Next steps:
The near future for software development
Editor's Notes

Call this "The Future Issue" of The Rational Edge. Not the distant, dreamy Isaac Asimov, science-fiction kind of future. But the near one, over the next 18-24 months or so. Do you wonder where the IT industry is heading? As the economy begins to move beyond the dotcom crash, it seems this industry is undergoing a self-examination led by both skeptics and optimists. For every headline that reads something like "The Over-inflated Myth of Web Services," there's another article heralding the power of a burgeoning Web Services infrastructure. While some believe that cybertheft is slowing the advance of e-commerce for consumers, others are confident that the growing market for security technologies suggests a rosy future for this sector.

For Rational customers and other Edge readers, we believe the real question is: "What does the near future hold for software development organizations?" To answer this, we began by interviewing representatives from three analyst firms about e-business trends, and from their discussion we created a two-part article. This month's installment focuses on e-business infrastructure and strategies. And there's more: IBM's director of worldwide communications offers his insights on the telecommunications industry, where he envisions numerous opportunities for software developers to deliver the next generation of telco capabilities. And Jason Bloomberg is back with another aspect of Web Services for us to consider: the principles of service-oriented application development.

Among our technical stories this month, you'll find techniques for business-process modeling with the UML, plus some guidelines on reverse engineering your legacy code with Rational Rose. For you project leaders and managers out there, Process guru Karl Wiegers advocates a contingency buffer in your project planning; and one of our RUP experts, Gary Pollice, shows us the very real need to involve customers early and often in your RUP-based projects. Sid Fuchs offers advice on securing your company's future through sound investment in the most important asset: your employees. He advocates creating a strong mentoring environment.
that helps employees grow and enables companies to "weather the tough
times and accelerate through the good ones."

Have you ever considered the ratio of men to women in the IT industry?
Check out this month's review of *Unlocking the Clubhouse: Women in
Computing*, which reports on a recent study that explores why so few
women, compared to men, enter the various technology fields associated
with computing. Most boys who are interested in computers have no idea
how different -- and difficult -- the future appears to girls who have similar
interests. If you're a man, I guarantee this review will make you think,
especially if you have a daughter.

Happy iterations,

Mike Perrow
Editor-in-Chief
Q&A with Industry Analysts

How Are e-Business Trends Impacting Developers and Development Teams?

Part I: Infrastructure and Internet Strategies

Development teams have an interesting relationship with e-business. On one hand they are responsible for driving e-business forward, building the technologies that propel it. On the other hand, they are driven by e-business -- constantly responding to new demands, new challenges, and the new technologies they create. In this relationship, in which all the parts are moving quickly and at the same time, it can be helpful to take a step back and look at the bigger picture. Recently, Rational Edge writer Jack Wilber asked three respected industry analysts --Liz Barnett of Giga Information Group, David Truog of Forrester Research, and Jason Bloomberg of ZapThink LLC -- for their insights and opinions on major trends in e-business and how those trends affect developers and development organizations. In Part I of this series, we share their thoughts on the demands development teams are facing and the technologies, tools, and strategies needed to address these demands.

Jack Wilber for The Rational Edge: In the aftermath of the e-commerce boom, we've seen predictions that companies would invest more of their e-business dollars and resources in corporate infrastructure to accommodate partners and integrate their supply chain. Is that what you're seeing now, or are businesses making other technology changes to pursue more practical and profitable business models?

Liz Barnett, Giga Information Group: I see a tremendous amount being invested in infrastructure -- the ability to deploy the new technologies and support business processes in a way that is cost-effective and pragmatic for IT
organizations. But integration problems are very significant, and solving these
problems is critical, whether it's for e-business, supply chain, or internal
integration.

The new infrastructure definitely poses new development challenges. It
impacts your process and therefore trickles back to the tools that you use and
new component models that you need to think about.

David Truog, Forrester Research: Actually, we are not seeing a lot of
enthusiasm for buying more of that kind of infrastructure right now, but we
are seeing a focus on making the most of what is already there. During the e-
commerce boom a lot of companies made huge investments at a frenetic pace -
- buying packaged applications and software to establish a Web presence and
implement e-commerce strategies. Now that things have slowed down, people
are looking at what they have. They are purging some of it and integrating
and extending the parts they want to use. This is where software development
teams have a huge impact, because they are the ones knitting these things
together.

On the other hand, if the question is: Are companies investing their software
development dollars more in connections with partners and the supply chain
as opposed to optimizing internal processes? Then the answer is "Yes." There
is increased investment in software for interconnecting companies. And that
does have a big impact on the industry, on software development practices, on
what programmers think about, and even on software architecture.

Jason Bloomberg, ZapThink: One thing to keep in mind is that we're in the
dip of a curve right now. When you have explosive growth and then a crash,
the period after the crash is the Golden Age of the current technology.
Although people do lose confidence after the crash, it's actually the beginning
of the best time. And that's what's happening now with information technology
as a whole. We're in a recession, so everyone is circling the wagons and
focusing on return on investment (ROI). People are thinking in terms of "Let's
just make sure we are saving money and making money." They're not inclined
to spend a lot of money on future technologies.

If an investment in infrastructure can better their short-term ROI, however,
then people are doing it. For example, if it is clear to a company that
upgrading their operating system will save money this year, then it is
something they will consider. But the potential has to be clear and relatively
short term.

Often what happens during the Golden Age of a technology is the development
of supporting and enabling technologies. So far, we have the Internet and we
have the Web, but many pieces are not in place yet. So a lot of the software
development right now isn't for the big picture stuff, but rather for the small
pieces -- removing all the roadblocks that prevent things from working
smoothly, developing technology that enables you to integrate systems cost
effectively and integrate supply chains. It's a time to get down and do the less
glamorous work of making all the nuts and bolts fit.

As I see it, Web Services is the number one enabling technology on the
horizon. Once Web Services get rolling, it could really revolutionize the entire
software industry and information technology in general. And that represents
more than an evolutionary step; it is bringing about a paradigm shift from a component-based architecture vision to a services-oriented architecture vision.

**JW:** What demands are developers and development teams experiencing as a consequence of these technology trends -- the new focus on selective infrastructure investment, on filling in the blanks for the Internet and the Web, and on Web Services?

**LB:** First, we have to realize that this whole next generation of technology is an order of magnitude more complex than the generation before it. So when you start thinking about deploying some of these component-based and Web Services-based environments and integrating them with legacy environments, then the demands on development teams are quite significant. Organizations have trouble adopting new technology under any circumstances, and this is probably the greatest challenge they've faced so far. But the truth is, the success of a technology adoption is really tied to process. Some of it can be manual process and just good discipline. In many cases though, a process is the glue you need to integrate these different technologies and to make developers more productive.

**JB:** Developers are the ones who have to build the key enabling technologies -- Web Services in particular. Web Services enable integration and reduce costs. System integration has been a major part of IT budgets for the last several years: Sometimes as much as 75 percent of a development project is simply getting it to work with the other systems in the enterprise. Companies are looking to use Web Services to save money by reducing integration costs. But Web Services are still bleeding edge; the promise is still way ahead of the reality. People are using them already to solve real business problems, but they are still only scratching the surface when it comes to the potential for Web Services.

Ultimately, developers are the people who have to deal with these bleeding edge technologies -- with the vaporware problems, bugs, shifting functionality -- all the problems you get when you're working with brand new software and brand new platforms. These are the things that developers are really concerned about. The CIO will say, "Let's do .NET" or "Let's do IBM's Web Services," and everybody will say, "That's a great idea." But the developers have to translate that into reality, and they'll often find that some of the code is still in Beta, that there are security problems, and so on. The list goes on and on.

**DT:** As a developer, if you're writing a system -- code, data structures, APIs -- that will be consumed by others in your organization, then the requirements and expectations are very different from a situation in which you are designing and building a system for someone you don't know and who you might never meet. In the extreme case, for Web Services, someone else's code is supposedly going to discover your code by discovering a WSDL file that they pluck out of a UDDI directory. In such a case, defining the interface becomes a much more important and rigorously controlled process. That's something that I believe a lot of programmers are not used to thinking about.

I've always felt that the best programmers think in terms of data structures first. In other words, their first instinct when they fire up the text editor is to declare a data structure or class, not to start writing lines of procedural code. I
don't think this is just my bias; it's generally accepted as a better way to go. In the case of business partnerships, when you're designing a Web Service, you need to think about the most effective data structures and message format -- and therefore XML schemas -- for integrating with partners you know and possibly, in the future, partners that you don't know. That is a challenge for a lot of programmers out there.

Also, you have to treat your software as a product. That is something that software vendors are used to doing, but not typically companies that use software. You have to document it more carefully, version it, and control releases of it. A text file or Word document that describes it is no longer sufficient. You may have to create an XML schema with annotations that describe the meaning of each field. That's a bigger challenge for most developers.

**JW:** Do you think most development organizations have the right strategies and technologies to respond adequately to the demands they're facing?

**LB:** I think they are working on it. There's a tremendous challenge right now. We talk about "faster, better, and cheaper," and the question is, "Can you do all three?" For a few years, "faster" was everything. Time to market was the imperative. Then for a while, quality started to suffer, and people focused on the "better" dimension. Now, with budget cutbacks, the cost pressures are so tremendous that "faster" has become even less important. Today, we're seeing much more of a balance: Quality is as critical as speed -- if not more so -- and the cost pressures are dramatic. So the three dimensions are really vying for top priority, and the order keeps changing. The tension there is really, really apparent to developers.

Do development organizations have the strategies and technologies they need? I don't know that they have them yet, but I think they clearly recognize that they need them. We're seeing a lot of changes in IT now, specifically toward custom development and component development. And organizations are asking not only "How am I going to do this?" but also "How am I going to do it faster, better, and cheaper simultaneously?"

**JB:** I would divide development organizations into two categories: those in companies that are building the tools and those in companies that just want to use the tools to solve their business needs. In the Web Services tool building space, companies are making great progress -- those organizations have the right strategies and technologies. As far as the companies using the tools, that's a lot more spotty. Some of them have the tools, but as a rule they don't understand enough about them to really have strategies.

It's always that way with new technologies; when you introduce something new, people continue to think of it in terms of the existing technologies of the time. When the first automobiles came out, for example, people thought of them as horseless carriages: They did the same thing as a carriage, except without the horse. All the early automobiles looked like carriages, too. It took twenty or thirty years before automobiles were really designed to be automobiles. Today, our early Web Services look like RPC architectures, like CORBA and DCOM. They are a little bit better -- they're a little easier to use and based on open standards -- but they are still in the horseless carriage
stage. So I think the challenge for development organizations is not just to adapt to the new technologies and use the new tools, but also to help bring about a shift in thinking.

DT: To really respond to these demands, I think that organizations will have to do more custom software development. Many people may disagree with this. They'll say, "You should buy packaged apps; you should not build software." My point of view has always been that, to compete effectively, you damn well better build software. Of course, you want to avoid building software that someone else has already written. You can buy that and then build software on top of it. If you're not writing software, then you're merely using a commodity that your competitors have access to as well. Any company that wants to compete in a medium that requires software -- the Internet, for example -- that does not invest in custom software development is effectively saying, "We're going to be just like our competitors, and we'll just compete on price." Most companies don't want to do that, unless they're Wal-Mart.

Forrester generally focuses on companies that are developing Internet-based and Internet-oriented software, as opposed to vendors that are going to sell software on a CD. For many of these companies developing Internet-oriented software, I'd say they lack a real understanding of professional software engineering. Often the development organizations of these companies were born in so-called new-media departments, and they tend to have a lot of scripting cowboys on staff who don't really understand design and engineering practices -- like process, quality control, and making tradeoffs. Most people developing Web sites, especially, aren't skilled in these areas. If these companies are going to succeed, that will really have to change. Those people are going to have to learn to architect data structures effectively, and class hierarchies will be next. The answer lies in a combination of skills, process, tools, and architecture. Does every developer have to learn how to write native XML Schema? Probably not. There will be modeling tools to help them. But they still need to be able to think that way -- and that's the skills issue. It maps to how you architect a system that takes advantage of those classes, and also to the development process you use for designing, implementing, and testing them. The skills and process are what's lagging behind at this point. There's a lot of knowledge about how to engineer, but it is just not applied often enough in software development -- and especially not in site development teams.

JW: You've touched on the importance of development processes and development automation in the new e-business environment. Can you elaborate on that?

DT: In his book After the Gold Rush, Steve McConnell talks about creating a true profession of software engineering. He notes that there was a time in e-business that was like the California Gold Rush, when any non-professional could get very rich. Eventually, though, the supply of gold that was easy to find dwindled, and you needed to have a little more skill and a better process. I think now a lot of people are going to find that their skills and their process are not quite up to snuff -- especially for inter-company software projects. True software engineers will be at more of a premium than ever, and the amateurs of yesterday will be scrambling to get good at real software engineering.
In many cases, the development process and the development environment are one and the same -- the process is embedded in the tools.

There are two dimensions here, and they're both really, really important. One is diversity in what it is you're building, and two is diversity in who is doing that building or development. First, all of the software artifacts, components, Web Services, documentation, specifications, models -- all of it needs to be managed cohesively. If you want to have any kind of reuse initiative at all and share these things across teams, then what you are sharing and how you are going to share it becomes a process in itself. Second, there's the issue of diversity in development. Who makes up your application development organization? These days we have content creators, graphics people, and business users, and they all play a really important role in development. The development tools and processes have to address all those constituencies, and the development technologies of the past really couldn't do that. Today, there are all different kinds of people building all different kinds of software -- so it's a two dimensional "diversity matrix."

The process must be relevant to all the different kinds of people who are building the software. A good example is change management. You can deal with software change management in one way with tools and a vocabulary designed for developers. But if you are dealing with content management or overall asset management for a non-technical audience, then the tools you give them and the vocabulary you use have to be very different. Although in essence, the same types of process and technology are needed to manage change, you have to address the different constituencies and the different types of assets.

There's a concept of agility that plays out through development and deployment, and that means being able to adapt software development processes, tools, and the deployment environment. Integration is the hub for this. The imperative is really to deal with the diverse environment today, to come up with pragmatic solutions to manage this diversity.

