, that doesn't necessarily imply that you have to implement quality management all at once or across the entire company in one fell swoop. A quality management plan and solution can be applied in stages. Once that is accomplished, different components of the quality management solution can be implemented in increments that are comfortable to everyone involved. Although quality management produces the greatest ROI-and is most successful when it is embraced by the entire design, development and deployment teams-genuine benefits will be realized by each group that adopts some type of quality management solution.

Quality management begins with linking requirements to test cases, thereby creating a system of software defect checks and balances throughout the delivery phases. This includes defect tracking, test management capabilities, and automated builds to make sure you create a defect-free deliverable that is ready for deployment into production and capable of meeting all exit criteria.

Automated builds are an important feature because automated processes help reduce the time the QA team has to spend on finding or duplicating, and fixing problems in bad builds. Good builds enable the test labs to work more efficiently. When they receive a good build, they can focus their efforts on finding genuine defects rather than wasting their time testing for problems that result from a bad build, or having to redo all their efforts when a good build is finally delivered to them. Depending upon your organization and the stage within the project, automated builds speed up the process of deploying your software into production and meeting exit criteria.

Successful implementation of quality management entails the acquisition and implementation of a comprehensive and repeatable automated framework to accelerate software delivery and improve development and deployment team efficiency across the organization. Continuous and integrated automation across and between the software delivery (or pre-production) processes allows each process to leverage the artifacts of the previous process for improved efficiencies, targeted workflows, auditable trails, and process validation.

## The Right Combination of Quality Management Solutions

IBM Rational ClearQuest-alone or in conjunction with IBM Rational Performance Tester and the rest of the IBM Rational family of software quality solutions-minimizes your IT and development costs. They automate, analyze, and ensure the quality, reliability, scalability, and performance of your business-critical enterprise applications. They also deliver the right combination of comprehensive tools and best practices to help you be successful in managing quality across every phase of the lifecycle, around the clock and around the globe.

These quality management solutions create a common framework to facilitate team-based quality management practices to identify and resolve software defects and bottlenecks. They ensure applications perform and respond as required, track people and resources, and reduce overall risk. They facilitate the measurement and prediction of quality and performance against business requirements and objectives. The overall quality of software applications is tested before they are deployed to make certain enterprise applications perform and scale as required. Administration and communication processes are enhanced, giving you better visibility across software projects and that ensuring the right people are working on the right code at the right time.

## IBM Rational ClearQuest Eliminates Barriers to Quality Management

IBM Rational ClearQuest is a complete out-of-the-box solution with "design once and deploy anywhere" capabilities on all major platforms that should be the cornerstone of any quality management strategy. Its powerful workflow management system effectively manages quality across the full software lifecycle, from requirements definition through application deployment. It automates much of the software development and delivery process to virtually eliminate the geographic and organizational communication and informational gaps that wreak havoc on development projects and delivery schedules.

You get a consolidated, real-time view of your development and testing processes, even if you are working in a heterogeneous environment using disparate processes and procedures. Development and QA teams can work together more efficiently, sharing common and complete views of all testing and development activities-especially traceability between development, testing, and other project assets. You can customize and enforce testing processes and workflows, as well as seamlessly coordinate work across distributed locations.

IBM Rational ClearQuest advances the overall quality of the delivery process using a common software platform for managing software development changes. Its fully customizable interface and workflow engine easily adapts to any development process and supports project of any size. Robust and error-free replicating and synchronizing mechanisms ensure centralized or geographically distributed development teams get instant access to defect and change data. A secure, centralized repository provides a traceable, auditable relationship between development, testing and project information, effectively linking requirements, code, build records, test cases, test results, deployment records, and other pre-production assets.

IBM Rational ClearQuest lets you easily discern what requirements were implemented, what code was updated, which test cases were run, what the test results were, who performed specific changes or tests, and when the application was deployed. More specifically, it provides information about the number of defects you have per requirement, and how many tests are passing and failing. It can tell you if the quality of your current iteration is better or worse than the last build you created, as well as what tests need to be re-run for all requirement changes that occurred in a specified time frame.

You can implement and maintain reliable, repeatable, and enforceable processes anywhere, anytime and for any type of environment you may have. You can recover work lost in the event of an authorized user's mistake, such as the accidental deletion or overwriting of source code files, etc. In addition, secure access guarantees only those people who are authorized to view or change project assets can do so.

IBM Rational ClearQuest easily scales to support the size of your team and projects. It adapts to the way your company works, letting you customize management of projects, engineering change orders, defect tracking, and more. You get tools to effectively communicate, collaborate, define, adhere to, and enforce consistent processes for development, testing, and approval, across the software lifecycle-no matter what the environment (e.g. open systems, legacy systems, offshore, in-house, co-sourced, or outsourced). Tight integration with leading IDEs, including WebSphere Studio, Eclipse, and .NET, delivers instant access to change information from within your preferred development environment.

Since IBM Rational ClearQuest manages the full range of testing activities, including test planning, test execution, capture and analysis of test results, you can more accurately define, create, and associate test cases with specific test plans. Thus, ClearQuest removes many of the manual steps you worry about today and helps automate and manage them, so you and your team can spend your time on other tasks and thereby improve efficiency. More importantly, it bridges the quality gap between development and testing activities, shortening delivery time and reducing the cost and time required for developing and deploying high-quality software applications.

## Rounding Out the IBM Family of Quality Management Solutions

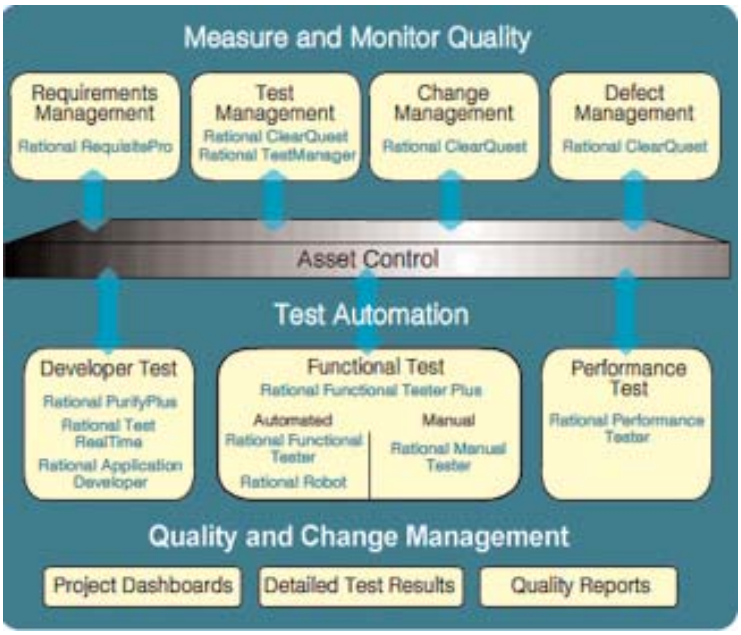
IBM Rational ClearQuest is also tightly integrated with complementary tools, including IBM Rational Performance Tester, IBM Rational Functional Tester, IBM Rational Build Forge, IBM Tivoli Provisioning Manager, and IBM Rational Manual Tester, for more efficient execution and management of tests and quality. Together, they enable delivery teams to manage quality-beginning with tracking requirements throughout the process of building and releasing software to managing testing processes and final product deployment. With the Rational portfolio, you are able to manage your requirements, changes, processes and quality from conception to operations, all within a single and integrated solution. Furthermore Rational test automation tools leverage popular industry technologies including Java and Microsoft VB.NET. Thus, your development teams can leverage skills they already have without having to learn new or proprietary languages when they begin using Rational tools.

A discussion on quality management solutions would be incomplete without discussing IBM Rational Performance Tester (RPT). The next release of RPT will ship with the IBM Performance Optimization Toolkit (IPOT). RPT allows you to validate Web application scalability and reliability, capture and correct performance problems, and avoid performance problems before deployment. You can measure an application's ability to support multiple concurrent users and detect performance deficiencies, scalability failures, and performance degradations in large, complex, or distributed applications, and retroactively discover and correct the source of the problem. IPOT is also available today as a standalone toolkit.

With this capability, RPT lets you pinpoint system bottlenecks before application deployment by simplifying the creation, execution, and results analysis of multi-user performance tests. It is a visual tool with a flexible, easy to use point-and-click interface that accurately simulates real world system loads. You can emulate any number of concurrent users and generate reports that identify performance bottlenecks and other problems to correct software defects before software is deployed. This works in the test lab or in production environments, resulting in faster resolution times; thus, creating less impact on your operational organization's service level agreements (SLAs).

Real time performance and throughput reports identify performance problems at any time during a test run. These reports provide multiple filtering and configuration options set before, during, and after a test run. RPT ensures accurate simulation of the system load of a real world user population. It also provides an automated data pooling capability that varies the test data set used by each simulated user.

The following Quality and Change Management figure illustrates how the products in the IBM Rational family interoperate to help you achieve your quality management goals.




*(Source: IBM Rational)*

Quality management is not an "all or nothing" initiative. You can implement it in stages or phases, but its greatest value is realized when organizational silos are relaxed and everyone on the project from design to development to operations can share information and artifacts (e.g. the RPT and the IPOT/ITCAM integrations for root cause analysis, etc.).

Quality management is ultimately about customer satisfaction-whether your customers are the operations team or your end users. The full benefits of your quality management efforts can be achieved by taking the steps of the quality management process throughout the design, development, and delivery stages of your project.

In a time when business agility, compliance, governance, and bolstering customer loyalty are so critical, quality management gives you a strategic development and career advantage. So take an active role in encouraging your company to acquire and implement IBM's quality management solutions. Anything you can do to improve your productivity, and help your company increase operational efficiency, lower total development costs, and decrease time to market is good for your career-and makes your job significantly easier.

***Rikki Kirzner*** *is a freelance writer and veteran computer industry professional with experience as an analyst and former Research Director for IDC, Gartner Group, and Meta Group and as a Senior Editor with Open Computing Magazine. Rikki covers software, development, open source, SOA, and mobile computing.*

# Key Principles for Business-driven Development

Level: Introductory

**Per Kroll**, Manager of Methods, IBM Rational, IBM
**Walker Royce**, Vice President, IBM Rational Worldwide Brand Services

First published by IBM at
www-128.ibm.com/developerworks/rational/library/oct05/kroll/index.html


15 Oct 2005

**IBM Resources**

***Rational Software Delivery Platform - desktop products v7***
***SWG TV***
***Webcast: Architecture Management***
***IBM Downloads***
***Webcast: Quality Management***
***Competitive Trade-up***

from The Rational Edge: As a major update of IBM Rational's six best practices for software development, this paper articulates a new set of principles that characterize a mature approach to the creation, deployment, and evolution of software-intensive systems.

Discuss this topic online! After you've read this article, click here to access the RUP forum where you can post questions and comments.

Since 1983, when the organization formerly known as Rational Software, now IBM Rational, was founded, our knowledge of the various processes and techniques for software development has increased through collaboration with a broad community of customers and partners. Over the years, many of them have joined our organization, and have continued to shape our understanding of software development practices as we learn what works, what doesn't, and why software development remains such a challenging endeavor.

As our customers, partners, and employees have heard us say many times, software development is a team sport. Ideally, the activity involves well-coordinated teams working within a variety of disciplines that span the software lifecycle. But it's not science, and it isn't exactly engineering, either -- at least not from the standpoint of quantifiable principles based on hard facts. Software development efforts that assume you can plan and create separate pieces and later assemble them, like you can in building bridges or spacecraft, constantly fail against deadlines, budgets, and user satisfaction.

So, in the absence of hard facts, the Rational organization has relied on software development techniques we call best practices, the value of which we have demonstrated repeatedly in our customer engagements. Rather than prescribing a plan-build-assemble sequence of activities for a software project, they describe an iterative, incremental process that steers development teams toward results.

Our six tried-and-true best practices have been the basis for the evolution of our tools and process offerings for more than a decade. Today, as we witness more companies pursuing software development as an essential business capability, we see these best practices maturing within the larger context of business-driven development. We think it's time to re-articulate our best practices for the broader lifecycle of continuously evolving systems, in which the primary evolving element is software.

This paper articulates a set of principles that we believe characterize the industry's best practices in the creation, deployment, and evolution of software-intensive systems:

- Adapt the process.
- Balance competing stakeholder priorities.
- Collaborate across teams.
- Demonstrate value iteratively.
- Elevate the level of abstraction.
- Focus continuously on quality.

We will explain each of these in order, describing the patterns of behavior that best embody each principle, as well as the most recognizable "anti-patterns" that can harm software development projects.

## Adapt the process

**Benefits:** Lifecycle efficiency, open/honest communication of risks

**Pattern:** Precision and formality evolve from light to heavy over the project lifecycle as uncertainties are resolved. Adapt the process to the size and distribution of the project team, to the complexity of the application, and to the need for compliance. Continuously improve your process.

**Anti-patterns:** Precise plans/estimates, track against static plan management style. More process is better. Always use the same degree of process throughout the lifecycle.

More process, such as usage of more artifacts, production of more detailed documentation, development and maintenance of more models that need to be synchronized, and more formal reviews, is not necessarily better. Rather, you need to right-size the process to project needs. As a project grows in size, becomes more distributed, uses more complex technology, has a larger number of stakeholders, and needs to adhere to more stringent compliance standards, the process needs to become more disciplined. But, for smaller projects with co-located teams and known technology, the process should be more lightweight. These dependencies are illustrated in Figure 1.

Second, a project should adapt process ceremony to lifecycle phase. In the beginning of the project, you typically have a lot of uncertainty, and you want to encourage a lot of creativity to develop an application that addresses the business needs. More process typically leads to less creativity, not more, so you should use less process in the beginning of a project where uncertainty is an everyday factor. On the other side, late in the project, you often want to introduce more control, such as change control boards, to remove undesired creativity and associated risk for late introduction of defects. This translates to more process late in the project.

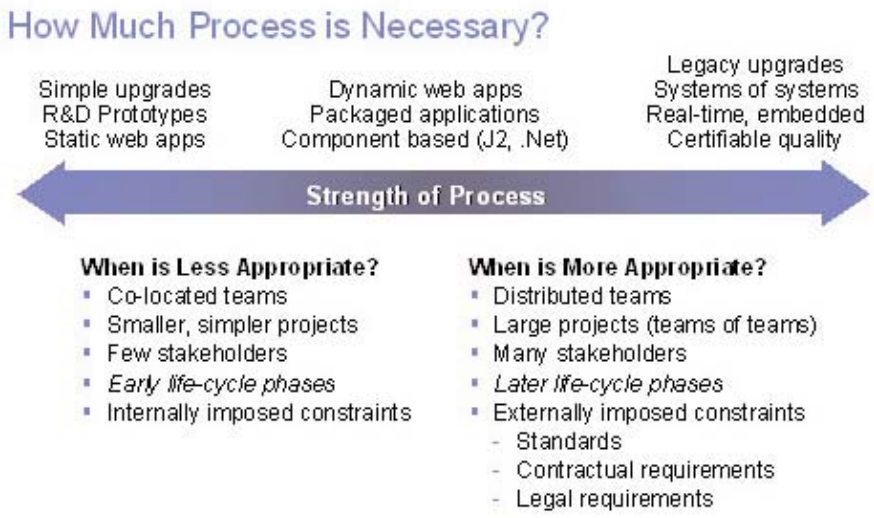
Third, an organization should strive to continuously improve the process. Do an assessment at the end of each iteration and each project to capture lessons learned, and leverage that knowledge to improve the process. Encourage all team members to continuously look for opportunities to improve.

Fourth, balance project plans and associated estimates with the uncertainty of a project. This means that early in projects when uncertainty typically is large, plans and associated estimates need to focus on big-picture planning and estimates, rather than aiming at providing 5-digit levels of precision when there are none. Early development activities should aim at driving out uncertainty to gradually enable increased precision in planning.

**Figure 1: Factors driving the amount of process discipline.** Many factors, including project size, team distributions, complexity of technology, number of stakeholders, compliance requirements, and where in the project lifecycle you are, steer how disciplined a process you need.

The anti-pattern to following this principle would be to always see more process and more detailed upfront planning as better. Force early estimates, and stick to those estimates.

**Figure 1**



## Balance competing stakeholder priorities

**Benefit:** Align applications with business and user needs, reduce custom development, and optimize business value

**Pattern:** Define and understand business processes and user needs; prioritize projects and requirements and couple needs with software capabilities; understand what assets you can leverage; and balance user needs and reuse of assets.

**Anti-pattern:** Achieve precise and thorough requirements before any project work begins. Requirements focus the drive toward a custom solution.

This principle articulates the importance of balancing often conflicting business and stakeholder needs. As an example, most stakeholders would like to have an application that does exactly what they want it to do, while minimizing the application's development cost and schedule time. Yet these priorities are often in conflict. If you leverage a packaged application, for example, you may be able to deliver a solution faster and at a lower price, but you may have to trade off many of your requirements. On the other hand, if a business elects to build an application from scratch, it may be able to address every requirement on its wish list, but the budget and project completion date can both be pushed beyond their feasible limits.

Rather than sending our programming teams out to attack each element in a requirements list, the first thing we need to do is to understand and prioritize business and stakeholder needs. This means capturing business processes and linking them to projects and software capabilities, which enables us to effectively prioritize the right projects and the right requirements, and to modify our prioritization as our understanding of the application and stakeholder needs evolve. It also means we need to involve the customer or customer representative in the project to ensure we understand what their needs are.

Second, we need to center development activities around stakeholder needs. For example, by leveraging use-case driven development and user-centered design, we can accept the fact that the stakeholder needs will evolve over the duration of the project, as the business is changing, and as we better understand which capabilities are the ones truly important to the business and end users. Our development process needs to accommodate these changes.

**Figure 2**


Third, we need to understand what assets are available, then balance asset reuse with stakeholder needs. Examples of assets include legacy applications, services, reusable components, and patterns. Reuse of assets can in many cases lead to reduced project cost; and for proven assets, their reuse often means higher quality in new applications. The drawback is that, in many cases, you need to trade off reuse of assets and perfectly addressing end-user needs. Reusing a component may lower development costs for a specific capability by 80 percent, but this may only address 75 percent of the requirements. So, effective reuse requires you to constantly balance the reuse of assets with evolving stakeholder needs.

**Figure 2: Balance the use of components with addressing requirements.** Using a component can radically reduce the cost and time to deliver a certain set of functionality. It may in many cases also require you to compromise on some functional or technical requirements, such as platform support, performance, or footprint (size of the application).

The anti-pattern to following this principle would be to thoroughly document precise requirements at the outset of the project, force stakeholder acceptance of requirements, and then negotiate any changes to the requirements, where each change may increase the cost or duration of the project. Since you lock down requirements up-front, you reduce the ability to leverage existing assets, which in turn drives you toward primarily doing custom development. Another anti-pattern would be to architect a system only to meet the needs of the most vocal stakeholders.

## Collaborate across teams

**Benefits:** Team productivity, better coupling between business needs, and the development and operations of software systems.

**Pattern:** Motivate people to perform at their best. Collaborate cross-functionally across analysts, developers, and testers. Manage evolving artifacts and tasks to enhance collaboration and progress/quality insight with integrated environments. Ensure that business, development, and operations teams work effectively as an integrated whole.

**Anti-pattern:** Nurture heroic individuals and arm them with power tools.

Software is produced by talented and motivated people collaborating closely. Many complex systems require the activities of a number of stakeholders with varying skills, and the largest projects often span geographical and temporal boundaries, further adding complexity to the development process. This is why people issues and collaboration -- what some have referred to as the "soft" element of software development -- have been a primary focus in the agile development community.1 This principle addresses many questions, including: How do you motivate people to perform at their best? And how do you collaborate within a co-located software team, within a distributed team, and across teams responsible for the business, software development, and IT operations?