I see a definite trend toward agile processes. You see this with lightweight processes like Extreme Programming and Scrum, but you also see it with the Rational Unified Process (RUP). Done right, the Rational Unified Process is just as agile as Extreme Programming or any of the others. My sense of the RUP is that it has been a closet agile process all along. There's a lot of stuff in there, so you can also use it in plenty of non-agile ways; it's powerful enough to do that. A lot of big companies and government shops really need that sort of heavy-documentation, lots-of-artifacts approach. But that's not really the vision of the RUP or other agile processes in which the idea is to pick and choose the bare minimum of artifacts that you need to get the project out the door. What I see happening is that some of the agile principles are extending beyond software -- because the line between what qualifies as software and what qualifies as business is blurring as companies become e-businesses. The whole point of an e-business is that the "e" and the "business" become one. It is not: "Here are the techies, and here are the business people. This is software and this is not." The line is much softer, and that goes hand-in-hand with software development getting more agile.

Also, one of the key principles of agile development is to involve the customer or user. As more companies adopt that practice, teams won't be limited to just
software development; instead, you'll see software/business process teams that are able to use agile processes and techniques across the enterprise. I see that as a very important trend.

The promise of development automation, basically, is it makes it easier for less technical people to do development. As development becomes less and less about just the software -- because the software and the business are becoming one and the same -- development automation will cause the code to disappear in the way that the technology inside a car has disappeared. In the old days, if you wanted to drive across the country, you had to be a mechanic; you had to carry a bunch of parts with you, and starting the car involved walking around to the front and cranking it. It was a real hands-on thing. Now, though, you just get in the car and drive, and you forget you're in a machine. The technology has disappeared into the background.

Similarly, as software development becomes more automated, the code part is going to disappear. Today we still have mechanics and people who want to roll up their sleeves and change the oil in their cars themselves; it will be the same with code; you'll always have people who want to get into it. The code will not really disappear, of course. It will still be there, but it is going to fade from the day-to-day activity of putting software together.

In next month’s issue of The Rational Edge, our analysts will focus more on Web Services, and on development platforms and their implications for developers. They’ll also offer their opinions on the one thing that development teams are not doing that they should be doing to help them succeed in the future.

Glossary

**CORBA** -- Common Object Request Broker Architecture
A language-independent and operating system-independent specification for distributed object interoperability.

**DCE** -- Distributed Computing Environment
Allows computers from a variety of vendors to communicate transparently and share resources such as computing power, files, printers, and other objects in the network.

**DCOM** -- Distributed Component Object Model
Extends the Component Object Model (COM) to support communications among objects on different computers on network.

**EDI** -- Electronic Data Interchange
The transfer of data between different companies using networks, typically to buy, sell or trade information.

**ebXML** - Electronic Business using eXtensible Markup Language (see XML).
RPC -- Remote Procedure Call
Specifies a low-level mechanism for calling a program running on a distant computer.

SOAP -- Simple Object Access Protocol
Provides a mechanism for inter-object communication across HTTP.

UDDI -- Universal Discovery Description and Integration
Provides the means for organizations to register, find, and then use Web services.

WSDL -- Web Services Description Language
Defines the XML-based language for specifying Web services.

XML - eXtensible Markup Language
Defines a mechanism for attaching tags and attributes to the structural elements of a document.

Notes

1 Except where noted, the descriptions used in this Glossary are from Grady Booch's article ARRG (Acronyms: a Reasonable and Rational Guide) on The Rational Developer Network.

2 Source: The Rational Unified Process


4 Source: http://www.webopedia.com

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
Web Services and a New Approach to Software Development

by Jason Bloomberg
Senior Technologist
Ashton Services

Web Services is by far the hottest topic in software circles since the dotcom implosion, but sometimes it’s hard to see what all the fuss is about. The most immediate promise of Web Services technologies like SOAP (Simple Object Access Protocol) and WSDL (Web Services Description Language) is that they will simplify the integration process. Now, there’s no question that integration -- especially on the enterprise level -- can do with some simplification. After all, up to 75 percent of the expense in many large-scale software projects often goes toward simply integrating new systems with old. Anybody who’s been in the trenches in any EAI (Enterprise Application Integration) project knows that the hard work is in building the wrappers -- the special code that allows disparate systems to communicate. The promise of Web Services is to simplify this wrapper code by boiling it down to XML-based standardized interfaces. That means that all wrapper code would follow the same standards, and integration would become simpler, and hence, less expensive.

If, in the end, Web Services do nothing more than reduce the cost of integration, they are still sure to be worth the trouble. However, if that’s all they’re good for, then they will be merely a relatively small evolutionary step in the discipline of software engineering. Not to worry. If you read my article in the March issue of The Rational Edge, then you know I believe Web Services have a much brighter future than this, although they are still in an early phase. Whereas my last article focused on business changes that might be wrought by Web Services, this article focuses on the world of software. As you read, keep in mind that today’s new Web Services-capable software development tools, while clearly more mature and sophisticated than their predecessors, still only scratch the surface when it comes to building powerful Web Services applications. This limitation is not the fault of the tools; rather, it is due to the limitations of the entire discipline of software development.
The Seven Principles of Service-Oriented Development

Software development has come a long way since the early procedural programming days. Object-oriented analysis and design led to distributed object frameworks, which in turn led to today's distributed component frameworks. Service-oriented development is the next logical step in this progression. However, just as developers had to "unlearn" procedural programming in order to fully grasp object-oriented techniques, so too must they adapt to the new service-oriented development principles that Web Services will require.

To break free from the old ways of thinking, software professionals must have both an understanding of the capabilities of the new tools, as well as the big picture of what can be built with them. The principles of service-oriented development that follow -- expressed as transitions from the old way of thinking to the new -- are, effectively, guidelines for reaching this new level of understanding.

Principle #1: Static Components -> Dynamic Services

Traditional component-based software development naturally focuses on building and integrating components. To build a component, the developer takes the specifications and puts together various pieces of code in an attempt to make some software that satisfies the specifications. In other words, the developer is done if the software does what it’s supposed to do.

If the developer is building a Web Service, however, it is not enough to build the Service with set functionality as specified. Instead, the Web Service should be defined by a dynamic description of its functionality, as shown in Figure 1. Such dynamic descriptions are currently expressed in a WSDL file for the Web Service. The developer, therefore, need only indicate where the WSDL file is to be found, and the software that seeks to invoke the Web Service can find the functional description at runtime.

One of the most significant ramifications of this principle is that it mandates the separation of the presentation logic from the business and persistence logic within a system that uses Web Services. When a developer builds a Web Service, he or she may have no idea whatsoever how that Service will be invoked, or what the user interface on the invoking software will be. Now,
n-tier systems traditionally separate presentation logic onto a different tier from the business logic and persistence tiers, but in the case of n-tier architectures, such separation is motivated by the benefits of manageability and reuse. Therefore, an n-tier architect has the option of mixing up the functionality of the tiers somewhat if there is a benefit that outweighs the loss of reusability and manageability. Not so with Web Services.

**Principle #2: System Integration -> Service Exposure and Reflection**

Today's approach to integration begins with system-level requirements. Depending on what the system is supposed to do, the architect plans out the various components and how they should be integrated. Instead of taking this top-down approach, service-oriented development takes a bottom-up approach. Each component in a system should be exposed as a Web Service before any system architecture is in place. Then, each service uses reflection to provide external systems with the information they need to access the data and functionality encapsulated in the service. (*Reflection* means that the Service allows the invoking software to query the Service about its interface.)

![Figure 2: A Service Assembly for a Service-Oriented System](image)

When building a system using service-oriented development, then, the architect starts with the system requirements and performs a *service assembly* step, as shown in Figure 2. Service assembly corresponds to identifying the appropriate components for the system, except that in the Web Services environment, the architect accesses the dynamic descriptions of the Services (as described in Principle #1). Once the required dynamic Service descriptions are identified, the architect can specify the structure of the system, even though the individual components and their interfaces are not fully described, and won't be until runtime.

There are two important aspects to Principle #2. First, when each data source or business object is exposed as a Web Service, the developer need not know how systems will access or utilize that Service. Second, when the system architect plans the system, the actual nature of the components (which application, platform, etc.) is irrelevant.

**Principle #3: Coding for Reusability -> Coding for Broad Applicability**

Coding for reusability is one of the pillars of object-oriented programming.
Write the code once and use it many times, thus saving both money and time. In practice, however, writing reusable code can be far more difficult than writing code for a single-use application. There are three main problems with attempting to write reusable code:

1. It takes more time than one-off code.
2. There's no guarantee that the code will actually be reusable.
3. What the developer produces may not be consistent with customer requirements. In many cases, the developer doesn't even know when to stop coding.

For these reasons, agile software methodologies like Extreme Programming (XP) shun reusability. In XP, developers code only what the customer actually needs. The customer actively participates on the development team, providing user testing and guidance. If extraneous functionality creeps in, then developers refactor the code, which means they rework the code until it's as simple as possible.

When developers refactor, they streamline the code until it is general enough to handle all the cases in which there was redundant code before. In essence, refactoring means building for a kind of reuse. However, refactoring is different from traditional coding for reusability, because its aim is to create code that is flexible and broadly applicable.

The difference between reusability and broad applicability is subtle, but essential to the Service-oriented development process. The first two principles imply a level of developer agnosticism with respect to the functionality of a particular Web Service: The description of the Service is dynamic, and the developer need not know how the service will be used. What, then, keeps the developer on task and scope creep in check? The answer is to couple the architectural principles of service-oriented development with the engineering principles of agile development. Development of Web Services should be an ongoing, iterative process that actively involves the opinions of the users. The services themselves, then, should be constructed to be as simple as possible, and developers can continually refactor them so that they are as broadly applicable as practical.

Principle #4: Disruptive Upgrades -> Ad hoc Upgrades

Modularity, along with reusability, is another fundamental tenet of object-oriented programming, and hence component architectures in general. If a system is modular, that means the individual components that make up the system can be upgraded or replaced without affecting the rest of the system. There is no question that modularity makes such good sense that even the most non-technical business user can understand it. Unfortunately, in today's enterprise component architectures, modularity is largely a myth.

The problem is that replacing or upgrading components in a complex system is never as simple or inexpensive as people would like. Many times, system components are not fully encapsulated: Despite the architect's carefully laid plans, there is often some internal hack that provides some essential piece of functionality which the carefully designed interfaces do not support. In other
In cases, the API (application programming interface) includes some semantic ambiguities. For example, the old component's `getQuantity()` method returns the number of boxes, but the new component returns the number of pallets. And of course, new APIs often differ from the old ones they're replacing, which then necessitates substantial reintegration work.

One of the primary goals of service-oriented development is to solve this morass of modularity issues. Instead of simply exposing APIs, components wrapped in Web Services expose dynamic service descriptions. If the underlying API changes, then the service description adjusts automatically, and the other components of the system can adjust to the changes at runtime. And because the separation of the different logic tiers is now mandated instead of recommended, hacks that break the encapsulation of the Web Services are much less likely. (Of course, developers will still be able to break the rules and code badly, but following the principles of service-oriented development will eliminate the need for such hacks).

Once the dust settles and service-oriented development begins to work the way it is supposed to, enterprises will realize that upgrades of component-based architectures can take place on an *ad hoc* basis. Unlike today's legacy upgrade project, which can require multiple years, millions of dollars, and many hours or days of downtime, upgrading a component exposed as a Web Service will be a low-risk part of regular maintenance. From a business perspective, then, enterprise architectures can remain current, smoothly upgrading themselves as necessary.

**Principle #5: Top-Down Scalability -> Bottom-Up Scalability**

Any architect who has been called upon to scale up an e-business system that is experiencing unexpected surges in traffic knows full well that scalability is far more complex than adding an additional server here and there. The fact of the matter is, in today's world, scalability must be carefully planned far ahead of time. Architects will begin with predicted usage patterns (both average and peak) and use those estimates to plan for load balancing and failover. They must be especially careful to watch out for bottlenecks in the system; a site of 1,000 pages can run smoothly for months, but if one page is featured on the news, then the resulting surge in traffic can bring the entire site down.

In a fully realized Web Services-enabled environment, scalability can be handled in a bottom-up fashion, instead of via the fallible, top-down approach used today. UDDI (Universal Description, Discovery, and Integration) registries can be set up to provide lists of backup Web Services, solely for the purposes of scalability and failover. If a system is experiencing unexpected traffic, it can automatically find backup services in a registry, obtain their service descriptions (also stored in the registry), and bind to the supplemental services on the fly. Furthermore, registries providing access to supplemental Web Services can also be configured recursively: A system can query a registry for supplemental Web Services, and even if that registry has no information about a suitable service, it can contain information about other such registries. The needful system can therefore hop from one registry to another until it finds the extra Web Services it needs.
Principle #6: Platform Dependence -> Platform Irrelevance

Established service-oriented architectures such as CORBA and Windows DNA have never offered much in the way of platform independence. Naturally, Windows DNA makes no apologies about being a Windows-only architecture. And CORBA, while theoretically platform neutral, in practice has typically been able to achieve cost-effectiveness only in single-platform implementations.

At the enterprise level, EAI (Enterprise Application Integration) has become the predominant approach to building cross-platform solutions. Each EAI solution is typically a "bus" that lies between the application servers and other components that support the user interfaces on the one hand, and the back-end legacy data sources and various business components on the other. Each EAI bus may support both synchronous and asynchronous communication, and these buses often contain sophisticated transaction processing capabilities as well. However, although EAI enables enterprises to build cross-platform architectures, these architectures are all very expensive. The EAI software alone often runs into the hundreds of thousands of dollars, and even that amount pales in comparison to the integration costs involved in an EAI implementation. The problem is that the real work (read: money) that goes into an EAI implementation project is in building the wrappers that enable the various component systems to interface with the EAI bus. Therefore, most of the work involved in EAI is actually platform-dependent development work after all.