The first step in effective collaboration is to motivate individuals on the team to perform at their best. The notion of self-managed teams2 has gained popularity in the agile community; it is based on making a team commit to what they should deliver and then providing them with the authority to decide on all the issues directly influencing the result. When people feel that they are truly responsible for the end result, they are much more motivated to do a good job. As the agile manifesto states: "Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done."

The second step is to encourage cross-functional collaboration. As we've mentioned for years, "software development is a team sport." An iterative approach increases the need for working closely as a team. We need to break down the walls that are often built up between analysts, developers, and testers, and broaden the responsibilities of these roles to ensure effective collaboration in an environment with fast churn. Each member needs to understand the mission and vision of the project.

As our teams grow, we need to provide effective collaborative environments. These environments facilitate and automate metrics collection and status reporting and automate build management and bookkeeping around configuration management. This efficiency allows fewer meetings, which frees team members to spend more time on more productive and creative activities. These environments should also enable more effective collaboration by simplifying communication, and bridging gaps in place and time between various team members. Examples of such an environment range from shared project rooms to networked or Web-based solutions, such as Wikis or integrated development environments and configuration and change management environments.

Our ultimate goal under this principle is integrated collaboration across business, software, and operation teams. As software becomes increasingly critical to how we run our business, we need close collaboration between 1) the teams deciding how to run our current and future business, 2) the teams developing the supporting software systems, and 3) the teams running our IT operations.

**Figure 3**

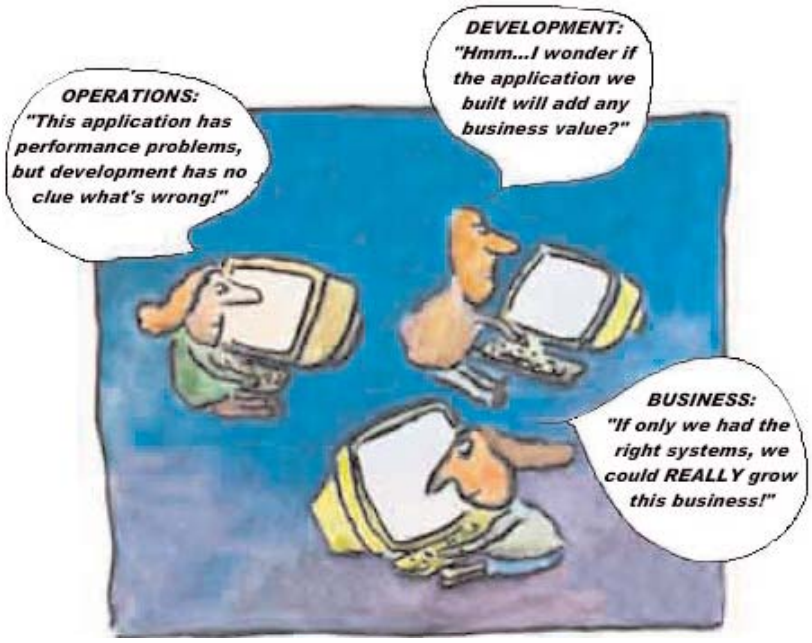



**Figure 3: Collaborate across business, development, and operations teams.** As software becomes more critical to how we run our business, we need to collaborate more closely around teams responsible for how to run the business, how to develop applications, and how to run the applications. In most companies, these three groups have poor communication.

The anti-pattern to following this principle would be to nurture heroic developers willing to work extremely long hours, including weekends. A related anti-pattern is to have highly specialized people equipped with powerful tools for doing their jobs, with limited collaboration between different team members, and limited integration between different tools. The assumption is that if just everybody does his or her job, the end result will be good.

## Demonstrate value iteratively

**Benefits:** Early risk reduction, higher predictability, trust among stakeholders

**Pattern:** Adaptive management using an iterative process. Attack major technical, business, and programmatic risks first. Enable feedback by delivering incremental user value in each iteration.

**Anti-pattern:** Plan the whole lifecycle in detail, track variances against plan. Detailed plans are better plans. Assess status by reviewing specifications.

There are several imperatives underlying this principle. The first one is that you want to deliver incremental value to enable early and continuous feedback. This is done by dividing our project into a set of iterations. In each iteration, you do some requirements, design, implementation, and testing of your application, thus producing a deliverable that is one step closer to the final solution. This allows you to demonstrate the application to end users and other stakeholders, or have them use the application directly, enabling them to provide rapid feedback on how you are doing. Are you moving in the right direction? Are stakeholders satisfied with what you have done so far? Do you need to change the features implemented so far, and what additional features need to be implemented to add business value? By being able to satisfactorily answer these questions, you are more likely to build trust among stakeholders by delivering a system that will address their needs. You are also less likely to over-engineer your approach, adding capabilities not useful to the end user3.

The second imperative is to leverage demonstrations and feedback to adapt your plans. Rather than relying on assessing specifications, such as requirements specifications, design models, or plans, you instead need to assess how well the code that's been developed thus far actually works. This means you focus on test results and demonstrate working code to various stakeholders to assess how well you are doing. This provides you with a good understanding of where you are, how fast the team can make progress within your development environment, and whether you need to make course corrections to successfully complete the project. You use this information to update the overall plan for the project and to develop detailed plans for the next iteration.

The third underlying imperative is to embrace and manage change. Today's applications are too complex to allow you to perfectly align the requirements, design, implementation, and test the first time through. Instead, the most effective application development methods embrace the inevitability of change. Through early and continuous feedback, we learn how to improve the application, and the iterative approach provides us with the opportunity to implement those changes incrementally. All this change needs to be managed by having the processes and tools in place so we can effectively manage change without hindering creativity.

The fourth imperative underlying this principle is the need to drive out key risks early in the lifecycle4, as illustrated in Figure 1. You must address the major technical, business, and programmatic risks as early as possible, rather than postponing risk resolution toward the end of the project. This is done by continuously assessing what risks you are facing, and addressing the top remaining risks in the next iteration. In successful projects, early iterations involve stakeholder buy-in on a vision and high-level requirements, including architectural design, implementation, and testing to mitigate technical risks. It is also important to retain information required to force decisions around what major reusable assets or commercial-off-the-shelf (COTS) software to use.

**Figure 4**

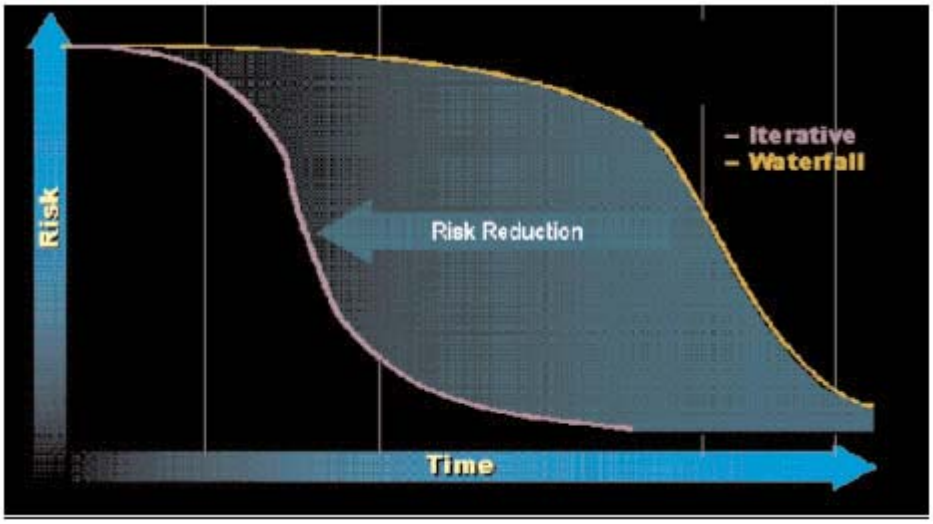



**Figure 4: Risk reduction profiles for waterfall and iterative development projects.** A major goal with iterative development is to reduce risk early on. This is done by analyzing, prioritizing, and attacking top risks in each iteration.

A typical anti-pattern (i.e., former software development practices that actually contribute to project failure) is to plan the whole lifecycle in detail upfront, and then track variances against plan. Another anti-pattern is to assess status in the first two thirds of the project by relying on reviews of specifications, rather than assessing status of test results and demonstrations of working software.

## Elevate the level of abstraction

**Benefits:** Productivity, reduced complexity

**Pattern:** Reuse existing assets, reduce the amount of human-generated stuff through higher-level tools and languages, and architect for resilience, quality, understandability, and complexity control.

**Anti-pattern:** Go directly from vague high-level requirements to custom-crafted code.

One of the main problems we face in software development is complexity. We know that reducing complexity has a major impact on productivity.5 Working at a higher level of abstraction reduces complexity and facilitates communication.

One effective approach to reducing complexity is reusing existing assets, such as reusable components, legacy systems, existing business processes, patterns, or open source software. Two great examples of reuse that have had a

major impact on the software industry over the last decade is the reuse of middleware, such as databases, Web servers and portals, and, more recently, open source software that provides many smaller and larger components that can be leveraged. Moving forward, Web services will likely have a major impact on reuse, since they provide simple ways of reusing major chunks of functionality across disparate platforms and with loose coupling between the consumer and provider of a service, as illustrated in Figure 6. This means that you can more easily leverage different combinations of services to address business needs. Reuse is also facilitated by open standards, such as RAS, UDDI, SOAP, WSDL, XML, and UML.