Web Services have gained attention in large part because they offer a way to dodge the expense of EAI. The big irony today is that the most contentious issues in the Web Services world are those of platform dependence. Microsoft's .NET initiative, their replacement for Windows DNA, is again, unapologetically, a Windows-based approach. The Java/J2EE camp, this time led by IBM, is circling its wagons and getting ready for a major platform brawl with the folks in Redmond. What's so ironic about this brewing battle is that service-oriented development promises to make the entire platform issue completely irrelevant.

The component-based way of thinking is to have components built on platforms that expose their functionality via their interfaces. Components then work with each other by accessing each other's interfaces. In essence, the components are soccer players who kick a ball to each other, and the platform is the field. Needless to say, it's difficult for two players to interact if they are on different fields.
The service-oriented way of thinking is that the entire software environment consists of dynamically described services that can be located and invoked on the fly when necessary, as illustrated in Figure 3. Instead of a platform in the true sense, the soccer field is now XML -- nothing more than a markup language whose extensibility allows all players to join in. Now, platforms aren't going away any time soon -- Web Services are still software, after all, and there still need to be systems that provide the user interfaces to the service-oriented environment. J2EE and .NET will therefore move to the periphery where the users are, providing the infrastructure necessary to write Web Services as well as the desktop, browser, and other interface support that users need to access those Services. In the middle of the fray, however, will be Web Services alone, playing on a field of XML.

**Principle #7: Dictatorship Software Model -> Federation Software Model**

Web Services' loose coupling solves many of the problems that result from the tight coupling found in component-based architectures with rigid APIs, including integration issues (Principle #2) and modularity issues (Principle #4). However, loose coupling and the other principles of Service-oriented development will have other, even more profound effects on the world of software. Traditional enterprise software (Siebel's core CRM offering, for example) is integrated at the database level, providing for an extraordinarily tightly coupled application suite. As component technologies have matured, enterprise software vendors have been more likely to build their software following object-oriented principles -- still tightly coupled, but now communicating via exposed APIs.

Taking this progression one step further, architects will be able to create enterprise software by assembling dynamically described collections of Web Services. Now, this involves more than simply taking a component-based architecture and using SOAP as the RPC (remote procedure call) or messaging protocol, which is as far as many companies are taking Web Services today. Software vendors who understand all seven principles of Service-oriented development will publish enterprise software that consists of a loosely coupled federation of Services, many found through registries on the fly, some not even developed in-house. As time goes on, the various Web Services that make up such an enterprise package will change from day to day, as developers perform *ad hoc* upgrades, and systems scale up and down at runtime by assembling supplemental Web Services as necessary. In
essence, tomorrow's enterprise software vendors will follow entirely different models -- both architectural and business models -- than today's vendors.

Where Will We Go from Here?

There's no question that the adoption of many of the principles we discussed above is still on the horizon, maybe as many as five years away. Some of them, however, are much more imminent. Software vendors who are planning their next product release must look a year or more out, because of the development time needed to bring a product to market. Looking a year out essentially means predicting the future, a notoriously fallible endeavor. A more reliable approach is to work within the context of a scenario -- a logically constructed series of possible events that are likely to take place, given certain assumptions about how the world will change.

The seven principles I've articulated effectively delineate such a scenario. Collectively, they can provide the context necessary for making intelligent predictions about the future of software architecture and development. Will everything I've outlined here come to pass? Probably not. Will this article instigate discussions that help people make decisions about where to concentrate their software efforts? I hope so.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
Software Development for Telecommunications: Looking Toward the Future

An Interview with Tim Greisinger
Director, Worldwide Communications Sector, IBM Software Group

The high stakes and frantic pace of technological and business changes in telecommunications make it among the most volatile -- and exciting - arenas for software application development. The global communications infrastructure is evolving rapidly, as service providers augment proprietary, voice-only networks with open, IP (Internet Protocol)-based networks that support an amalgam of voice and data services. Developers of the business, operational, and network systems software that will support or utilize this fast-proliferating infrastructure face new challenges -- as they attempt to integrate new services with legacy business systems, for example. In this environment, defining and communicating system architecture and design are more important than ever.

To find out more about trends in the telecommunications marketplace and their ramifications for software engineering, we asked reporter Scott Cronenweth to interview Tim Greisinger, Director, Worldwide Communications Sector, IBM Software Group. As a telecommunications industry expert, Greisinger specializes in the software infrastructures that service providers need to launch, integrate, and support new applications. He also helps focus IBM software offerings to meet the specific business requirements of the energy and utility, and media and entertainment industries.
**Q:** Over the past year, the telecommunications industry has been changing rapidly and profoundly. Service providers are scrambling to implement new services, new business models, new physical networks, new standards -- you name it. From your perspective, what are the biggest challenges these companies are facing?

**A:** The global telecommunications industry is being swept by three broad waves of transformation. The deepest and most fundamental of these transformations is undoubtedly the conversion of telecommunications networks from proprietary, voice-only infrastructures to IP-based, open networks that support an ever-growing range of voice and data services. These new networks define a whole new set of ground rules for the development of applications to deliver those voice and data services, add value to them, and support the business needs of the service providers that offer them.

In order to make the staggering investments required to rapidly build out these new networks, service providers must initiate and sustain two other transformations within their own businesses.

First, they need to find ways to generate new revenues. Traditional voice services have become a commoditized offering, really no different from electricity or gas. This means that service providers must differentiate themselves via new services and product offerings.

With margins driven lower and lower, it takes longer and longer to recover costs or generate a profit from an individual customer. Given a customer churn rate in excess of 20 percent, and even higher for wireless carriers,\(^1\) it's easy to see how absolutely critical it is for service providers to create a service model that will help them retain customers. Then, consider the costs involved in transforming the network. As an example, service providers in France, Germany, and particularly the UK have spent the equivalent of billions of US dollars just on licenses for 3G (third-generation), IP-based networks. And that's before the cost of the first antenna or the first modification to an application infrastructure required to deliver a service. To finance these long-term investments, today's service providers need new, differentiating services that will drive solid revenue streams.

Second, in parallel with the need to keep profitable customers and generate new revenues, service providers have a mandate to cut costs across every area of their business. Realize what a tough order that is. Not only must they build out new networks and deliver new services as fast as possible while maintaining their existing networks and serving their current customers -- they must do it all at the lowest possible cost. Efficiency is critical in every area, from the supply chain to business process improvement, to outsourcing, to maximizing new subscriber ROI.

It all adds up to a huge, worldwide race to bring about these transformations while the baseline voice business is being eroded and the global economy is sluggish.
Q: Let's talk about the implications of those new service offerings. Will developers be called on to build new software systems to support them?

A: What will involve a significant software development effort is the creation of an application delivery platform to support not only services but also business processes. One example of a vital business process is provisioning and billing for those new services. Companies are striving for what the industry calls "Flow-Through Provisioning." This refers to the spectrum of e-business capabilities that minimize the cost of marketing to and acquiring customers, taking their orders, processing orders through the call center and other business support systems, provisioning the requested service on the network, delivering it, and billing for it. Ideally, end-to-end provisioning should be completely automated; it should be possible for customers to conveniently "turn on" new services themselves. Then, if support is required later on, it can also take place via the Web, rather than through more costly call center interactions. Increasingly, customers want the instant access to information that the Web offers.

The capability for optimized business processes like Flow-Through Provisioning is as important as the viability of the services themselves. These new systems must simultaneously drive higher levels of customer satisfaction and convenience while streamlining operating costs.

"From where I sit, there's never been a better time for telecommunications companies to think about their software development process ..."

Think about what it might take to cost effectively roll out DSL (digital subscriber lines) to a new subscriber base. You want to minimize the cost and time required to get each new customer up and running, especially in light of the industry's high churn rates. Right now, some companies are looking at partnering with hardware vendors to prepackage PCs with pre-installed, pre-configured cards to enable DSL. The driving forces here are to avoid one-off processes, and the high cost associated with physically rolling a truck.

The bottom line is to reduce the need for low-value human intervention across every business process. This benefits customers because they can obtain a service automatically, on demand, without errors or delays. And the cost advantages for service providers are enormous.

Q: OK, but how achievable is Flow-Through Provisioning, really? What's required from a software engineering standpoint to make that level of service and efficiency happen?
A: For most telecommunications companies right now, the reality is that they have literally dozens of disparate systems comprising various parts of the provisioning equation. Frequently, a great deal of manual intervention is required to move information end-to-end, and process work such as exception handling still happens manually, especially across departments. For example, the call center may have no recourse but to fax information to a service crew. Obviously, that's a slow and costly process.

New applications and significant integration efforts will be necessary in most cases to implement new platforms, connect business systems to the new network, and deliver capabilities like true Flow-Through Provisioning. The level of effort required will lead the majority of service providers to enlist the help of an integrator, such as IBM Global Services. To meet the urgent demand for new capabilities, we anticipate that more and more service providers will look for software-based products that are specific to their needs and can be rapidly adapted to their unique environments.

In other words, service providers will not only seek help in integrating their existing systems, but will also look to vendors to rapidly deliver comprehensive solutions that they can execute -- end-to-end DSL provisioning, for instance, starting from the moment the customer places an order to the point where the new service is turned on and invoiced.

In this scenario, development teams will be called upon to deliver custom services that execute specific business processes in a specific organizational environment. Each company's combination of networks and legacy systems is unique; service providers will have to look beyond off-the-shelf applications. Plus, as new capabilities are added to existing systems, IT will be called upon to do upgrades and maintenance.

Q: Is there a component of Flow-Through Provisioning that most service providers are addressing first, in terms of priority?

A: Yes -- billing. Every service provider realizes that to compete even in the current market environment, their company needs to present a unified, portfolio view of services and charges tailored to meet the expectations of their customers. The old service-specific, commodity view, whereby a customer receives a separate bill from the same provider for long distance, DSL, and a cell phone -- that's no longer acceptable. It has become mission critical to support unified billing.

Obviously, a substantial amount of application development is required to achieve this. And it needs to happen very, very quickly; new and better services need to be in place ahead of the competition. And of course, each organization -- most likely in partnership with one or more third parties -- must continually integrate and coordinate more and more applications as it puts new ones in place.

Billing is an end-to-end process; it relates to the new, multi-service network and nearly every business system. That includes what we call BSS (Business Support Systems) services such as customer care, mediation, or
electronic bill presentment; and also OSS (Operations Support Systems) services like provisioning, network management, and service level monitoring.

Consider that a large carrier might easily have thirty legacy billing systems, from its national long-distance account market, to wire lines, to cable services, to the ISP space. Many might have been created in-house; others might have been brought in via mergers and acquisitions. Some of these systems might have been in place for twenty years or more; so you can't just rip-and-replace them, especially in this time of cost constraints. But somehow, service providers still need to offer a rich, collective view of the customer that supports convergent billing, applicable discounts, and rates based on usage patterns or business relationships, as an example.

One way to address this challenge is via an evolutionary, coexistence strategy rather than outright replacement. We think many service providers will choose to bring in new, more flexible systems that can provide real-time, event-based billing for things like movie downloads and video on demand. And a big part of deploying those new solutions will be to marry the new system with the legacy billing systems.

Q: Once you perform that marriage, though, you still have the challenge of supporting all those new business systems and services we talked about earlier. Do service providers think they'll be able to connect everything seamlessly?

A: Service providers today are looking very closely at ways to link systems and data without making point-to-point connections among hundreds of systems. For example, it's critical to integrate billing with customer care systems or CRM (customer relationship management)-type applications, which might be based on Siebel or SAP. This is absolutely essential if you want to present the customer with a common business face or to cross-sell/up-sell services.

Achieving this level of integration entails the ability to map data from one system to another, and to initiate business processes that move across systems. The better the job you can do in terms of deploying a middleware infrastructure that automates and connects people and systems through business process automation and integration, the more quickly and flexibly you'll be able to respond to market forces.

From the standpoint of application development, it's imperative for service providers to define business requirements and model the interrelationships among them. The goal is to define critical processes and optimize the systems and human interactions that participate within them. This makes it easier to identify key connection points, leverage legacy components, and provide a solid foundation for interaction with integrators, customers, and other third parties.

It's also important to consider modeling the way these systems exchange data. Development efforts on this scale make it very worthwhile to
consider data modeling as well as message modeling. It might also be advisable to implement various emerging Web standards, as well as applicable standards for systems design and development. By adopting standards for data, processes, and integration across the broadest possible range of services and departments, organizations can achieve the economies of scale that make these kinds of efforts really pay off.

Q: What are the critical development issues regarding communication among all these software systems -- both new and old -- and the new, IP-based networks?

A: Development managers in and around the network and support systems must focus on the means by which they will deliver new services dynamically, whether they're broadband, wireless, or land line-based. What must the application delivery environment look like from the IT and development perspectives?

Ideally, every service provider wants to be able to rapidly turn both internally and externally supplied services, applications, and content on and off. This type of environment will best facilitate third-party contributions, such as dating services, ring tones, movie clips -- you name it. Which is a very different perspective from the voice-only days, when the service providers "did it all" themselves.

In other words, the service providers want to create an application delivery environment that is not only highly scalable, available, and secure, but also highly flexible. An example of this is NTT DoCoMo's highly successful iMode architecture, a mobile Internet access system based on cHTML (compact HTML), running over the CDMA (Code Division Multiple Access) transmission protocol. iMode is one of a number of emerging solutions that successfully separate the network from the applications that run on it. IBM recently announced its Service Provider Delivery Environment to help address this broad-based need for a dynamic application delivery platform.

Another big IT challenge in telecommunications concerns the huge business opportunities to be pursued in providing commercial applications -- for hosting e-mail or supporting enterprise mobile services, for example. But, again, to capitalize on these opportunities, the application delivery environment must make it possible to provide the service and bill for it in a way that's convenient for the customer and cost-effective for the provider. Take as an example traditional services, which bill per minute of voice usage at a flat rate. Can you bill for real-time usage per click, per download, per e-mail?

Usage-based services require fundamental transformations. Every service provider needs and wants an open application delivery environment that's supported by integrated BSS and OSS services, providing a common login, security, directory services, customer data, billing -- all the service infrastructure that must be in place before you can drop in new services at will.
Say you want to offer a service that sends subscribers an e-mail when an American athlete wins an Olympic gold medal. The window of demand for that service will be only a few weeks, maybe even a matter of days. How do you quickly roll out that new service from a third party, and share the revenue and provision and bill the customer? You can't think about doing that if your delivery environment isn't built for it. You can't mount a one-off network integration effort for every application you want to host.

**Q:** What's it going to take to achieve that kind of network infrastructure, from an IT perspective?

**A:** Partnership will certainly be key, whether it's at the level of integrators and/or applications and/or hosting. It's important to mention in this context that service providers generally have a strong skill base when it comes to executing on networks that are proprietary, analog, and switch-based. However, general-purpose computing skills are becoming far more relevant today on the network side of the telecommunications industry, with the conversion of the network to IP. Business partners like IBM and Rational Software represent one way to bring these skills to bear on the new network.

In terms of new technology, for example, open standards are being developed in telecommunications, such as one called Parley that IBM is involved with, which are meant to open up the network even further -- so applications outside the network can utilize network services. That's unlike the typical situation today, where implementing a service requires changes to the network. And these are highly structured, difficult changes to make; they require specific sets of skills, like the ability to program switches. That doesn't allow you to leverage people with general purpose computing skills. Parley, on the other hand, allows non-network programmers to use Java to link business and consumer applications with services and functions previously only accessible from within the network.

The introduction of these new standards would bring about a fundamental change in application development that's specific to the telecommunications industry and its particular infrastructure transformation. Think about the number of people who know how to program a specific model of switch, versus the number who can code in Java. With the ability to leverage general purpose computing skills, companies could get more work done, and get it done faster and more cost effectively.

From where I sit, there's never been a better time for telecommunications companies to think about their software development process and how to make it as efficient and repeatable as possible. Perhaps no industry would benefit more from visual modeling, a standardized, well-defined software engineering process, and so forth. The impact of adopting these practices on integration efforts, reuse efforts, maintenance efforts, and overall developer productivity would be huge.

Telecommunications service providers are struggling to be relevant --
literally not to be bypassed in this dynamic environment, and not to end up relegated to providing a commodity voice service. They have to move quickly because both the business and consumer markets are demanding and expecting up-to-date services. The winners must not only build out services faster, but also weather the current economic pressures and manage staggering deployment costs. The wildcard in the deck is the flexibility of the infrastructure. Because nobody knows for sure what the "killer app" on these next-generation networks will be.

Notes

1 For example, wireless phone customers in the US currently switch providers at a rate of about 3 percent per month.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
Saving for a Rainy Day

by Karl Wiegers
Principal Consultant
Process Impact

An article in a recent home remodeling magazine noted that "the prudent homeowner sets aside 20 percent of his budget for cost overruns and unexpected expenses." This sounds like good advice, doesn't it? Most remodeling projects include surprises, like the time the carpenter who built my deck spent an unplanned, exhausting half-day augering through a tangle of twisted tree roots to dig holes for the deck supports.

Most software project teams also experience cost overruns and unexpected expenses. They begin with uncertain estimates and encounter schedule setbacks. They experience requirements growth and technology shifts. They discover essential tasks that weren't planned. They are thwarted by slips in other projects that tie up people or resources they expected to be available on a specific date. Risks materialize into schedule-destroying problems. Life is just full of surprises.

But few software projects create a schedule or budget reserve to help them deal with such eventualities. To respond effectively to your project's changing realities, it's wise to save a little time and money for a rainy day.

This article makes the case for incorporating a contingency buffer (also known as management reserve or safety time) into your project plans to accommodate the unforeseen and the unknown. We'll discuss ways to determine the size of contingency buffers and how to include them in your negotiations with managers and customers. We'll also look at critical chain management analysis as a technique for deriving cumulative contingency buffers to account for estimation inaccuracies.

Selling the Skeptics

Before we dive into the mechanics, let's explore how to convince managers and customers that contingency buffers will increase our chances of success. Honest estimation and project planning often leads to schedules that senior
managers regard as unacceptable. When managers spot a task labeled "contingency" in the plan, they typically think of it as unnecessary padding that can be removed to shorten the schedule. They may not recognize that thoughtfully derived contingency buffers are a practical acknowledgment of reality, not a crutch for weak-kneed estimators who are afraid to make commitments.

If you're a project manager, discuss with your managers early on the value of including some safety time and money, instead of surprising them with a project plan that they will perceive as padded. Then, use these two strategies to produce convincing numbers:

- **Analyze Historical Records.** These can be your most effective weapons in battling unreasonable imposed schedules. Compare previous schedule and budget estimates with actual results, identify factors that contributed to differences, and use this information in your buffer calculations.

- **Use a Proven Method for Calculating Your Buffers.** You can make your case more effectively if you can describe how you came up with your contingency buffer durations. We'll examine a couple of options in the next section.

Once your plan is in place, resist the pressure to remove your buffers and make impossible promises, no matter what your managers, customers, or marketing want to hear. If your manager or customer claims that some other provider promises a shorter schedule without that pesky contingency buffer, ask how likely the other provider is to achieve that target. Does their company have better people and processes, or are they just more optimistic? Also ask about the other provider's track record of actual performance compared to estimates and promises (and be prepared to show your own).

In addition, you can point out that building in slack will increase the chances of delivering on schedule and avoiding litigation that might arise from an unsatisfied contract. I recently consulted on a lawsuit involving a vendor that began missing deadlines partly because extra communication cycles were required to pin down the client's requirements. As the vendor's schedule included no contingency buffers, it was impossible for them to absorb any of these slips without directly affecting the delivery date.

**Bring in the Reserves**

Project managers typically use one of the following approaches to calculate contingency buffers:

   Option 1: Add a "safety increment" to every task.

   Option 2: Place a buffer at the end of a set of project activities (e.g., a major milestone, a development phase, or an interim release) or as a separate task at the very end of the project schedule.

I do not recommend Option 1 -- simply padding your estimate for each task -- because it increases the planned duration and cost of the entire project by the
expansion factor without differentiating among tasks. Protecting the tasks on the project's critical path (the longest sequential path of essential activities that runs from initiation to delivery) against overruns must take top priority. Tasks that do not lie on the critical path already have some slack time following them; therefore, they can tolerate some slip without affecting the project's schedule as a whole and shouldn't need additional safety margins.

Of course, if a task that is not originally on the critical path slips by too much, it might move onto the critical path. In addition, if your estimated duration for a specific task is far too low, then it might look as though that task is not on the critical path, even though it really is. Even when using contingency buffers, you need to generate the most accurate estimates you can for each task.

Expanding each task estimate also increases the risk of succumbing to Parkinson's Law: "Work expands to fill the time allocated to it." That is, if you include a safety day in a task that you estimate will require four days of effort, then you're likely to spend all five days completing the task. It's an easy trap to fall into, although some studies have suggested that software people are not as prone to Parkinson's Law as you might expect. The fear managers have that Parkinson's Law will prevail, however, leads them to cut estimates and compress schedules to keep the pressure on. This is usually counterproductive. As consultant Tim Lister pointed out, "People under time pressure don't think faster," and software development involves a lot of thinking.

A far better approach is Option 2: to incorporate contingency buffers at the end of major development phases, at the end of the entire project, or both. In this scheme, you estimate how much additional money and time (as discussed below) you need to yield a high probability of completing on schedule and within budget. You assign these safety margins to discrete tasks placed at the end of your project schedule and/or after major milestones. Don't assign resources to these tasks, as they simply represent extra quantities of time or money that the resources already assigned to the work tasks might need to complete them. In the Rational Unified Process you could include a contingency activity either at the end of the Inception, Elaboration, Construction, and Transition phases, respectively, or at the end of each planned iteration.

As your project progresses, some tasks will take longer than estimated, and some of those tasks will be on the critical path. As this happens, you must reduce your remaining contingency buffer time by one day for every day of overrun you experience on critical-path tasks. (Without a contingency buffer, if a task on the critical path slips, then the project as a whole will slip by the same amount.) Keep an eye on the balance remaining in your contingency buffer as part of project tracking. If you are depleting your buffer more rapidly than your team is completing the planned work, then you may need to renegotiate your plan with stakeholders.

Your team members should aim to meet the schedule targets they originally estimated for individual tasks without the contingency buffers -- which are intended as a safety margin for overly optimistic estimates. As a project manager, however, base your commitments to stakeholders on the estimated schedule for all project tasks plus the contingency buffers. That is, work internally to the nominal estimates, which will lead to your planned delivery
date. But include the buffers in your committed delivery date. And if you don't consume the entire buffer, you'll finish early!

Some contracts base a reward structure on the portion of the buffer that remains unused when the project is completed. When the Rochester, New York, airport repaved its main runway, the contractors received a bonus for every day they came in ahead of the committed schedule. They finished early, they got their bonus, and those of us who lived in the flight path for the secondary runway were delighted. Of course, it's possible that a bonus incentive would lead a vendor to pad a bid with an excessive contingency buffer, which is why clients should prepare their own estimates as a reality check.

**How Big Is Your Buffer?**

As a general guideline, the total project schedule reserve should be 5 to 10 percent of the sum of the estimated duration of all project activities (not 5 to 10 percent of the overall project schedule; that's not enough). The higher number is appropriate for especially complex projects or those that involve unusual risks, uncertain or churning requirements, or bleeding-edge technologies.

To estimate an appropriate contingency buffer for your project, first identify areas of uncertainty that could lead to poor estimates. These might be the most technically challenging or innovative parts of the project, tasks that use unfamiliar techniques or technologies, or poorly defined requirements. Once you've identified the major sources of uncertainty, set an appropriate safety margin (perhaps 25 percent) for the time you allocate to corresponding tasks.

Next, examine your past projects for recurring patterns that led to delays, such as chronic requirements growth, overlooked tasks, or consistently overly optimistic estimates. One company came to me for help because they were consistently overrun by at least 25 percent. When we looked at the paper trail, we saw that their projects experienced an average scope creep of about 25 percent. It didn't take a rocket scientist to figure out that they should include a 25 percent contingency buffer in their project schedules to accommodate requirements growth.

Finally, use risk analysis to estimate the possible schedule impact if any of the serious risks (known factors that could pose a threat) materialize or if external dependencies are not satisfied. Multiplying the estimated probability that the risk could become an actual problem by the potential schedule impact if it does become a problem yields a risk exposure in units of time. You should factor the risk exposures from your top ten or so risks into your contingency buffers.

For example, suppose you plan to subcontract a critical component of your next project to a specific vendor. The last four times you used this vendor, they delivered on schedule twice and were four weeks late twice. Therefore, your best estimate of the vendor's future performance (assuming that the vendor is doing similar work for each project) is that they have a 50 percent probability of being four weeks late, for a risk exposure of two weeks.

You should also reconsider any remaining contingency buffers at the end of
each phase or major milestone. Suppose you based your contingency in part on a possible late delivery for the component this vendor supplies. Once you receive the component, you can perhaps shrink the contingency buffer for a future phase, because that risk can no longer delay the project. Alternatively, you might want to increase a future buffer if you identify new risks, if the potential exposure from a specific risk increases, if a dependency fails, or if previous buffers turn out to be too small.

Your risk analysis will also help you sell managers and customers on the need for contingency buffers. I know of a telecommunications company that incorporates a risk analysis into its standard estimation process for that very reason.

**Critical Chain Project Management**

As we already noted, Option 1 above (add a "safety increment" to every task) is not a good approach because there's a risk that Parkinson's Law will prevail. In addition, you would have to increase the estimates by a surprisingly large amount to substantially increase the chance of meeting your schedule.

As Figure 1 illustrates, the probability distribution for an estimate is not symmetrical: A long tail extends to the right along the time axis. That is because many factors and events could potentially delay the task completion, but the most damaging events have a low probability of occurrence. However, a minimum time is required to complete the task even under the best of circumstances, so the probability distribution is skewed to the right. The area under the curve from the left edge to a given time indicates the probability that the task will be completed by that time. Adding enough safety time to move from, say, the 50 percent probable estimate to the 90 percent confidence level can double the duration. Including a safety factor in each task therefore greatly extends the schedule, although it does increase your chances of success.

![Figure 1: Probability Distribution for a Task's Estimated Duration](image)

Note the long tail extending to the right. To move from a 50 percent probability of being done by a specific date to a 90 percent probability, you might have to double the estimated task duration.
Critical chain project management offers a more sophisticated approach to building contingency buffers.\textsuperscript{9,10} Unlike the task-oriented critical path concept, the critical chain considers both task and resource dependencies. The critical chain of tasks is the longest sequence of tasks that leads to project completion, after all resource conflicts have been resolved. In critical chain planning, you estimate each task at the 50 percent confidence level, which means that about half of the tasks will be completed early and about half will be late. In addition, you remove the safety time from individual tasks, which encourages team members to strive to meet the stated estimates.

The schedule safety net is provided through \textit{feeding buffers} included at the end of major activity sequences along the critical chain of project tasks and an additional \textit{project buffer} placed at the end of the critical chain (Figure 2). To calculate the buffer sizes, first estimate the safety time that would increase the estimate probability from 50 percent to 90 percent for each task in the critical chain. Then, take the square root of the sum of the squares of these individual tasks.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{critical-chain-diagram.png}
\caption{The Critical Chain Approach. Feeding buffers and a project buffer are treated as tasks in the schedule.}
\end{figure}

This results in a buffer considerably smaller than one you would get by simply totaling the individual safety times for the critical chain tasks. However, the project schedule buffer should be at least 25 percent of your critical chain duration.\textsuperscript{12} The safety buffers in critical chain project management represent a pooling of risk -- the risk of not completing tasks on time -- as a technique for dealing with tasks that do slip. Studies suggest that using critical chain contingency buffers for planning can substantially improve a project’s schedule performance.\textsuperscript{13}

\section*{Your Turn}
To start incorporating contingency buffers into your project plans, try the following steps:

1. Identify past projects and situations in which contingency buffers would have been helpful.
2. Select a method (e.g., Critical Chain Project Management) that will help you estimate appropriate contingency buffers on your projects.
3. Use historical records in combination with the calculations that result from applying the method you selected to persuade your managers and other stakeholders to accept your buffers.