**Figure 5**

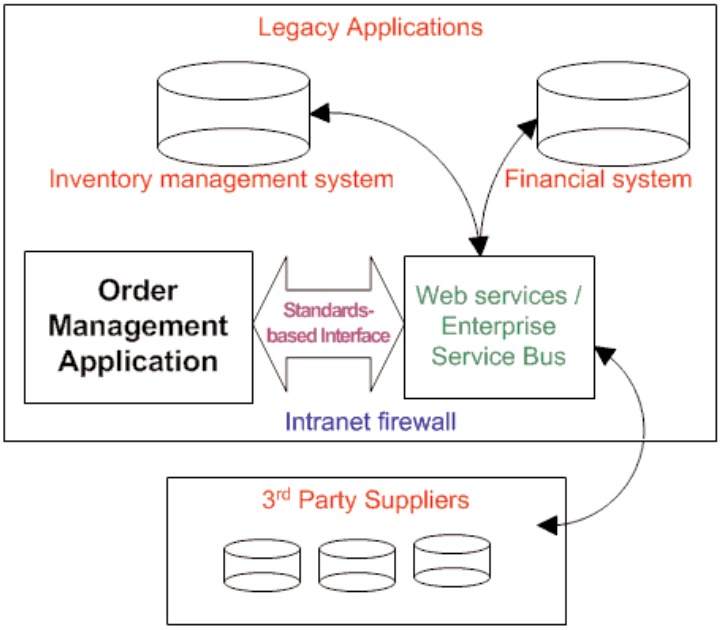



**Figure 5: Reuse existing assets through service-oriented architectures.** One of the problems with reuse is that two components need to know about each other's existence at development time. Service-oriented architectures alleviate that problem by providing what is called "loose coupling," shown here via the double-headed black arrows. A consumer of a service can dynamically find a provider of a service. You can hence wrap existing components or legacy systems, allowing other components or applications to dynamically access their capabilities through a standards-based interface, independent of platform and implementation technology.

Another approach to reducing complexity and improving communication is to leverage higher-level tools, frameworks, and languages. Standard languages, such as Unified Modeling Language (UML), and rapid application languages such as EGL6 provide the ability to express high-level constructs, such as business processes and service components, to facilitate collaboration around high-level constructs while hiding unnecessary details. Design and construction tools can automate moving from high-level constructs to working code by providing wizards to automate design, construction, and test tasks by generating code and enabling usage of code snippets, and by converting integration and testing into seamless development tasks through integrated development, build, and test environments. Another example is project and portfolio management tools, which allow you to manage financial and other aspects of multiple projects as one entity versus as a set of separate entities.

A third approach to managing complexity is to focus on architecture, no matter whether you are trying to define a business or develop a system or an application. In software development, we aim at getting the architecture designed, implemented, and tested early in the project. This means that early in the project we define the high-level building blocks and the most important components, their responsibilities, and their interfaces. We define and implement the architectural mechanisms, that is, ready-made solutions to common problems, such as how to deal with persistency or garbage collection. By getting the architecture right early on, we define a skeleton structure for our system, making it easier to manage complexity as we add more people, components, capabilities, and code to the project. We also understand what reusable assets we can leverage, and what aspects of the system need to be custom built.

The anti-pattern to following this principle would be to go directly from vague, high-level requirements to custom-crafted code. Since few abstractions are used, a lot of the discussions are made at the code level versus a more conceptual level, which misses many opportunities for reuse, among other things. Informally captured requirements and other information require many decisions and specifications to be revisited over and over, and limited emphasis on architecture causes major rework late in the project.

## Focus continuously on quality

**Benefits:** Higher quality and earlier progress/quality insight

**Pattern:** Team responsibility for end product. Testing becomes a high priority given continuous integration of demonstrable capabilities. Incrementally build test automation.

**Anti-pattern:** Postpone integration testing until all code has been completed and unit-tested. Peer-review all artifacts, rather than also driving partial implementation and testing, to discover issues.

Improving quality is not simply "meeting requirements," or producing a product that meets user needs and expectations. Rather, quality also includes identifying the measures and criteria to demonstrate its achievement, as well as the implementation of a process to ensure that the product created by the team has achieved the desired degree of quality, which can be repeated and managed.

Ensuring high quality requires more than the participation of the testing team; it requires that the entire team owns quality. It involves all team members and all parts of the lifecycle. Analysts are responsible for making sure that requirements are testable, and that we specify clear requirements for the tests to be performed. Developers need to design applications with testing in mind, and must be responsible for testing their code. Management needs to ensure that the right test plans are in place, and that the right resources are in place for building the testware and performing required tests. Testers are the quality experts. They guide the rest of the team in understanding software quality issues, and they are responsible for functional-, system-, and performance-level testing, among other things. When we experience a quality issue, every team member should be willing to chip in to address the issue.

One of the major benefits of iterative development is that it enables a test early and continuously approach, as illustrated in Figure 2. Since the most important capabilities are implemented early in the project, by the time you get to the end, the most essential software may have been up and running for months, and it is likely to have been tested for months. It is not a surprise that most projects adopting iterative development claim that an increase in quality is a primary tangible result of the improved process.

As we incrementally build our application, we should also incrementally build test automation to detect defects early, while minimizing up-front investments. As you design your system, consider how it should be tested. Making the right design decisions can greatly improve your ability to automate testing. You may also be able to generate test code directly from the design models. This saves time, provides incentives for early testing, and increases the quality of testing by minimizing the number of bugs in the test software. Automated testing has been a key area of focus for, among others, the agile community, where the aim is to automate testing of all code, and where tests are written before the code is written (so called test-first design.)

**Figure 6**

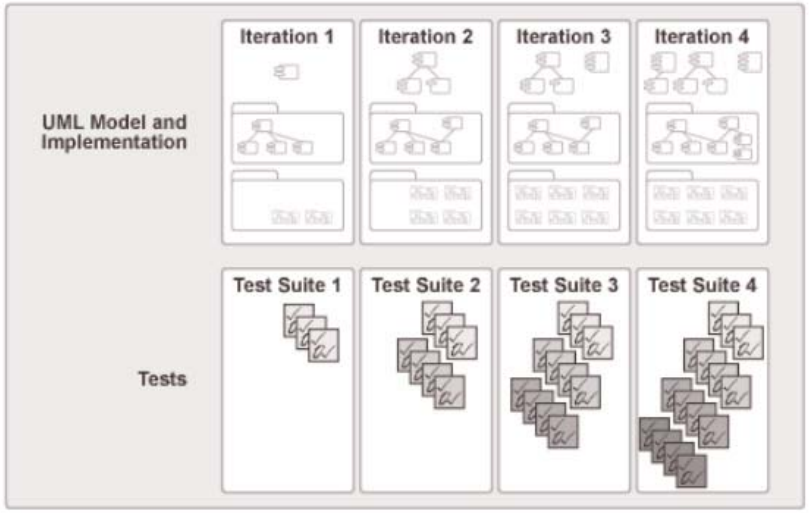



**Figure 6: Testing Is initiated early and expanded upon in each iteration.** Iterative development enables early testing. Software is built in every iteration and tested as it is built. Regression testing ensures that new defects are not introduced as new iterations add functionality.

The anti-pattern for this principle involves conducting in-depth peer-review of all intermediate artifacts, which is counter productive since it delays application testing, and hence, identification of major issues. Another anti-pattern is to complete all unit testing before doing integration testing, again delaying identification of major issues.

## Summary

As we noted at the beginning, these principles have evolved from their origins over a decade ago, when software developers generally worked in a somewhat more limited context. Our knowledge about software development, along with the overall conditions for project success, has matured and expanded. Today's business-driven software development organizations need guideposts that map a broader landscape, which includes geographically distributed development, IT governance and regulatory compliance needs, service oriented-architecture, and more.

Just as software development is not hard science, we believe that these principles should not be taken as statements of absolute, everlasting truth, but rather as guides to improved software project results. These principles will continue to evolve -- not at a breathtaking pace, but gradually, as we acquire more experience about what works, what doesn't, etc. The one thing that won't change is the long-standing Rational commitment to helping the success of customers whose businesses depend on developing and deploying software.

## Notes

[1] See **http://www.agilemanifesto.org/principles.html**
[2] See Ken Schwaber and Mike Beedle, *Agile Software Development with SCRUM*, Prentice Hall: 2001.
[3] According to the Standish Group's Chaos Report, 2002, 45 percent of features implemented on the average project are never used.
[4] As illustrated in Walker Royce's book, *Software Project Management: A Unified Framework*, Addison-Wesley, 1998
[5] See "Four Factors for Software Economics," in Royce, op. cit.
[6] Enterprise Generation Language, see **http://en.wikipedia.org/wiki/Enterprise_Generation_Language**

## About the authors

**Per Kroll** is the director of the Rational Unified Process development and product management teams at IBM Rational Software. He's been working with customers as a trainer, mentor, and consultant on the RUP and its predecessors since 1992 and was the original product manager for the RUP when the product team was launched in 1996. He's also been heavily involved in certifying partners and training Rational personnel to deliver services around the RUP.

**Walker Royce** is the vice president of IBM's Worldwide Rational Brand Services. He has managed large software engineering projects and consulted with a broad spectrum of software development organizations. He is the author of Software Project Management, A Unified Framework (Addison-Wesley Longman, 1998) and a principal contributor to the management philosophy inherent in the Rational Unified Process. Before joining Rational and IBM, he spent sixteen years in software project development, software technology development, and software management roles at TRW Electronics & Defense. He was a recipient of TRW's Chairman's Award for Innovation for distributed architecture middleware and iterative software processes and was a TRW Technical Fellow. He received his BA in physics from the University of California, and his MS in computer information and control engineering from the University of Michigan.

# Myths and Realities of Iterative Testing

Level: Introductory

**Laura Rose**, Quality Engineering Manager, IBM

First published by IBM at
www-128.ibm.com/developerworks/rational/library/apr06/rose/


15 Apr 2006

**IBM Resources**
***Rational Software Delivery Platform - desktop products v7***
***SWG TV***
***Webcast: Architecture Management***
***IBM Downloads***
***Webcast: Quality Management***
***Competitive Trade-up***

from The Rational Edge: Software testing expert Laura Rose questions and debunks some widely held myths pertaining to iterative development in general and iterative testing in particular. She explains how iterative development principles can address these common misunderstandings and lead to a pragmatic testing methodology.