As my colleague Michael Green says, "It's not my estimating process that's weak. It's my chronic inability to see the invisible, know the unknowable, and predict the unpredictable that keeps throwing me off." Given the many sources of uncertainty on software projects, it's wise to use contingency buffers to protect your team from undue stress and help them meet their commitments.

**Acknowledgments**

The author appreciates the contributions made to this article by Chris Fahlbusch, Ellen Gottesdiener, Michael Green, Karen King, Kathy Rhode, and Johanna Rothman.

**Notes**


For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
Mentor Capital: Growing a Company's Most Precious Resource

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

Mentors can provide a number of valuable functions within any business group: role model, teacher, listener, coach, and all-around supporter. Mentoring can also assume many different forms, but fundamentally, a mentor's job is to enable others by sharing experience and knowledge. It is widely acknowledged that an organization's ability to mentor or coach others is a corporate asset, so logically, we would expect most companies to make teaching others an integral part of their culture. If they invest working capital (cash, for example) to maximize its worth and help drive their business, why wouldn't they do the same with their people?

Unfortunately, too often organizations either overlook or just plain ignore this critical aspect of corporate investment, which is a huge mistake. In fact, some promote the opposite of mentoring: By placing too much emphasis on looking good in front of the boss as well as constant demands on employees to prove themselves in order to preserve their jobs, they encourage people to hoard information and skills so others can't "steal" them. This is shortsighted; without "mentor capital," an organization has a dim future. The reasons for this are quite simple: Unless everyone is on the same page, executing against the same goals and owning responsibility for their actions, teams can quickly become fragmented and internally focused. This is a characteristic of a low-trust environment, in which people feel the need to protect and preserve their own interests above those of the team.

For software development organizations, mentor capital is especially crucial because they typically cannot take advantage of "economies of scale." Most software projects, in fact, suffer from a "diseconomy of scale," which means they can't achieve improvement (in performance,
quality, schedule, etc.) simply by applying more resources to the problem. Therefore, they have a great need to leverage the improvements mentors can help bring about -- in skill sets, communication and cohesion, data sharing, and focus -- if they want to improve software development performance overall.

This article addresses the basic requirements for growing "mentor capital" and providing a sound foundation for an organization's future. It explores the fundamentals of mentoring, discusses the benefits, and suggests criteria for good mentors.

Building a Mentoring Environment

To understand what sort of environment is required for effective mentoring, think for a moment about the world of sports. Can you recall a single instance in which a coach refused to advise his team during a game? Refused to hold a practice? Neglected to give timely and honest feedback to his players (just think Woody Hayes of Ohio State!)? I certainly can't. In that world, coaches know that the team can't deliver what is expected of them unless the coach, as leader, helps them along by sharing knowledge, providing moral support, and paying attention to what motivates his players. Also, coaches know that they need to build leaders within the team -- players in the trenches who can motivate their peers to achieve.

So what can we do to cultivate that kind of understanding and behavior in the business world, specifically within the world of software development?

There are several ways to implement a mentoring environment within your organization, and all must start at the top. If an organization's leaders don't value mentoring, then neither will the rest of the team members. Although we don't have an official mentoring program at Rational, our field guidelines and core values demand knowledge sharing and leadership development on the part of every manager throughout the organization. An environment like ours is a prerequisite for a more "official" program, too. There will be no guarantee of involvement or success unless you first create a corporate culture that makes people feel that mentoring is the right thing to do. If your company already has core values that encourage mentoring, then a well-conceived program might strengthen good practices that are already in place and create a more effective mentoring environment.

To establish a mentoring program, start with the end in mind. Do you want to create an environment that grooms future managers, or one that improves skills, confidence, and ability within people's current positions? Also, keep in mind that personnel in sales, product development, and consulting have different needs, desires, skills, and career paths, so the program should be designed to fit specific career fields. Trying to impose a "one size fits all" mentoring program on an entire organization typically dilutes the initiative's effectiveness and value. And it's important to anticipate the needs of those who want to cross disciplines: engineers who want to be in sales, product managers who want to be field consultants, and so on. For those engineers, for example, it would make sense to
assign a mentor who works with salespeople rather than match them with another engineer.

One approach I use to strengthen the mentoring activities in my organization is to include them as a measure of both individual and team performance. When I do my teams' annual performance review, I spend a good part of the time discussing what knowledge transfer, skills improvement, and general corporate citizenship activities took place during the review period. I find that if I do this consistently, then both leaders and team members begin to look at situations with an added dimension: How to help others be successful and reach their goals.

**Mentoring Fundamentals**

No matter what mentoring approach you decide to use, you'll want to keep a few fundamentals in mind. In general, good mentors do the following:

- **Facilitate thinking instead of giving the answer.** Good mentors make you think, form your own conclusion, and then execute. They see themselves as facilitators and drivers.

- **Encourage teams to take risks.** Mentors push teams to experiment with new approaches and to rely on their instincts and experience to move in new directions -- as opposed to always doing things the same way, or relying too heavily on numbers to drive a decision.

- **Allow themselves to be vulnerable.** Leaders who don't put up a shield that protects them from making a mistake (or from simply revealing that they are human and don't know everything) make it easier for their teams to build trust up the chain of command.

- **Make team members feel responsible for everyone's success.** A goal of any mentoring program should be to make many people, and not just the manager, feel responsible for every team member's individual success as well as team success overall. This will give you a huge increase in the amount of support and the number of ideas at your disposal to improve performance. You'll also get better cohesion across the team.

**Benefits of a Mentoring Environment**

According to Noel Tichy, author of *The Leadership Engine*, all effective leaders strive to create mentoring environments within their organizations and teach others. There are many obvious reasons why this is advantageous: It increases an organization's capability, helps get the work done more efficiently, and so on. But what about the reasons that aren't so intuitive? In truth, these provide just as much or more value. Let me offer the following list.

**Promotes Creativity and Risk-Taking**

In cultures in which trust is lacking, people go into "protective mode"; they grow risk averse and act only to ensure self-preservation. In
contrast, when people feel they can trust others to give them honest and sincere feedback for the right reasons (i.e., to benefit the company as opposed to forwarding a selfish, personal agenda), they are more willing to open up, take risks, and forge ahead. In a software development organization in which communication and information sharing is critical for success, having an environment that breaks down territorial boundaries between and within teams can only improve performance and productivity. Development organizations that foster open communication through mentoring can enjoy the benefit of the whole team's collective brainpower and effort.

**Smoothes Out the Dips**

Having a culture that promotes open communication and trust also helps teams have a better chance of weathering the tough times and accelerating through the good ones. Good mentoring can produce an organization consisting of people who are on the same page, capable of making the right decisions without being micromanaged, and able to execute both within a team and as individuals. This gives you a powerful foundation for getting through rough spots and persevering. Although ultimately it takes much more than trust and communication to achieve goals, having this foundation for the team is a must.

In addition, when challenging situations require everyone's effort, it's better to have a team of problem solvers rather than a collection of order takers. Mentoring can help ensure that team members know how to leverage their collective brainpower and work together to solve problems. They'll be far more likely to succeed than a team that is unable to think innovatively and unwilling to take chances.

**Attracts the Right People**

Winners like to be with winners. If you create an environment with a high level of trust and a high sense of mission and purpose, and if you give people the opportunity not only to reap financial rewards but also to grow as individuals, then you will attract people who appreciate and respond to those opportunities. Typically, people who have a bad case of the "What have you done for me lately" syndrome -- who are interested only in their own good and how much money they can make -- are not attracted to mentoring environments that require a lot of give and take.

You'll also be able to attract good people because a trusting and mentoring environment provides an infrastructure that can help all team members succeed. If a prospective employee knows that your software organization takes on entry level personnel or people with skills that need to be developed, and that the entire team is vested in their success (and ultimately the team's success) and development, then he or she will feel totally confident about joining you. At the same time, new team members will know that they won't be able to "just get by," and that they must pull their own weight in helping to achieve the team's goals. Mentoring environments usually attract high performers to whom results and professional development are important.

People who believe in the corporate mission and like to work
collaboratively are the people who help companies sustain performance through good times and bad. And that brings me to my next point.

Ensures Long-Term Survival

In *Good to Great,* Jim Collins says that companies who have charismatic leaders do very well while that leader is running the show but begin to stumble after the leader goes. The reason? Charismatic leaders often force their people to do what the leader believes is right instead of what is actually right for the company. In other words, people focus on making the leader happy even if it means doing the wrong thing for the company. Also, Collins found, charismatic leaders make very poor mentors; they are wrapped up in their own world and have little interest in those around them.

Here's an interesting hypothetical question for all you managers out there: If your organization continues to execute very well while you are away on vacation, do you feel sad that they could do without you, or proud that they could do without you? Likewise, if they stumble when you go on vacation, do you stick out your chest and proclaim, "They can't do it without me!"?

If you fall into the camp of feeling sad when they succeed without you or comforted when they don't, then that's a sign of trouble. Chances are there is not enough mentoring going on within your organization, and people depend on you to make all the decisions. Ultimately, if you do not groom new leaders, then the failures will increase -- and you will look bad, too.

Remember: Good managers teach their teams how to fish instead of just giving them the fish.

Who Should Mentor?

Mentors don't have to be managers. I remember reading a case study for a management course years ago that focused on discovering who the real "influencers" were in an office. Over a period of several months, the researchers tracked how many visits each person in the office received during the day from coworkers and plotted the results on the office seating plan blueprint. Without seeing this plot, you might assume that managers received the most visits over time, but this was not the case: Non-managers received far more visits than managers. The centers of power and influence in an organization do not necessarily reside within the management structure, this study found. When people come seeking advice and wisdom, it doesn't matter where you sit, how much money you make, or how fancy your title is. What matters most is a willingness to listen, give advice, provide feedback, and take a genuine interest in a coworker's situation.

These qualities are key, whether your organization has a formal mentoring program or not. If the environment encourages it, then more people will work on developing good mentoring skills.
Generally, effective mentors do the following:

**Share a Teachable Point of View.** Not all great athletes make great coaches, and the same can be said about knowledgeable, experienced employees vis à vis mentoring. Of course, it's important for mentors to have a broad spectrum of experience and interests so they can advise on multiple areas and topics. But what really makes for an effective mentor is the ability to translate those experiences into a form that is useful to others. That means being able to truly understand the core issues involved in another person's work so you can shape your lessons to address those issues. It also requires the ability to analyze that person's strengths and weaknesses so you can support the former and help him or her overcome the latter. For software development organizations, it helps if the mentor has been involved in failed, as well as successful, projects in several capacities such as project manager, lead developer, or analyst, for example. It is also a benefit if the mentor has worked on various types of projects with different kinds of drivers and constraints. All of this variety leads to a well-rounded body of knowledge and experience one can draw from.

**Provide Honest and Direct Feedback.** Some people avoid giving honest feedback because they want to avoid confrontation. To be of any help, however, a mentor must be able to tell people what they're doing wrong and provide constructive suggestions about how to correct it. Typically, this is a skill that people acquire over time, but some people never really become adept at it. If you plan to mentor, it helps to remember all the reasons why this feedback is important: to help someone see things in a different light and improve, and ultimately to contribute to the good of the organization.

**Exercise Patience and Discretion.** The last thing anyone wants or needs when they're in a tough situation is to have someone broadcasting their issues throughout the organization. If there is going to be trust, then there must also be discretion. Mentors need to make sure they keep things to themselves and do not violate the trust of people they are advising. Being patient is also a key requirement when you take on the task of helping others. That includes allowing others to discover the answers instead of just blurting them out yourself. The discovery process is key to both learning and to developing the motivation to take corrective action.

**Understand the Organization and How to Navigate.** Any situation involving a number of people includes dynamics that need to be understood and appreciated: Where does the power reside in the organization? What is driving the customer? Where are the icebergs and landmines? Why is a certain tool being chosen over another? Mentors should be able to play an important role in helping teams navigate through organizational issues, especially in software development, where lots of time, money, and resources are usually being consumed in order to deliver a system. It takes a wise and experienced person to recognize that people and organizations are not always driven by the obvious demands of quality, schedule, and so on; they might be trying to comply with secondary factors such as company culture, fear of change/failure, and risk mitigation.
Are You a Candidate for Mentoring?

Just as it takes a certain kind of person to mentor, not everyone is a prime candidate for being mentored. When you link up with a coach, you must be willing to open yourself to constructive feedback, face reality, and attack the hard stuff. On the plus side, this can lead to achieving your goals and making remarkable improvements.

Of course, the first step in this process is recognizing that you need help or guidance. A few years ago, I hired a personal coach to help me smooth out a few of my "rough" spots -- to listen better, communicate more effectively, cultivate patience, and address a few other areas that needed attention. To do this, I had to take a hard look at myself and analyze where I most needed help. Then, I had to be willing to listen to feedback that was sometimes painful. "You're impatient"; "You talk too much"; that's what I heard from my coach at first. But realizing that this process was going to make me a better person kept me focused and determined to go through with it.

The process began with giving concrete examples of the issues, why I felt they were a problem, and what I wanted the end result to be. Then, I formed my own solutions. As I noted above, mentors and coaches are there to guide and assess instead of just giving you the answers. Coming up with a solution is part of the learning process and makes the whole exercise worthwhile. In the end, if you come up with the solutions, and those solutions work, then you will value and appreciate the lessons you learn that much more.

Of course, in the workplace, not all mentoring gets down to such a personal level, but listening to criticism of your work can be just as painful. I have found that to really get the most out of mentoring, you have to be willing to attack the toughest areas first. After that, the rest is much easier. Opening up and letting someone you trust dig, pry, and inject a dose of reality into your world is very liberating. If they're doing it for the right reasons, it can make a critical difference in your career.

Once you're certain that you're ready for a mentor, finding an appropriate person is not always easy or obvious. It helps if you know what you need from a mentoring program: Is it to develop technical skills or people skills? To get help with "reading" the organization or managing people? Knowing where you want to focus will help narrow your search and identify the type of mentor you are looking for.