Myths arise from a lack of direct experience. In the absence of information, we form beliefs based on what we think we know, often with a skeptical feeling towards what we don't know. In the realm of software development, myths can make it difficult to approach real-world problems objectively, thus putting budgets and schedules at increased risk.

In my career as a quality assurance manager, I have gained experience with a number of software development practices, including iterative development and the so-called "waterfall" approach. The former is generally presumed to be a more modern method than the latter. But this is, in fact, a myth: both approaches have been around since the 1960s. Another myth led to the popularity of the waterfall method in the 1970s. The thinking of Winston Royce, often cited as the father of the waterfall method, was actually misinterpreted. He recommended the single-pass waterfall method only for legacy maintenance projects. Royce actually suggested an iterative "do it twice" approach for software applications being developed for the first time.

From these inauspicious beginnings, the world of software development has given rise to numerous other myths. This article will question and debunk some of the most widely held myths pertaining to iterative development in general and iterative testing in particular. I'll explain how iterative development principles can address these common misunderstandings and lead us to a pragmatic testing methodology that mitigates or avoids altogether many common software development pitfalls, some of which our "mythology" holds as inevitable.

**Myth:** In a majority of software development projects, we quickly code a prototype application to reduce risk and prove a concept, with the intention of throwing this code away.

**Reality:** There is nothing wrong with this approach. But, because of schedule pressure or because the results are encouraging, we do not discard that code. The reality is that our prototyping is actually early coding. That code becomes the foundation and framework for our new application. However, because it was created under the presumption that it would be thrown away, it bypassed requirements reviews, design reviews, code reviews, and unit testing. Our next-generation application is now built on an indeterminate foundation.

In an iterative development lifecycle, continuous experimentation is viewed as a good thing, and early prototyping is encouraged in each development iteration. But any code that is submitted into the product needs to follow best practices to assure its stability, reliability, and maintainability. One way to encourage proper software development practices from the very start of a project is to use the initial coding effort to plan, prove, and present the processes you intend to use throughout the various iterative development phases. As part of the iterative testing method, you can use your early coding cycles to test your product's concept and flush out glitches in your development processes at the same time.

**Myth:** Initiating testing activities earlier in the development cycle increases delivery time while reducing the number of features in the product.

**Reality:** Testing is not the time-consuming activity in a development lifecycle. Diagnosing and fixing defects are the time-consuming, bottleneck activities.

Testing is not the obstacle that shipwrecks our projects -- testing is the lighthouse that keeps our projects off the rocks. Defects are in the product whether we look for them or not. Iterative testing moves the detection of problems closer to where and when they were created. This minimizes the cost of correcting the bug as well as its impact on schedules.

**Myth:** You can't test if you don't have a product to test.

**Reality:** Iterative testing isn't limited to testing code.

Every artifact your team produces can be verified against your success criteria for acceptance. Likewise, each process or procedure you use to produce your deliverables can be validated against your success criteria for quality. This includes the product concepts, the architecture, the development framework, the design, the customer usage flows, the requirements, the test plans, the deployment structure, the support and service techniques, the diagnostic and troubleshooting methods, and even the procedures you follow to produce the product.

As you can see from this partial list, the majority of these items don't involve code. Therefore, you miss opportunities to support quality and reduce risk by waiting until you actually have deliverable code. I disagree with the widely accepted notion that "you can't test quality into a product." You can test quality into a product. You just need to start early enough.

**Myth:** You are more efficient if you have one developer (or a single source) specialized in each development area. In the simplest argument, if you have thirty developers and employ pair programming, you can code fifteen features. If you assign one developer per feature, you can code thirty features in the same amount of time. With thirty features, you have a fuller and more complete product.

**Reality:** The risk associated with having one developer per feature or component is that no one else can maintain and understand that feature. This strategy creates bottlenecks and delays. Defects, enhancement requests, or modifications are now queued to that single resource. To stay on schedule, your developers must work weekends and extra hours for extended periods, because they are the only ones who can continue new feature development and fix defects in this area. Your entire project schedule is now dependent on heroic efforts by a number of "single resources."1 When that resource leaves the team, goes on vacation, or becomes overwhelmed, it causes delays in your schedule. Because of the way you've chosen to implement and manage your development strategy, you are now unable to provide your team with relief.

Pair programming, pair testing, code reviews, and design reviews are sound practices that not only increase the quality of the product, but also educate others on each feature or component, such that they increase your pool of resources to fix and maintain project code. The two members of a pair don't have to be equally sophisticated or knowledgeable in their area. They can be just knowledgeable enough to eliminate the inevitable bottlenecks created by specialization as discussed above.

Moreover, dividing development activities into logical, smaller, independent chunks (a.k.a. sprints) allows developers of different skill levels to work efficiently on the various pieces. With multiple capable resources, you can avoid actually assigning the different tasks to a specific developer. When we have more than one resource capable of accomplishing the task, assigning specific tasks to specific individuals creates a false dependency. Similar to multiple bank tellers servicing a single line of waiting customers, efficiency improves when developers instead sign up for the next

task when they've completed the last task.

**Myth:** Producing code is the developer's primary task.

**Reality:** The primary task of everyone on the development team is to produce a product that customers will value. This means that during the requirements review activities, for example, the developer's primary task is "requirement review." During the design activities, the developer's primary task is creating and reviewing the design documents. During the coding activities, the developer's primary task is generating bug-free and customer-relevant code. During the documentation review activities, the developer's primary task is making sure the user assistance materials and error messages serve to flatten the customer's learning curve. During installation and setup, the developer's primary task is to make sure customers can set up and configure your product easily, so that they can get their "real" job done as efficiently as possible. The greater the effort required to use software to accomplish a task, the lower the customer's return on investment and the greater the abandonment rate for the application.

**Myth:** Requirements need to be stable and well defined early for software development and testing to take place efficiently.

**Reality:** If our ultimate goal was "efficient software development and testing," then having stable and well-defined requirements up-front might be a must. But our actual goal is to produce a product that customers will value. Most of us can readily admit that we don't always know what we want. Since there are constant changes in the marketplace, such as new products and options, we often change our minds after being introduced to new concepts and information. Acknowledging the above illustrates one key reason why it is rarely effective to keep requirements static. Frequent interaction with the product during development continually exposes the customer to our efforts and allows us to modify our course to better meet customers' changing needs and values.

**Myth:** When the coding takes longer than originally scheduled, reducing test time can help get the project back on schedule.

**Reality:** Typically, coding delays occur because of unexpected difficulties or because the original design just didn't work. When it's apparent that we've underestimated project complexity, slashing test time is a very poor decision. Instead, we need to acknowledge that, since the test effort was probably based on the underestimated coding effort, the test effort was probably underestimated as well. We may therefore need to schedule more test time to correct the original estimation, not less.

Iterative testing increases test time without affecting the overall schedule by starting the testing earlier in each iteration. Also, the quality of the product determines the amount of testing that is required -- not the clock. For instance, if the product is solid and no new defects are being found in various areas, then the testing will go very smoothly and you can reduce testing time without reducing the number of tests or the test coverage. If the product is unstable, and many defects are being discovered and investigated, you'll need to add test cycles until the quality criteria are met. And don't forget that testing is not the project's time-consuming activity in the first place.

**Myth:** Finding and fixing all the defects will create a quality product.

**Reality:** Recent studies illustrate that only 10 percent of software development activities, such as creating customer-requested features, actually add value for the customer. A percentage of features, for example, are developed in order to stay competitive in the market, but are not necessarily used or desired by the customers. Finding and fixing defects related to these features also does not add customer value, because the customers might never have encountered these bugs.

Iterative testing, on the other hand, actually reduces defect inventory and customer wait time based upon what is of value to the customer. By involving customers in each iteration, iterative testing compresses the delivery cycle to the

design partner customers, while maximizing the value of the application to this customer.

**Myth:** Continually regression-testing everything every time we change code is tedious and time consuming…but, in an ideal world, it should be done.

**Reality:** Regression testing doesn't mean "testing everything, every time."
Iterative regression testing means testing what makes sense in each phase and iteration. It also means modifying our coverage based on the impact of the change, the history of the product, and the previous test results.

If your regression tests are automated, then go ahead and run all of them all the time. If not, then be selective about what tests you run based on what you want the testing to accomplish. For instance, you might run a "sanity regression suite" or "set of acceptance tests" prior to "accepting" the product to the next phase of testing. Since the focus of each iteration is not necessarily the same, the tests don't need to be the same each time. Focus on features and tests that make sense for the upcoming deliverables and phase. For instance, if you're adopting components from a third party, like a contractor or an open source product, the sanity regression suites would focus on the integration points between the external and internal components. If, during your initial sanity regression testing of this third party module, you find defects or regressions, you may choose to alter your regression suites by adding additional tests based on the early results.

If, on the other hand, you're adopting a set of defect fixes that span the entire product that is within your control, the sanity regression suite would be focused and structured entirely on your end-to-end, high profile customer use cases. If the change is confined to just one area and the product has a stable quality track record, you could focus the regression suite on just that area. Likewise, in the end-game, you may want a very small sanity regression suite that covers media install, but not in-depth or end-to-end tests. Once again, the focus of the sanity or acceptance regression suite depends upon what was tested in the previous cycle, the general stability of the product, and the focus of the next iteration.

**Myth:** It's not a bug -- the feature is working as designed.

**Reality:** The over-explanation of why the product is doing what it's doing is a common trap. Sometimes we just know too much. When defects are triaged and reviewed, we often explain away the reasons for the defect. Sometimes we tag defects as "works as designed" or "no plans to fix" because the application is actually working as it was designed, and it would be too costly or risky to make the design change. Similarly, we explain many usability concerns as "it's an external component" or "it's a bell and whistle." Our widgets or UI controls may have known limitations. Or we may even tell ourselves that "once the user learns to do it this way, they'll be fine."