Sound Investment, High Returns

Mentoring capital is just as important to an organization as its cash reserves. If you invest carefully in creating a strong mentoring environment, then the returns can be very high for mentors, for those they advise, and especially for the organization as a whole. As Jim Collins points out in Good to Great, leaders who are ambitious for their company rather than for themselves seek to develop other leaders who can help drive the company in the right direction. Through mentoring, they build
leaders for generations to come, who can sustain the company's success over many years. In the context of a software development organization, passing on wisdom and best practices to develop leadership capability translates to continuous productivity, higher project success rates, and winning the confidence and support of the company's non-technical managers.

Do you feel your organization would improve with the help of a mentoring program? If you begin exploring the possibility, then you will soon find others who want to participate, and your organization can start realizing substantial benefits very quickly. A sound mentoring program can bring about improvements in capability, performance, communication, and team dynamics. It can also help those who mentor by teaching them how to lead more effectively and giving them a chance to make a positive contribution to the overall good of the organization.

Notes


For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
Involving Your Customers in Your RUP Project

by Gary Pollice
The Rational Unified Process
Rational Software

Recently I visited a Rational customer company that had just delivered the first release of a new product to one of its own customers. The development team had an unfortunate, but all too common, story to tell. In creating the product, they'd had to spend a lot of extra time rewriting code because their customer refused to participate in the project. "We're too busy to think about this," the customer had told them. "That's why we hired you." So the team had resorted to trial and error: They would write some code, show it to their customer, and be told that the system was not what was needed. Then, they would go back and try to figure out what the customer really wanted and begin the cycle all over again. This went on for several cycles, and miraculously, in the end, they had something that was "good enough" for their customer to use. Now, they were about to initiate another development cycle to improve the product and attempt to produce the system their customer wanted in the first place.

Although this team was certainly happy about delivering a product on time, it was clear that their heroic efforts were not sustainable. In fact, the team was burned out. When they asked me: How could we have done things differently? I had an answer: By convincing your customer to collaborate with you in using the Rational Unified Process® (RUP®).

We all know that rework is a fact of life in software development. It happens when the requirements are not completely understood, which is most of the time. Nevertheless, it is possible to cut way back on the amount of rework you have to do by getting your customer to work with you in a systematic way and ensure that you're on the right track. As a step toward accomplishing this, below I will share my ideas about the various roles customers should play during the development process.

People, Process, and Tools
Let's begin by looking at the RUP itself. The RUP is a process framework that provides a wealth of knowledge on how to apply industry-proven best practices to software development projects. It is part of Rational's three-part approach to customer success, which includes:

- **People.** The most important component of this approach, people are what can either make or break a project. The best-run projects enable people to work in an optimal, sustainable manner. Rational provides process, Rational support people, and tools to assist all the people on a development project team, to help them jump-start their efforts and become more efficient.

- **Process.** The RUP is a role-based process framework that provides guidance for all members of a software development project. It also supplies tool-specific guidance for practitioners who use Rational tools or tools from one of our partners.

- **Tools.** The Rational family of software development tools supports best practices described in the RUP and, by extension, the people who use them. The tools are an integrated set of products that enhance a software team's communication and overall effectiveness, enabling them to produce better software in a more predictable manner.

Underlying the RUP is an assumption that the key mechanism for enabling people to work together better is to improve their communication. Ensuring that all people understand their responsibilities and how they relate to those of others on the project is a second important mechanism. This goes for all people involved in the project -- not just the development team.

**Customer Responsibilities**

In addition to discussing the generic "Stakeholder" and "Any Role," which may refer to customers as well as other team members and people in the organization, we will discuss the activities and responsibilities of the following roles (see Table 1 for a summary):

A good customer is an involved customer. Your first priority is to build a relationship with your customers so they will work with you in a cooperative manner, rather than in an adversarial fashion. Take time to help your customer understand how important it is for them to participate in the project. Review the responsibilities, activities, and artifacts for which they might be responsible, and get agreement that they will deliver on them. If they balk at anything, then negotiate a compromise and offer your help; it pays to help them help you succeed. And make sure they understand that this is a win-win situation: When you succeed, they will succeed.

Now, let's look at what a good customer is actually responsible for in a typical software development project. We will approach this systematically by looking at the roles defined in the RUP and determining which part of
each role's responsibilities could be assumed by a customer. In addition to discussing the generic "Stakeholder" and "Any Role," which may refer to customers as well as other team members and people in the organization, we will discuss the activities and responsibilities of the following roles:

- Business-Model Reviewer
- Requirements Reviewer
- Design Reviewer
- Tester
- Test Analyst
- Change Control Manager
- Project Manager
- Project Reviewer

**Business-Model Reviewer**

Not all projects involve business modeling. When they do, however, the customer is the one who knows best how the business operates. However, not all customers are able to articulate this and effectively communicate it to the development team. If you are using a Business Designer to formally analyze and design new business processes, then the customer's responsibility is to help the Business Designer understand the specific business context and review the Business Designer's output for accuracy. In this capacity, the customer assumes the responsibility associated with the role of Business-Model Reviewer. The associated activity that the customer is most likely to perform is Review the Business Use-Case Model. Although the customer should not be the only one reviewing the model, doing a review without customer participation would be much less effective.

The customer is also a good candidate to participate in the Business Object Modeling Workshop. This is not mentioned in the Business-Model Reviewer role, but is a part of the Find Business Workers and Entities activity assigned to the Business Designer. If the customer participates in the workshop, then the development team has more information and background for performing review activities.

**Requirements Reviewer**

Of course, the customer should review the requirements! Many people would say this is the most important thing a customer should do on any project. Yet, I'm constantly amazed at how many times I encounter situations in which the customer is not involved in requirements review. Requirements, regardless of how you communicate them, are the primary instrument for ensuring that you will build the right system. Any communication -- even if you are working full-time and on-site with your customer -- can be misunderstood. If you have a remote customer or many customers, then you must write down, review, and revisit requirements continuously throughout the project, and make sure your...
customer is on board for each review.

The RUP has several requirements artifacts: The Vision, Stakeholder Requests, and Supplementary Specifications are document based. Use cases are the preferred vehicle for communicating functional requirements. And prototypes -- not just user-interface prototypes -- are excellent vehicles for communicating requirements. Allow time to walk your customers through your prototypes. This lets them know that you understand their requirements, and it gives you a chance to explain exactly what your solution will do for them.

Most customers freely assume the Any Role activity of submitting change requests; but then they also need to take the Requirements Reviewer responsibility seriously. Change requests effectively become requirements, so they need to be prioritized and planned for.

**Design Reviewer**

Normally, the customer will not be involved in reviewing the design. The customer's main concern is to get software that meets the requirements -- most need not see (and may not understand) the underlying design that will accomplish this. However, if your customer happens to be a software development organization, then they will care about the system design. Perhaps you are delivering a component for a specific system or a reusable library of components. You might be designing software that has built-in extensibility mechanisms. In these cases, your customers should participate in design reviews. They might not contribute input that affects the final design, but at least they will know early in the game what they need to know and do in order to use the software you plan deliver.

**Tester**

It might seem strange to some customers when you tell them that they should be testing the software. Although customers should not have to run unit tests, they should assume responsibility for designing and performing acceptance tests. The software belongs to the customers. They are the ones who have to accept it. They should formulate acceptance criteria carefully, including a stipulation that customer personnel are able to run the system. Customers who accept software based on seeing correct output only are missing a big part of the picture. It is important that customers work side-by-side with the development team to make sure the software meets all their acceptance criteria.

**Test Analyst**

In the Test discipline for RUP v2002, there is a new Test Analyst role responsible for identifying what will be tested based upon various plans. Customers need not perform all of the activities for this role, but they must be involved in determining appropriate test targets as they relate to system acceptance. Customers should also perform the Define Test Details activity for the acceptance tests.

**Change Control Manager**
Customers should not be the primary Change Control Manager, although they should participate in activities such as Establish Change Control Process and Review Change Request.

Change requests will come from many sources other than the customer, which is why the activity is assigned to Any Role. Customers, however, should have a strong voice in determining which change requests are accepted and how important they are.

There are several ways a team can review and prioritize changes. The Change Control Manager is responsible for establishing the change control process, but the customer should be involved in determining the appropriate change control mechanism for the project. Having an agreed upon change control process is especially important when the development team is working on a fixed-price contract or when there is very tight budget control. Naturally, customers will always want as much as they can get, so there needs to be a way of agreeing upon the impact and cost of changes. Time spent striking an agreement on this process up front will pay great dividends during the project lifecycle.

**Project Manager**

Usually, a person from the development organization or a program management group runs a project, but sometimes the customer fills that role. However the project is organized, customer involvement is essential to the project's success. Without a customer, there is no project. The customer should be very visible in four project management activities:

- **Identify and Assess Risks.** There are many types of risk on a project. Some are technical, and some are business risks. The customer can best identify and rank the business risks, and they are often the ones who need to determine appropriate mitigation strategy for business risks.

- **Develop a Product Acceptance Plan.** As we've already noted, from a management viewpoint, the customer has to accept the system and is in the best position to determine what constitutes an acceptable system. Although customers may not author the Product Acceptance Plan, such a plan without customer representation is a useless document.

- **Develop the Business Case.** The RUP says: "The Business Case provides the necessary information from a business standpoint to determine whether or not this project is worth investing in." Unless you are building a software product for resale and are looking for investors, the customer should make the business case for the new system. In most IT organizations, the customer, not the development team, is responsible for justifying the cost of a new system.

- **Iteration Planning and Assessment.** The customer should play an important role in two separate RUP activities for planning and assessing iterations. When planning an iteration, the customer helps adjust priorities and determines what goes into an
iteration. When assessing an iteration, the customer helps decide if the iteration goals were met from the customer's viewpoint.

Project Reviewer

The RUP says: "The Project Reviewer role is responsible for evaluating project planning artifacts and project assessment artifacts at major review points in the project's lifecycle. These are significant review events because they mark points at which the project may be cancelled if planning is inadequate or if progress is unacceptably poor." As you can see, this role encompasses much of the responsibility we described in the previous section, although we acknowledged that customers do more than just review; they actually drive much of the project. In any case, clearly the customer must be included in any review that affects the final software deliverable.

Generic Roles: Stakeholder and Any Role

In the RUP, a Stakeholder represents "anyone who is materially affected by the outcome of the project." Although the Stakeholder has no defined responsibilities, in reality responsible Stakeholders will participate in a project appropriately for the stake they have in it. A Stakeholder with a significant investment will usually participate more and have greater clout in the project. All Stakeholders should at least participate in major milestone reviews.

Any Role does have assigned responsibilities and activities, but everyone knows the customer will want to change the system while it is being built -- and probably after it has been delivered. Therefore, the customer should actively submit change requests by taking on the Any Role responsibility. As noted above, I am not suggesting that the customer is the only one who will take on that responsibility, merely that a good customer will accept this role when it is appropriate.

Table 1 summarizes the RUP-based roles and activities we described in this section. The "Comments" column also notes what can go wrong if customers are not involved in the activity. Although your customers might assume other roles and responsibilities during the course of a project,
these provide a good starting point. Chances are that some of them will not apply to your project; as always, you need to adapt the process to your situation.

Table 1: Roles for Customers of RUP-Based Projects

<table>
<thead>
<tr>
<th>RUP Role</th>
<th>Activities</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any Role</td>
<td>Submit Change Request.</td>
<td>Customer should submit change requests that reflect their business needs.</td>
</tr>
<tr>
<td><strong>Business-Model</strong></td>
<td><strong>Reviewer</strong></td>
<td>Review the <em>Business Use-Case Model</em>. Some customers may also want to participate in a Business Object Modeling Workshop.</td>
</tr>
<tr>
<td>Requirements</td>
<td>Review Requirements.</td>
<td>Unless customer continually reviews the project's requirements, communication may break down, effort will be wasted, and potential for building the wrong system escalates. Make sure the customer reviews all forms of requirements, including prototypes.</td>
</tr>
<tr>
<td>Design Reviewer</td>
<td>Review the Design.</td>
<td>This activity is appropriate only if your project is delivering software for reuse or extension.</td>
</tr>
<tr>
<td>Tester</td>
<td>● Implement Test.</td>
<td>Customer must formulate acceptance criteria and conduct acceptance tests to ensure that the system meets their needs.</td>
</tr>
<tr>
<td></td>
<td>● Execute Test.</td>
<td></td>
</tr>
<tr>
<td>Test Analyst</td>
<td>● Identify Targets of Test.</td>
<td>Customer should help determine appropriate targets for acceptance tests.</td>
</tr>
<tr>
<td></td>
<td>● Identify Test Ideas.</td>
<td></td>
</tr>
<tr>
<td>Role</td>
<td>Activity</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td><strong>Define Test Details.</strong></td>
<td>For acceptance tests, the customer should specify the conditions that must be tested and define the expected results.</td>
</tr>
<tr>
<td><strong>Change Control Manager</strong></td>
<td>Establish Change Control Process.</td>
<td>It is especially important for customers to participate in this activity for fixed-price contracts.</td>
</tr>
<tr>
<td></td>
<td>Review Change Requests.</td>
<td>Customers should be committed to scope management and help prioritize change requests.</td>
</tr>
<tr>
<td><strong>Project Manager</strong></td>
<td>Identify and Assess Risks.</td>
<td>Customer is responsible for business risks.</td>
</tr>
<tr>
<td></td>
<td>Develop <em>Product Acceptance Plan.</em></td>
<td>If customer does not participate in -- or, preferably, drive -- this activity, then there is greater risk that the wrong system will be built and that disagreements between customer and development team will occur during deployment.</td>
</tr>
<tr>
<td></td>
<td>Develop <em>Business Case.</em></td>
<td>Customers know the business problems and the value of a solution. Use that knowledge.</td>
</tr>
<tr>
<td></td>
<td>Develop <em>Iteration Plan.</em></td>
<td>Customer input can help in prioritizing activities and tasks and managing project scope.</td>
</tr>
<tr>
<td></td>
<td>Assess Iteration.</td>
<td>Represent the customer viewpoint (i.e., business viewpoint) in the iteration assessment.</td>
</tr>
</tbody>
</table>
### Notes

1. Here *customer* refers to anyone who either contracts for the software or will use the completed system.


3. When we say an artifact is document-based, it does not mean that it has to be a large, complex document. We really mean that the artifact has some persistent form, possibly a simple, handwritten document, depending on the formality of the situation.