In short: We know the ins and outs of the code, and why it's working as it is. And at that component level, we're making very logical and sensible decisions. But we're lacking the top-down view of the entire customer experience. We aren't appreciating the end effect of our coding trade-offs and workarounds. Our primary focus is getting the feature code completed on time. By focusing on just the individual feature or component, we are unintentionally making code decisions that negatively impact the overall flow and usability of the product. It's usability, after all, that most affects the customer's efficiency. It's also usability (or lack thereof) that drives user abandonment numbers.

Elevating our view of the project above those layers of detail allows us to see the product as the customer would -- in its entirety, rather than as individual components. This elevated view helps us to ignore all the reasons why the product works the way it does. Customers don't really care if a given design was quicker to code. It's of little concern to them if the UI controls are conflicting with the API of component X, which is outside your particular development task. They are only concerned about whether the application is assisting or obstructing them in the pursuit of their goals.

Table 1 illustrates some ways we can act as advocates for the customer's perspective as we test an application.

**Table 1: Ways to advocate for the customer's perspective**

| When we encounter this response... | we should... |
|---|---|
| We already know this isn't right. | Identify explicit things that need to be fixed by the release date and what needs to be fixed in a subsequent release. Identify owners and deadlines. |
| It's on our list for future consideration (with no scheduled date). | Realize that if it has a "future consideration" tag (or similar), it's not real. Work with tech support, field consultants, and customers to illustrate the importance of the bug or feature. Once the team is assured of the customer value, schedule a release date and owner. |
| It's in someone else's code outside our department. | Create a SWAT team that includes "outside department" staff in addition to development staff. Schedule a release date and owner of the change. |
| Customer or pilot error. | Study the documentation or usage flow. Customer errors are often the result of some unclear and unintuitive step. Explicitly identify the change that needs to occur, schedule it, and set an owner. |
| It's working as designed. | Request a full review of the design in the workflow perspective. Walk through how the customer or role will play out from beginning to end. |
| It's working this way because XXX. | Realize that if XXX isn't "because the customer wants it this way," then XXX is irrelevant. Eliminate that reason from the discussion and move on to the next point. |

**Myth:** A tester's only task is to find bugs.

**Reality:** This view of the tester's role is very limited and adds no value for the customer. Testers are experts with the system, application, or product under test. Unlike the developers, who are responsible for a specific function or component, the tester understands how the system works as a whole to accomplish customer goals. Testers understand the value added by the product, the impact of the environment on the product's efficiency, and the best ways to get the most out of the product.

Taking advantage of this product knowledge expands the value and role of our testers. Expanding the role of the tester to include customer-valued collateral (like tips, techniques, guidelines, and best practices for use) ultimately reduces the customer's cost of ownership and increases the tester's value to the business.

**Myth:** We don't have enough resources or time to fully test the product.

**Reality:** You don't need to fully test the product -- you need to test the product sufficiently to reduce the risk that a customer will be negatively affected.

The reality of changing market demands generally means that, indeed, it's actually not possible to exhaustively test a product in the specified timeframe. This is why we need a pragmatic approach to testing. Focus on your customers' business processes to identify your testing priorities. Incorporate internal customers to system test your product. These steps increase your testing resources, while providing real-world usability feedback. You can also do your

system testing at an external customer lab to boost your real-world environment experience without increasing your maintenance or system administration activities.

**Myth:** Testing should take place in a controlled environment.

**Reality:** The more the test environment resembles the final production environment, the more reliable the testing. If the customer's environment is very controlled, then you can do all your testing in a controlled environment. But if the final production environment is not controlled, then conducting 100 percent of your testing in a controlled environment will cause you to miss some important test cases.

While unpredictable events and heterogeneous environments are difficult to emulate, they are extremely common and therefore expected. In our current global market, it is very likely that your application will be used in flexible, distributed, and diverse situations. In iterative testing, we therefore schedule both business usage model reviews and system testing activities with customers whose environments differ. The early business usage reviews identify the diversity of the target customer market, prior to coding. System testing at customer sites exercises our product in the real world. Although these "pre-released" versions of the product are still in the hands of our developers and running on our workstations, they are tested against customer real-world office (or lab) environments and applications. While this strategy doesn't cover every contingency, it acknowledges the existence of the unexpected.

**Myth:** All customers are equally important.

**Reality:** Some customers are more equal than others, depending upon the goal of a particular release. For example, if the release-defining feature for the January release is the feature that converts legacy MyWidget data to MyPalmPilot data, then the reactions of my customers that use MyWidget and MyPalmPilot are more important for this particular release than the input of other customers.

All our customers are important, of course. But the goal of iterative testing is to focus on testing the most important features for this particular iteration. If we're delivering feature XYZ in this iteration, we want expert customer evaluation of XYZ from users familiar with prior XYZ functionality. While we welcome other feedback, such as the impressions of new users, the XYZ feature takes precedence. At this stage of development, users new to the market cannot help us design the "right XYZ feature."

**Myth:** If we're finding a lot of bugs, we are doing important testing.

**Reality:** The only thing that finding a lot of bugs tells us is that the product has a lot of bugs. It doesn't tell us about the quality of the test coverage, the severity of the bugs, or the frequency with which customers will actually hit them. It also doesn't tell us how many bugs are left.

The only certain way to stop finding bugs is to stop testing. It seems ridiculous, but the thought has merit. The crux of this dilemma is to figure out what features in the product actually need to work. I've already mentioned that there are many workflows in a product that aren't actually used -- and if they aren't used, they don't need to work. Incorporating customer usage knowledge directly into your test planning and defect triage mechanism improves your ability to predict the customer impact and risk probability associated with a defect. Incorporating both risk- and customer-based analysis into your test plan solution will yield a more practical and pragmatic test plan. Once you're confident in your test plan, you can stop testing after you've executed the plan.

How do you build that kind of confidence? Start, in your test planning, by identifying all the areas that you need to test. Get customer review and evaluation on business processes and use cases so that you understand the frequency and importance of each proposed test case. Take special care to review for test holes. Continually update and review your test plans and test cases for each iteration. Your goal is to find what's not covered. One way to do this is to map bug counts by software areas and categories of tests. If a software area doesn't have defects logged

against it, it could mean that this area is extremely solid or that it hasn't been tested. Look at the timestamps of the defect files. If the last defect was reported last year, maybe it hasn't been tested in awhile. Finding patterns of missing bugs is an important technique to verifying test coverage.

**Myth:** Thorough testing means testing 100 percent of the requirements.

**Reality:** Testing the requirements is important, but not sufficient. You also need to test for what's missing. What important requirements aren't listed?

Finding what's not there is an interesting challenge. How does one see "nothing?" Iterative testing gets customers involved early. Customers understand how their businesses work and how they do their jobs. They can tell you what's missing in your application and what obstacles it presents that stops them from completing their tasks.

**Myth:** It's an intermittent bug.

**Reality:** There are no intermittent bugs. The problem is consistent -- you just haven't figured out the right conditions to reproduce it. Providing serviceability tools that continually monitor performance, auto-calibrate at the application's degradation thresholds, and automatically send the proper data at the time of the degradation (prior to the application actually crashing) reduces both in-house troubleshooting time and customer downtime. Both iterative testing and iterative serviceability activities reduce the business impact of undiscovered bugs.

Better diagnostic and serviceability routines increase the customer value of your product. By proactively monitoring the environment when your product starts to degrade, you can reduce analysis time and even avoid shutdown by initiating various auto-correcting calibration and workaround routines. These types of autonomic service routines increase your product's reliability, endurance, and run-time duration, even if the conditions for reproduction of a bug are unknown.

In a sense, autonomic recovery routines provide a level of continuous technical support. Environment logs and transaction trace information are automatically collected and sent back to development for further defect causal analysis, while at the same time providing important data on how your product is actually being used.

If we acknowledge that bugs are inevitable, we also need to realize the importance of appropriate serviceability routines. These self-diagnostic and self-monitoring functions are effective in increasing customer value and satisfaction because they reduce the risk that the customer is negatively affected by bugs. Yet even though these routines increase customer value, few development cycles are devoted to putting these processes in place.

**Myth:** Products should be tested under stress for performance, scalability, and endurance.

**Reality:** The above is true. But so is its opposite. Leaving an application dormant, idle, or in suspend mode for a long period emulates customers going to lunch or leaving the application suspended over the weekend, and often uncovers some issues.

I recommend including sleep, pause, suspension, interrupt, and hibernating mode recovery in your functional testing methods. Emulate a geographically-distributed work environment in which shared artifacts and databases are changing (such as when colleagues at a remote site work during others' off-hours) while your application is in suspend or pause mode. Test what occurs when the user "wakes it up," and the environment is different from when they suspended it. Better yet, put your product in a real customer environment and perform system testing that emulates some of the above scenarios.

**Myth:** The customer is always right.

**Reality:** Maybe it's not the right customer. You can't make everyone happy with one release. Therefore, be selective in your release-defining feature set. Target one type of customer with a specific, high-profile testing scenario. Then, for the next release or iteration, select a different demographic. You'll test more effectively; and, as your product matures, you'll increase your customer base by adding satisfied customers in phases.

**Myth:** Automate! Automate! Automate!

**Reality:** Automate judicially and with ROI in mind. The more the test environment resembles the final production environment, the more reliable the testing. If the customer's product is 100 percent automated, then Automate! Automate! Automate! If your product isn't meant to be automated in the customer's environment, then you want a combination of automation, ad hoc, exploratory, and customer scenario testing.

It's also helpful to expand your definition of automation. Automated test cases can retest areas that you have already tested manually. But that doesn't increase your test coverage or your understanding of how users interact with the application. Instead, create automation that actually allows you to invest more effort in creative manual testing. Automate the things you need to do frequently and that take you a long time to accomplish, like:

- The breakdown and setup of clean environments for testing each build.
- Sanity acceptance testing for every build, integration point, or iteration.
- Testing the same suite across various platforms, operating systems, and languages.
- Unit and command line testing of low-level components.