---

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

---

Copyright [Rational Software](#) 2002 | [Privacy/Legal Information](#)
Book Review

Unlocking the Clubhouse: Women in Computing
by Jane Margolis and Allan Fisher

MIT Press, 2002
Cover Price: US$24.95
172 Pages

The information technology revolution is transforming almost every aspect of society in positive ways, but researchers Jane Margolis and Allan Fisher, authors of Unlocking the Clubhouse: Women in Computing, are still concerned. Girls and women, the authors contend, are largely left out of the loop. They participate in the dominant global industry primarily as guests, by invitation, or by compromising their own values, goals, and ethics.

It is a serious accusation, but one that is borne out through the authors' research. Their studies were based on interviews with more than one hundred computer science students of both genders from Carnegie Mellon University over a period of four years, and coupled with classroom observations and conversations with hundreds of high school and college faculty.

The conclusions? That unless we change the way that we educate our children, there will always be a gender divide in the most important area of 21st century life. And women will always be on the wrong side of that divide, the have-nots in the real power centers of our culture.

Consider the following observations made by Margolis and Fisher:

- Workplace systems are built around male cultural models;
- Entertainment software fulfills primarily male desires;
- Early voice-recognition systems were calibrated only to typical male voices;
- There is a tacit understanding that working in computer science involves giving up a balanced life.

This state of affairs isn't just detrimental to women: the authors make it
clear that by excluding women, the computer culture is losing a healthy and vital source of contributions.

For me, these conclusions seemed obvious. But based on my experience, many men are not even aware of all the ways in which girls feel excluded from various computer-related experiences. After all, if one feels comfortable in a given environment, one doesn't typically wonder if everyone else there feels that way, too -- unless something jolts them out of their ordinary way of thinking. For example, as a Caucasian, I rarely stop to consider what it means to people of color to be consistently underrepresented in the entertainment media. But when I went to see The Lord of the Rings recently and happened to sit behind an African-American couple, suddenly, I became acutely aware that everyone on the screen in front of us was Caucasian -- even Tolkien's imaginary beings. I think that men might have a similar "awakening" as they read this book; it provides an excellent opportunity to view the world of computing through another lens.

In The Beginning...

The authors begin by focusing on the fundamental, ongoing debate about nature versus nurture. Are we the way that we are because gender-related characteristics are encoded in our DNA, or is it because we were taught by our parents, our culture, and our world how to be male and female?

Margolis and Fisher found that "ýas early as kindergarten, girls use the computer skillfully for writing their stories, but boys race to the computers for free time and play." It appears to the authors that, from the moment the computer is first introduced into children's lives, it is seen as a tool by girls and as an end in itself by boys.

And then there is the home environment. "Overall, the women we interviewed had done less hands-on exploration of the computer than the men. They gave fewer accounts of working beside their fathers and more stories of watching from the sidelines. Computing and tinkering had not been their main childhood activities or focus but one interest among several." In fact, the authors go so far as to describe what they call a "father-son internship," pointing to studies showing that fathers play with their sons 50 percent more than with their daughters, with the inevitable unfortunate results.

And the early years are just the beginning. By elementary school, the researchers say, adults -- both parents and teachers -- have developed unconscious expectations about boys' projected success in computer science. And as time goes by, those expectations deepen until they become self-fulfilling prophecies.

Adolescence: The Plot Thickens

At this point, the authors shift the blame almost completely onto the schools, and with apparent good reason. Computer science curricula reflect boys' interests and experience levels, and girls, it seems, may either adapt or leave.
In one school the authors studied, a group of boys developed an exclusive computer "recess club," during which they named the computers used in the school lab. The teacher allowed these names to be used even during computer classes, so that the boys ended up making references to which only they and the teacher were privy. The ability -- the authority -- to name is a clear indication of power and ownership, a point that was not lost on the girls.

In observing another high school, Margolis and Fisher noted that the computer lab became a sanctuary of sorts for "geeks," a place where they could belong. And yet, having been rejected by the appearance-oriented, "cool" majority of students, the geeks were themselves quick to reject others. The book tells of "several incidents when male African American students came into the lab. Usually these students were ignored, stood around, watched awhile, and then left. Neither the other students nor even the teacher invited them to join them at the computers."

Girls fared no better than did minorities.

Girls were perpetually teased about their bodies, their appearance, and their competence. The male teacher did not intervene on behalf of the girls. One of the women students asked the teacher why he always used football examples; he replied that she could do the programming assignments on anything she wanted. At that cue, a male student turned to her and mockingly said, 'Do it on sewing,' which drew laughs from the other students. Another woman student used football statistics in her program (similar to everyone else's program). She was ridiculed because she used the name of a baseball team instead of a football team. Two male students were observed playing Concentration and trying to modify it. One of the new prizes was the services of a prostitute who charged $200. There was no teacher response to these incidents. None of the high school girls enrolled in Computer Science 2 went on to enroll in Computer Science 3.

"Now there's a surprise," I muttered to myself as I read the last sentence.

The authors go on to point out that this locker room atmosphere, coupled with the fact that, as noted earlier, boys get an "internship" with their fathers and spend their childhood playing computer games (whose usually aggressive and sometimes sexist themes are unattractive to girls), leads to the inevitable result that "girls in high school are often sitting shoulder to shoulder in classes with boys who have spent endless hours learning everything they can about computers and who have friends to turn to when they want to learn even more." Add the hormonal sea change of insecurity and fear experienced by girls in adolescence, the authors note, and it's a marvel that any emerge willing to take on computer science at all! Another strike against girls, the authors point out, is a gender difference surrounding the issue of competition. Whereas boys race to find solutions, girls take time out to help others get there as well. Boys want to play games with a clear winner and a clear loser; girls want to play games that leave everyone feeling good. Computer science classes in American
public schools, they found, are geared to reward the former and punish the latter.

"Good grief," I thought to myself after getting through this section of the book. "It's no mystery that there are limited female enrollments in university computer science programs. The wonder may be that there are any at all."

**College Days**

So why do some girls hang in there to study computer science? Here's what Margolis and Fisher report:

> We have found that women decide to major in computer science based on a broad set of criteria (ý). For many male students, in contrast, the decision to major in computer science barely reaches the level of conscious consideration; it is a natural extension of their lifelong passion for computing.

That broad set of criteria included, in many cases, a care for making positive contributions. One first-year student at Carnegie Mellon, Louise, describes a lecture:

> Everyone just said how boring it was: 'who cares that computers did not benefit anyone? We like computers! We love computers! We know computers! And who cares about the rest of the world?'ý. And if you're trying to make something that's going to change the world, that's going to help the world, you have to have some sort of concern about what's your long-term goal. Not just to produce Word 8ý or Excelý whatever. How is this helping? Or is it helping? Go see if that stuff is doing anything.

Sound familiar? Remember how the researchers found that in kindergarten, girls used computers to do or create something else, while boys used them simply for the act of using them? Louise clearly reflects that same theme: she wants to use the computer to make the world a better place, rather than to simply make a new version of Excel.

In fact, Margolis and Fisher heard many variations on this theme, which led them to conclude their discussion of educational issues with some suggestions for the future:

> Can a creative person, a 'people person,' care about the world and people and be happy in computer science? While the stereotype says no, a broader vision of what the field is and how it is best taught answers in the affirmative. Computing can be taught in an interdisciplinary setting, honoring the goal of 'solving the world's problems.' Furthermore, this does not require devaluing the single-minded pursuit of technical virtuosity that marks some of the best computer science students. Instead, it establishes multiple standards of excellenceý
What they are urging the powers that be in our education system to consider, in other words, is that there is never just one way to do things -- as programmers, of all people, surely should know.

**Dreaming in Code**

One of the most interesting -- and to me discouraging -- parts of the book is a discussion of the dramatic differences in the way that men and women regard and *use* computers.

When we asked students, during their first interview, to describe their computer science peers, both men and women responded with the same image. They described a person in love with computers, myopically focused on them to the neglect of all else, living and breathing the world of computing, 'at the computer 24/7.'

This "geek mythology," as the authors call it, has historical/cultural roots in the genesis of computer technology. They cite Steven Levy's book *Hackers: Heroes of the Computer Revolution*, which describes the founding fathers' lifestyles; Stephen Segaller's *Nerds 2.0.1: A Brief History of the Internet* provides a similar take. One can obviously understand the fierce commitment of these early pioneers and perhaps even envy the opportunity they had to be on the cutting edge of something entirely new and exciting, their drive to do more, discover more, create more. Unfortunately, however, as the researchers point out, all of this engendered a culture in which programmers continue to treat their work that same way now, even without the stimulus of being "the first ever" to do something. It has become a habit.

And what we have come to call "geek culture," they add, is essentially and unquestionably male. It is not, by and large, a woman's style of working.

One-third of the male students we've interviewed say they differ from the stereotype, that they have a broader range of interests than just computing. But twice as many women (more than two-thirds of those we interviewed) feel different from the stereotype. And 20 percent of the women we interviewed question whether they belong in computer science because they feel they do not share the same intensity in focus and interest that they see in their male peers.

Donna, a junior at Carnegie Mellon, describes questioning whether she belongs in computer science: "In my free time I prefer to read a good fiction book or learn how to do photography or something different, whereas that's their hobby, it's their work, it's their one goal. I'm just not like that at all; I don't dream in code like they do."

Neither does my friend Karen, although she has been working in the high-tech arena for years. Donna's account reminded me that early in her career, that sense of not "fitting in" almost drove Karen, now a tech support manager, to become part of the "leaky pipeline," which sees women leaving the computer sciences each year, changing their majors,
quitting their jobs, deciding that it is not for them. They are not leaving because women are genetically predisposed to giving up, but rather due to discouraging experiences with teachers, peers, curricula, and workplaces that cause them to doubt themselves and their fitness to pursue computing. In college, it becomes clear to many women that what they are experiencing there is but the tip of the iceberg.

So all you guys out there, turn now to any woman engineer you know and express respect and admiration for what she has survived. She has earned it!

**To Boldly Go Where No Man Has Gone Before**

In the last section of *Unlocking the Clubhouse*, the authors examine the tremendous societal implications of the absence of women in computer science. Just as the originators of *Star Trek* didn't see anything wrong with using *man* as a generic term, so too do we not notice that there is more to this issue than our easy assumptions that girls just don't like math and science and computers as much as boys do.

The absence of women in computer science is, as Margolis and Fisher point out, a social justice issue. Just as we are starting to understand the implications of the digital divide between those who have access to technology and those who do not, so too do we need to understand the implications of the hierarchy that exists within the technological framework and acknowledge that there is a clear gender divide there.

The authors suggest that we need to begin "changing the conversation in computer science." To challenge parents and teachers to reinvent the view of computing presented to children of both genders. To urge industry leaders to compete to enlist the talents of women as well as men in innovation and design.

**Conclusions**

My personal reactions to this book were overall very positive. First, I felt that the study the authors conducted was well done and long overdue. A cursory examination of Bowker's *Books in Print* revealed surprisingly few other books about women in computing. As a woman, I resonated with the authors' conclusions and identified with their examples. I found myself alternately shocked and energized by their findings, and my interest in the topic has been tremendously stimulated by reading this book.

If the book has a weakness, it is perhaps that it makes assumptions about the level of sensitivity among readers. When talking about it with male friends and colleagues, I encountered a surprising amount of resistance to the authors' findings and a reluctance to consider their premises. "So what -- maybe geeks are happy being single-minded about stuff!" was not an atypical response. The book might have been richer and able to reach more people if the authors had explained more about why this male/geek culture is in fact unhealthy, and why the presence of women in the computing workplace would make it a healthier environment.
A little-known song by Harry Chapin asks the question, "Why do little girls grow crooked, while little boys grow tall?"

Why were the little girls all frightened  
To be just what they are?  
The boys were told to ask themselves  
How high, how far  
The girls were told to reach the shelves  
While the boys were reaching stars  
That's why little girls were frightened  
To be just what they are.

The hope inherent in Unlocking the Clubhouse: Women in Computing is that some day, if we keep working at it, the little girls won't be frightened anymore.

-Jeannette Angell Cézanne  
Technical Writer  
Rational Software  

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

UML Components: A Simple Process for Specifying Component-Based Software
by John Cheeseman and John Daniels

Addison-Wesley, 2001
ISBN: 0-201-70851-5
Cover Price: US$29.95
208 Pages

This book provides a lot of in-depth, practical guidance for specifying component-based systems, while remaining compact and readable. Although Cheeseman and Daniels have developed their own unique approach to business modeling, analysis, design, and implementation, they were influenced by many sources, including the Rational Unified Process® (RUP®), Unified Modeling Language (UML), and Catalysis (see References). For the purposes of this review, I will be covering only the design and implementation aspects of their approach.

Cheeseman and Daniels focus on Component Specification, which is a detailed contract that any Component Implementation must fulfill in order to function correctly within the component framework.

To providing sufficiently detailed component specifications, they introduce their own set of UML stereotypes and modeling conventions. Table 1 describes some of these stereotypes (all applied to the UML Class).

Table 1: Cheeseman and Daniels UML Stereotypes

<table>
<thead>
<tr>
<th>Stereotype (of UML Class)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component Specification</td>
<td>● Specifies a component that can be plugged into a component framework, such as EJB or COM+.</td>
</tr>
</tbody>
</table>
| **Subcomponent** | • Similar to a component, but not necessarily separately replaceable.  
• Belongs to a component that *is* replaceable. |
|-------------------|------------------------------------------------------------------------------------------------|
| **Data Type**     | • A structured data type.  
• Has attributes but not operations. |
| **Interface Type**| • Essentially a UML Interface, with greater semantics borrowed from the UML Class; for example, associations with Information Types specify state data. |
| **Information Type** | • Describes the state underlying an Interface Type.  
• Can be thought of as classes, with a clear distinction that the classes represent specification of internal state, not internal design. |

I found this taxonomy clear and quite useful. In particular, I liked the clear distinction between the following kinds of classes:

- Simple data structures.
- Specification of internal state for a component.
- Internal design classes.