For more ideas on increasing your return on investment in automation, visit http://www-128.ibm.com/developerworks/rational/library/may05/rose/.

**Myth:** Iterative development doesn't work.

**Reality:** It's human nature to be skeptical of the unknown and the untried. In the iterative development experience, benefits accrue gradually with each successive phase. Therefore, the full benefit of the iterative approach is appreciated only towards the end of the development cycle. For first-timers, that delayed gratification can truly be an act of faith. We have no experience that the approach will work, and so we don't quite trust that it will. When we perceive that time is running out, we lose faith and abandon the iterative process. In panic mode, we fall back into our old habits. More often than not, iterative development didn't actually fail. We just didn't give it a chance to succeed.

Iterative testing provides visible signs at each iteration of the cumulative benefits of iterative development. When teams share incremental success criteria (e.g., entrance and exit criteria for each iteration), it's easier to stay on track.

Because we are continually monitoring our results against our exit criteria, we can more easily adjust our testing during the iterations to help us meet our end targets. For instance, in mid-iteration, we might observe that our critical and high defect counts are on the rise and that we're finding bugs faster than we can fix them. Because we've recognized this trend early, we can redistribute our resources to sharpen our focus on critical-path activities. We might reassign developers working on "nice-to-have" features to fix defects in "release-defining" features or remove nice-to-have features associated with high defect counts.

## Conclusion

I've touched on just a few of the software development myths and assumptions that we encounter every day. The more assumptions we make, the less we're open to discovering the unexpected -- which is what software testing is all about.

Unfortunately, our myths are very seductive. They are disguised as answers, and they conveniently end the dialogue. When we're understaffed and under pressure, it's very tempting to accept assumptions as facts.

Because it's often difficult for us to distinguish myth from reality, we need to test our answers. This simple and effective mantra, which sums up everything I've talked about in this article, will help keep you on that path:
Iterative testing is never being satisfied with the right answer.

## Notes

[1] In the Capability Maturity Model (CMM, or CMMI), the maturity level described by depending upon heroic efforts by several individuals is symptomatic of the Level 1, better known as the Chaos Level.

## About the author

**Laura Rose** is the quality engineering manager responsible for automated performance test tools at IBM Rational. In addition to leading projects in both software programming and testing environments, she has thirteen years of programming experience and ten in test management. She has been a member of the American Society for Quality, the Triangle Quality Council, and the Triangle Information Systems Quality Association, and has published and presented at various test and quality conferences. You can reach her at llrose@us.ibm.com.

# Roadtrip! A Vacationer's Guide to Iterative Development

Level: Introductory

**Laura Rose**, Quality Engineering Manager, IBM

First published by IBM at
www.ibm.com/developerworks/rational/library/jul06/rose/index.html


14 Jul 2006

**IBM Resources**
***Rational Software Delivery Platform - desktop products v7***
***SWG TV***
***Webcast: Architecture Management***
***IBM Downloads***
***Webcast: Quality Management***
***Competitive Trade-up***

from The Rational Edge: If you or someone you know is skeptical about the viability and value of iterative and incremental software development methods, perhaps what's needed is a familiar framework for understanding the basic concepts. Here, Laura Rose steps outside the realm of software to describe one form of iterative project: a cross-country roadtrip.

The iterative and incremental development process is built on the work of Barry Boehm's spiral model,1 and it has been enhanced by many individuals and organizations, including the IBM Rational team, over the past two decades. The process is characterized by continuous discovery, invention, and implementation. With each iteration, the development team drives the project's artifacts to closure in a predictable and repeatable way.2 The iterative approach accounts for changing requirements, it mitigates risks earlier, and it allows for tactical changes as developers learn along the way.

Even though iterative and incremental development methods have proved highly successful for both large and small development organizations, many still believe that iterative development does not work. While it is human nature to be skeptical of the unknown and untried, many, if not most, of us have implemented something very similar to the iterative development lifecycle in our life outside the software development arena. That is, we've taken a lengthy car trip across country -- many of us more than once.

This paper illustrates how iterative development techniques are used in everyday activities such as taking a family vacation.

Few would disagree that life involves continuous discovery, invention, and implementation. No one has their careers, goals, and accomplishments all plotted and well-defined at birth. We readily acknowledge that we learn as we grow, changing our strategies and even our goals with every experience and observation. Yet, when it comes to software development, we have difficulty applying those concepts. We tend to believe that a carefully laid-out plan and tight schedule is the only way to develop a new piece of software. In fact, iterative and incremental techniques offer a better approach to creating software; small steps in the process are accomplished, demonstrated, and evaluated, and discoveries often suggest new ways to go about achieving the next steps.

To illustrate this iterative process, let's compare it to something many of us are familiar with: managing a family vacation. Even if you've never taken a family vacation (or it's been a long time), I think you'll see the value of this cross-country trip metaphor.

## The project

Let's pretend that our sister Jane is getting married on June 3 in Orlando, Florida, USA, and we live in San Diego. We have also agreed to pick up other family members along the way.

Already, as in an iterative development project, we have our mandatory requirements (get to Orlando!), release-defining features (pick up other relatives), and a fixed release date (our sister is getting married on June 3, whether we're there or not). Note that, in real life, we have no trouble accepting hard deadlines -- like wedding dates. How often have you heard someone complain that a selected wedding date is too aggressive? We simply accept the invitation, or we send our regrets.

Just as at the start of a software development project, we have multiple methods to accomplish our goals. We can fly, take a train, rent a car, or combine any of these options. We can take three days or three weeks to get there. And, as in a software project, we have other people (stakeholders) involved. Imagine that our stakeholders are Aunt Marian in Santa Fe, New Mexico; Uncle Cid in Baton Rouge, Louisiana; and cousin Tanya in Jackson, Mississippi.

## Project features

At the start, everyone agrees that it would be fun to make this a three-week family vacation, with our primary stops at the three cities where our relatives live. To increase value and enjoyment, we plan some additional excursions en route. These additional excursions will be optional, depending on our progress, but they surely will be "nice to do."

So far, this trip partly resembles the Inception phase of an iterative development project: It's common to have a few release-defining (primary) features and several secondary or nice-to-do features. The release-defining feature set includes the mandatory and highly publicized components. As this title suggests, these features determine the critical path. By definition, our release date will be postponed if those selected features are not of the agreed quality or completeness. The development team continues to work on the secondary features within the same schedule. But we do not intend for the nice-to-do features to affect the release date. Because the secondary features are not publicized, we have the flexibility to reposition our resources away from those secondary features in order to assure that the primary feature set is completed on time.

The side-excursions (secondary vacation stops) are initially selected based upon the routes to the various family member pickup points. If time gets tight, these nice-to-see landmarks can be omitted to get us back on schedule. So, even though we don't know all the stops along the way, we do have a skeleton of the route (project plan), when we need to be at our various pickup points (our requirements, milestone dates, and iteration phases) and the wedding date (project end date).

We secure a friend's RV as our travel method. We think that this would be relaxing and comfortable. The RV adds flexibility, because we are not required to stay at hotels every night. But the RV does have limited space. So for this to be successful, we need everyone to conform to certain traveling criteria, such as luggage limits, a commitment to the team's expenses, what they need to supply for the trip, and our various timetables. If at any point along the journey, we don't meet our agreed-upon travel criteria, we need to stop, evaluate our situation, and adjust.

## Legs of the journey

In the iterative development method, there are initial entry and exit criteria that govern whether each leg of our development journey can begin. These targets are used to guide and validate the stability, reliability, and accuracy of the product under development. These agreed-upon criteria are created at the beginning of the project, with the understanding that they can be appropriately altered along the way.3 An example of an iteration exit or entrance criteria would be:

1. All planned features delivered for this iteration
2. All stakeholders reviewed and approved planned test coverage for this iteration
3. All planned tests executed, with 95 percent passing
4. No known blocking or high priority issues

We continually monitor the project according to these criteria along the way. If at any point along the development lifecycle, we recognize that we're off course, we stop, evaluate our situation, and adjust. It's also important to realize that "adjust" can mean modifying the entrance/exit criteria.
Because we will be at different points of our vacation at each iteration milestone (see Figure 1), the entrance and exit criteria for each leg of the journey will be different for each participant. As an example, you and I will be traveling for three weeks. Therefore, our luggage allocation would be larger than someone who is picked up on the last few days of the vacation.

At the start, we identify a general plan and important criteria for the trip. After a few modifications to the plan, everyone agrees to the terms, conditions, and timetable.

We're on our way.

**Figure 1**


**Figure 1: Three milestones before the end of our iterative trip: Santa Fe, Baton Rouge, and Jackson.**

Although we know we need to make three stops on this trip, we're not too concerned with our third stop in Jackson just yet. As we begin the journey, we focus on getting to Aunt Marian in Santa Fe. We refine a more detailed route for that leg of the journey (the first iteration). Since we have friends in Phoenix, we pop in there for a brief visit. We aren't exactly sure what they have in store for us, but we know that they plan to take us sightseeing in their hometown. We had already called ahead with our timetable and constraints. Our friends in Phoenix schedule their activities to coincide with our requirements. We also make sure to periodically touch base with them on route.

Note that leaving the planning of the time in Phoenix to our friends is similar to contracting or outsourcing a feature or component in a development project. We establish certain timetables, constraints, checkpoints, and communication procedures; but we leave the details to the third party.

We have a great time in Phoenix and we successfully get to Santa Fe on time.

## The first hiccup

When we arrive in Santa Fe, Marian is ready and we review our status. Marian meets the agreed-upon luggage criteria, she has budgeted the proper spending cash, and she's packed the items that were on her list of things to bring to the wedding. The only hiccup is that she is insisting on bringing her puppy, Dogma. We didn't originally plan for this, so we need to evaluate what type of adjustment is needed. Although you or I don't have a specific problem with that, we need to check with all the stakeholders (similar to conducting a change management review). We regrettably find out that Cousin Sid (the object of the next leg of our journey) is allergic to dogs. Dogma can't travel with us.

In many styles of software development, adding a feature after the project plan has been approved is termed as "feature creep." But in the iterative development lifecycle, we expect changes in requirements and requests. We intend to constantly learn from the previous iteration. Since progressive requirements refinement is one of the essential concepts in iterative development, conducting requirement reviews at the start or end of each iteration is critical.

In our case, Marian is disappointed, but she understands. Because she previously did a risk-assessment and realized that Dogma may not be allowed to come with us, her contingency plan was already in place. Her friend (several towns away) had already agreed to take the dog. Dropping off the dog will take us several hours off the planned route, so we review our day's schedule with this adjustment in mind. We decide that we can easily include the route change without any negative consequences. Marian's dog-sitting friend also offered to give us an expert sightseeing tour of that area. The consequence of the change was trivial and even beneficial: i.e., because we adjusted to this change, we had a more exciting interactive tour (than the landmark previously planned).

In iterative development, this is much like substituting one optional feature for another as we continually learn about our project and our customer needs. Even though the gathered knowledge from the previous iteration might change the contents or course of the next iteration, our primary goals and end date stays stable.

We successfully deliver Dogma. We have a great time meeting and visiting with Darla. We also learned that Darla's son and daughter will be attending the same high school as our son Josh. We made some future plans to travel together for different school events and even investigate the possibility that the children might room together. To sum up, we are able to make some tentative plans about the future, all because the flexibility built into the iterative process allows us some deviation from the original plan.

## An actual setback

Our next major iteration milestone is Cousin Sid in Baton Rouge. On the way, we want to stop at three cities in Texas: Austin, San Antonio, and Houston. We spend several enjoyable days at Austin and San Antonio. Everything is going to plan until the RV breaks down outside of San Antonio. It's unknown exactly what it will cost or how long it will take to fix the RV. The optimistic estimate is a few days, but the mechanic can't tell us for sure until he spends more time with the RV (much like investigating a defect in our code). Unfortunately, we didn't foresee something like this occurring. It has the potential for setting us back several days.

In any software development project, the team tries to identify risks, the probability of occurrence, impact to the overall project, and contingency plans. In iterative development cycles, risk management and identification are continuously reviewed in each iteration.

So, just as in a software project, we have a conference call with all the stakeholders. Since we don't know exactly when the RV will be fixed, we need a new plan. We can't change the end date: Jane is getting married whether we're there or not. If we stick to the current project plan, there's a high probability we will miss the wedding. Since Marian is the maid-of-honor, she is not pleased. She feels very strongly about taking immediate action to assure we get there on time.

We discuss several options. The plan we agree on is to have Tanya (in Jackson) fly to Baton Rouge to meet Sid. When the RV is fixed, we will go directly to Baton Rouge and pick up both of them at that iteration milestone. Eliminating the side excursion to Jackson gives us a seven-day buffer.

Tanya agrees to the new expense of her airfare. Of course, she isn't the only one bearing unforeseen expenses: We have the RV to repair, and everyone has a different opinion about how the new expenses should be handled. After some discussions, we eventually arrive at a solution that is acceptable to the team (this is similar to acquiring additional equipment and resources needs in a software project to recover from an unexpected setback).

## Refining the contingency plan

Because the RV repair time is still unknown, Marian is nervous about making our final destination on time. So she suggests a refinement to better offset the risk: Limit the time we wait for the RV repair. For instance, if the repairs take longer than five days, she suggests we abandon the RV and use a different method of travel. In software development, this is called timeboxing the "wait" or "delay" cycle. This method allows us to control the unknown items, which eliminates the day-to-day float that takes most projects off schedule. Marian looks into various travel methods like plane, train, and bus. Considering everyone's budgets, travel schedules, comfort, and other concerns, she recommends that if the RV isn't drivable within five days, we take a train to Orlando. Tanya and Sid would leave from Baton Rouge, and we would go directly from San Antonio. I volunteer to stay with the RV, understanding that I might miss the wedding. So now we had an iterative and refined contingency plan for the risk that the RV may not be fixed in time.

Marian wonders if we should call and tell Jane about our current situation. Jane is much like our customer in a software development project, who is interested in the final delivery of our software application. After discussing it, we decided that Jane has too many other things to worry about. If we have to put our contingency plan into effect, we may want to give her a heads-up. But right now, we think we can contain these events. Any one of us can take the train (or even fly) directly to Orlando, at any time, decoupling the dependency to the rest of the family. Even though traveling together was an important "requirement" at the start of the project, we understand that sometimes priorities change.

In times of crisis, the requirements can be altered to fit the current situation or need. In this example, the requirement for all of us to get to the wedding remains intact. But because of current circumstances, we've changed it a bit -- i.e., it's no longer mandatory that we get there all together or at the same time. The final goal is still met, but it's accomplished in a slightly different manner. In an iterative development lifecycle, modification to requirements based on the current situation or need is not only acceptable, it's actually expected.

Marian agreed that if she felt anxious about the time, she could always go directly to Orlando on her own. In iterative development, this is similar to deciding how much change you can completely contain within your group before raising a red flag that affects external teams.

Now that we have a plan in place for our current situation and a contingency plan for our risk, we can relax a little. We won't have the RV for a few days, so we elect to find a nice bed and breakfast. The hosts are gracious, fun-loving, and knowledgeable about the area. They are sympathetic to our story and promise to make our few days of captivity as enjoyable as possible. We use this creatively, in ways that expand our knowledge.

## Adjustments and opportunities

In iterative development, we often encounter delays in a project, which can give us time to reduce the defect inventory and stabilize the iteration's deliverables, and thus help keep the overall mission on track. Often delays occur while waiting for other parts of the design project to synchronize with the main branch. But these delays can present opportunities for other team members, who are given unexpected time to stabilize and reduce defects in the baseline code.

And wouldn't you know it? The RV is actually fixed in three days; we are on the road again, but because we're

a little wary of the stability of the RV, we go directly to Baton Rouge. In fact, we arrive ahead of schedule, and Tanya's flight from Jackson doesn't arrive until the next day. So we visit with Sid until Tanya catches up with us.

At this point, we are pretty much at end-game. Let's take a closer look at the iteratively developed vacation events. As we look back over the progress of the trip, we can chart the iterations (both planned and actual) as shown in Figures 2 and 3. Figure 2 portrays the vacation plans at the start of our vacation project. Through constant strategy reviews, adaptability, and iterative task planning, our actual travel plans successfully resulted in Figure 3.

**Figure 2**

| | |
|---|---|
| Iteration 1 | |
| | Borrow Van |
| | Visit friends in Phoenix |
| | Pick up Marian in Sante Fe by May 5 |
| Iteration 2 | |
| | |
| | Other places |
| | Visit Austin |
| | Visit San Antonio |
| | Visit Houston |
| | Pick up Cid in Baton Rouge |
| Iteration 3 | |
| | Visit Columbia |
| | Visit Hattleburg |
| | Pick up Tanya in Jackson |
| Iteration 4 | |
| | Visit Tallahassee |
| | Visit Gainesville |
| | Visit Daytona Beach |
| | Arrive in Orlando by June 3 |
| | Planned Happy Ending |

**Figure 3**

| | |
|---|---|
| Iteration 1 | |
| | Borrow Van |
| | Visit friends in Phoenix |
| | Pick up Marian in Sante Fe by May 5 |
| Iteration 2 | |
| | Added side-trip to Darla's house to drop off dog. |
| | Removed other visits |
| | Visit Austin |
| | Visit San Antonio |
| | Have to fix RV |
| | Tanya Travels to Baton Rouge to meet Cid |
| | Wait for RV to be fixed in San Antonio |
| | Pick up Cid and Tanya together in Baton rouse |
| | |
| Iteration 3 | |
| | Arrive in Orlando early |
| | Visited Disney World |
| | Actual Happy Ending |

**Figure 2: Original iteration plan at start of vacation**

**Figure 3: Actual iteration activities during vacation, adjusted from those shown in Figure 2**

## Continuous testing

Just like the previous pickup points (iteration milestones), we "test" to ensure that we've met the proper luggage limits and budget, and have all the items we need to bring to the wedding. And... ah ha! We find that, in the rush to secure alternative travel plans, Tanya forgot the items she was supposed to bring for the wedding. But thanks to our iterative testing at each iteration point, we caught this oversight before the final deadline (the wedding itself). Not a big problem. We do some manageable shopping to get Tanya back on track. (This illustrates the importance of continuous testing at each iteration and verifying our iteration entrance and exit criteria as our software project progresses.)

Now there is only the ten-hour trip to Orlando left. We take a leisurely drive, sightseeing at Mobile, Alabama, and

Tallahassee, Florida. We eliminate the other side trips and arrive in Orlando ahead of schedule. Some of us spend a day at Disney World, while Marian uses her time for her bridesmaid duties.

## The iterative adventure

As you can see in this simple example of iterative traveling, we've encountered similar events to software risks, feature creep, unexpected breakdowns beyond our control, and recovery plans. Just like our travel plans, iterative development and testing can be used effectively to increase the probability of meeting your software schedules.

## Notes

[1] Barry W Boehm, "A Spiral Model of Software Development and Enhancement," IEEE Computer, May 1988, pp. 61-72.
[2] Philippe Kruchten, The Rational Unified Process An Introduction, pp. 7-8.
[3] Acceptance criteria, such as entrance and exit standards, should continually be reviewed and evaluated based on continuous knowledge and project refinement. It's even acceptable to alter them, if appropriate, based upon new learnings or course changes.

## About the author

**Laura Rose** is the quality engineering manager responsible for automated performance test tools at IBM Rational. In addition to leading projects in both software programming and testing environments, she has thirteen years of programming experience and ten in test management. She has been a member of the American Society for Quality, the Triangle Quality Council, and the Triangle Information Systems Quality Association, and has published and presented at various test and quality conferences. You can reach her at llrose@us.ibm.com.