In this approach, Component Specifications are more than just interface descriptions. They define detailed component behavior.

- **The component state is represented by "information types."** Information types have attributes and associations with other information types.

- **Preconditions and postconditions on operations are used to specify behavior.** The preconditions and postconditions are optionally described in Object Constraint Language (OCL), and may reference information types. (Otherwise, information types are not visible across interfaces.)

- **Component collaborations also specify the behavior of a component.** Component collaborations are similar to collaborations as described in the RUP and UML, but the objects are instances of components. The "role" is the name of the interface. These collaborations represent specification of component behavior and
interaction, not internal design and/or implementation.

So why is Component Specification important?

- **A Component Specification is typically higher level than the Component Implementation, and is therefore easier to understand.** However, unlike other kinds of "high-level design," a specification is complete -- it contains all the information that a client of the component needs to know. There is no need for clients to delve into a lower level of detail before using the component.

- **A Component Specification makes it easier to buy, sell, and replace components** -- if the component fulfils its *contract*, it should function correctly in the system.

Unfortunately, Cheeseman and Daniel's process stops with the Component Specification. This is a major drawback. Among other things, the reader is left wondering:

- How does one translate the Component Specification into a Component Implementation?

- How does one verify that the Component Implementation complies with the Component Specification?

- What are the overheads of maintaining both a specification model and an implementation model? Practically speaking, the detailed design of a component may be "good enough" in many environments and for many classes of components, especially those intended for assembly into a single system. One consultant I spoke with complained that creating and maintaining Cheeseman/Daniels-style specifications was a significant cost and effort driver on his project, and he was unconvinced of the benefits.

Another drawback of the Cheeseman and Daniels approach is that it does not stay comfortably within the bounds of standard UML. Practitioners may wish to adapt many of their ideas to fit within existing UML-based modeling tools and process descriptions; however, the divergence from standard UML may make this quite challenging.

Nevertheless, Cheeseman and Daniels make a strong case for the benefits of component specifications, especially when assembling systems from preexisting components, or developing components for assembly into multiple systems. Even with its limitations, their book contains important, practical advice for developers riding this new wave of software development.

---

**Bruce MacIsaac**
Senior Software Engineer
Rational Software

References

   [Note that since *UML Components* was written, UML 1.4 has been released.]


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

**Copyright Rational Software 2002 | Privacy/Legal Information**
Reverse Engineering Legacy Code with Rational Rose

by Francesco Bonfiglio
Technical Lead
Rational Software Italy

Faced with new sources of competition and new forms of partnership, many enterprises seek to integrate, streamline, or replace existing business applications and data sources. Often these efforts entail the intensive reuse of legacy code. Take a manufacturer that currently has a different, standalone quality management system (QMS) in each of its three plants. Many such companies are experiencing pressure to manage quality and other aspects of production across factories, to better serve both customers and suppliers. Whether management chooses to interconnect the existing quality systems or to standardize on a new, third-party offering, development staff will face significant challenges. Among them:

- The need to create a scalable, maintainable, and extendable solution in the shortest possible time.
- The need to work with object-oriented legacy code bases for which there may be no up-to-date architecture model, and little or no documentation.
- The need to incorporate off-the-shelf components from third parties.

Ideally, systems analysts and designers want to build new applications on a component-based foundation that leverages existing code wherever possible. But how best to model, build, and document these legacy components? One option is a laborious, manual re-engineering process, spearheaded by the programmers who know the code best. But few organizations have time for such brute-force efforts these days.

Fortunately, there is an alternative: to reverse engineer the legacy code into a visual model using Rational® Rose®, create appropriate components, and forward engineer the implementation code from there. While human
intervention is still required, reverse engineering with Rational Rose can greatly simplify and accelerate the task of delivering component-based applications that incorporate existing code. Specifically, Rational Rose can help you sort out critical modules much faster, align a visual model with business requirements and system requirements, and keep the implementation code, the model, and the requirements synchronized during the development and maintenance cycles.

In this article, I'll explain in detail how to use Rational Rose to reverse engineer existing object-oriented code, construct a UML model of the classes you need, and create robust, reusable, documented components from the model.

What Is Reverse Engineering?

Reverse engineering with Rational Rose comprises the creation of a UML model, based on the analysis of existing libraries and components. Designers can refine and enhance this model at any time during the software development lifecycle, and re-synchronize the model with the implementation code as needed (more on this below). With help from Rational Rose, teams can integrate legacy code into a new system architecture both more quickly and more accurately than is possible by hand. Provided, of course, that the code is supported by Rational Rose.

In its simplest form, reverse engineering using Rational Rose involves nothing more than the automatic generation of a class diagram from an object-oriented code base. The problem is that such a diagram is typically of little use, unless the code in question is already organized into discrete components or packages. Figure 1 illustrates what I mean. This example is not meant to be readable, but even if it were, the complexity would make it very difficult to interpret.
As you can see, code that did not begin life with a component-based structure does not often yield a picture that lends itself to easy interpretation. If the code was not well structured originally, then a class diagram does not make the relationships and dependencies among classes that much more obvious. For this reason, Rational Rose alone cannot adequately solve the problem of documenting, understanding, and repackaging most legacy code.

Instead, you need to start by working with the code step by step, in order to identify and organize the classes you actually want to reuse. Ultimately, you'll create separate class diagrams and use-case realization structures that define reusable components and can serve as a basis for generating implementation code. The process is straightforward, though it usually requires a day or more of concerted effort on the part of programming staff.

You might well ask, "Why is using Rational Rose better than starting from scratch?" Using Rational Rose is better because it simplifies and automates much of the step-by-step work you'd have to do anyhow. It documents the repackaged components automatically. And it relates your efforts to a UML model, thus leaving you with a far more maintainable and reusable end result. Today's new code is tomorrow's legacy code, after all.

**The Business Case for Visual Modeling**

One way Rational Rose helps automate the repackaging of legacy code is by enabling you to produce a visual model from your code, and then
automatically generate code from your model. When you change the code
generated by the model, you can easily fold the changes back into the model,
and vice versa. In the context of iterative development, this process of
forward engineering, reverse engineering, and re-synchronization of
implementation code with the model is collectively referred to as *round-trip
engineering*.

The business reasons for building a visual model in the first place, and then
keeping your code and your UML model synchronized during implementation,
are convincing in my opinion:

- **Greater Efficiency.** A visual model saves time and reduces wasted
effort by allowing all team members to visualize, understand, and refine
business requirements for the solution and its software architecture,
before coding even begins. A visual model also documents design and
implementation decisions for future reference, and enables you to trace
the architecture model back to both the business process model and the
system requirements. Moreover, you can use the UML model to help
automate the creation of test suites, design specs, and more.

- **Higher Quality.** A visual model helps the entire team visualize existing
code as it is, in reality. Likewise, a visual model helps everyone
visualize an application under construction as requirements mandate it
should look. In effect, the model creates a blueprint that guides the
construction of the application. So everyone has a clear idea of what the
team is building, and how far along they are.

- **Greater Simplicity.** A visual model introduces consistency both within
and across applications, for easier maintenance, greater architectural
stability, easier reuse, and a better way to manage change and reduce
complexity.

For example, why not manage class relationships within a UML model? Think
about how difficult it can be to understand class relationships from reading
code versus how easy it is to view class relationships in a Rational Rose class
diagram. With Rational Rose, you simply:

1. Choose a class and expand it, as shown in Figure 2.
2. Select the elements you want to expand (see Figure 3).

3. Then you can clearly see all the relevant relationships pertaining to the class (see Figure 4).
Despite the automated help Rational Rose offers, some organizations choose to re-engineer legacy code from scratch without visual modeling. Others may wish to reverse engineer the code into UML with the help of Rational Rose, and then forward engineer it by hand.

In particular, I hear objections even from today's Rational Rose users about the value of automatic code generation ("No code generator can write better code than me!"). And that may well be true. But consider that perhaps 70 percent of object-oriented code like Java or C++ is structural code. The code generator can build that for you competently, whereas the manual task of writing all those declarations is time-consuming and subject to error. Particularly in a model-driven project, is it really worth it to model the use cases, extract all the classes, relations, operations, and attributes needed to realize them, create class diagrams, and then write the structural code by hand -- in effect, using the UML model only as a template? In my experience, Rational Rose performs those jobs quickly and reliably, while still enabling you to revise the generated code as required.

**Laying the Foundation**

It is imperative, in my experience, that any large-scale reverse engineering effort begin with an analysis session. The purpose of the analysis is to help analysts and designers understand what parts of the legacy system should be reverse engineered -- because you never have to reverse engineer everything. If they're using the Rational Unified Process® (RUP®), some organizations will
choose to precede an analysis of the code with business modeling; others will start with a formal requirements definition process. Either way, once you establish a clear understanding of what the system needs to do, you can begin analyzing the legacy system in order to develop use-case models in UML. These models not only specify what your system should do, but also guide your reverse engineering activities.

I usually plan on an analysis period of several days or less, depending on the size and complexity of the code base involved. The people it is most important to include are the hands-on developers and architects who know the code and its functionality best. For example, it is extremely important to have on hand someone who understands all the interconnection mechanisms, service mechanisms, relationships among modules, and so on.

Once the right people are assembled, you can begin identifying the classes to be reverse engineered. When upgrading existing systems, these might be subsystems that are highly dynamic, components that contribute to high failure rates, a legacy Web application, or code that has proven difficult to maintain. For new applications, these might simply be the legacy subsystems that can meet the system requirements.

Moving in Reverse

Having identified the key processes to be reused or adapted, you are ready to begin the reverse engineering process in earnest. Table 1 provides a quick overview of the steps involved.

At the conclusion of these steps, you will have created a package for each use case that shows its realization. Within each package, a class diagram identifies all the classes needed to perform the actions described in the use case. Each package will also contain a sequence diagram that identifies what messages are exchanged between objects in those classes, in order to implement those actions.

In effect, you will have created the foundation of a Design Model as described in the RUP. Readers who are familiar with the RUP may already have noticed that the reverse engineering process I just described looks somewhat like the
RUP Analysis and Design workflow, run in reverse.

You will also have created a comprehensive, robust visual representation of your system architecture, one that enables you to navigate from classes to processes and back. This model can be reused, maintained, and expanded along with the code, simply by re-synchronizing as needed.

**Table 1: Steps for Reverse Engineering**

<table>
<thead>
<tr>
<th>Step</th>
<th>Summary</th>
<th>Description</th>
<th>Roles Involved</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Describe what the system will do.</td>
<td>Create a use-case model representing the principal processes within the legacy application that will be reused to meet the system requirements. <a href="#">Figure 5</a> illustrates a use-case model.</td>
<td>Business Analysts, System Analysts, Developers, Users</td>
</tr>
<tr>
<td>2.</td>
<td>Begin identifying the classes you need.</td>
<td>Interview developers to identify the classes within the legacy code that perform or support the actions described in the use cases. List the identified classes in a text note.</td>
<td>System Analysts, Designers, Developers</td>
</tr>
<tr>
<td>3.</td>
<td>Choose the correct settings.</td>
<td>Configure Rational Rose to optimally support the specific reverse engineering activity you want to perform. Settings will vary by programming language, IDE, and other factors. In particular, all elements should be referenced using path-map variables, as opposed to absolute paths. This is important because files might be installed in different locations on different systems. All external references therefore need to be relative. <a href="#">Figure 6</a> illustrates a path-map definition.</td>
<td>Developers, Designers</td>
</tr>
<tr>
<td>Step</td>
<td>Description</td>
<td>Details</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>Reverse engineer the code.</td>
<td>Run Rational Rose against the appropriate subset of legacy code to reverse engineer the classes selected in Step 2 into the UML model. The result is a class diagram, as shown in [Figure 7]. Developers, Designers</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>Start organizing the classes into a logical structure.</td>
<td>In the Rational Rose Logical View, create a use-case realization structure (see [Figure 8]) that contains a package for each use case identified in Step 1. In each package, create a class diagram that identifies the classes that realize that specific use case. Developers, Designers</td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>Identify and add more classes.</td>
<td>Beginning with the class diagram associated with each use case, work with those who know the code to add the main classes that realize the use case. Use the Query Expand capability (as shown in [Figure 4]) to get a complete picture of the additional classes involved. Then double-check the results with your team. [Figure 9] illustrates an expanded class diagram. Developers, Designers</td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>Create sequence diagrams.</td>
<td>Again, working closely with experienced developers, consider the behavior described in each use case. First identify the main flow of events; then identify one or more alternate flows. For each flow scenario, create a sequence diagram (see [Figure 10]) within the use-case package. Then populate the diagram with instances of the classes you identify as resolving the use case. Do this, at a minimum, for the main Developers, Designers</td>
<td></td>
</tr>
</tbody>
</table>
8. Construct the flow of messages between objects. Now work closely with the developers to specify the flow of messages between the objects that realize a specific flow scenario within a specific use case. Figure 11 shows a sample flow scenario. When you create a message in Rational Rose, you can right-click on it to see all the operations for that object. This greatly simplifies the process.

| 8. | Construct the flow of messages between objects. | Now work closely with the developers to specify the flow of messages between the objects that realize a specific flow scenario within a specific use case. Figure 11 shows a sample flow scenario. When you create a message in Rational Rose, you can right-click on it to see all the operations for that object. This greatly simplifies the process. | Developers, Designers |

**Conclusion**

As you can see, reverse engineering remains mostly a human-powered process. There is no Harry Potter magic involved, although the capabilities of Rational Rose simplify things dramatically. I've used this same, basic approach on some very large projects, and across many different technologies (VB, ASP, JSP, C++, etc.). Customers who adopt the process outlined above are almost invariably happy with the results.

Often, businesses that have positive experiences with reverse engineering begin to use UML and round-trip engineering on new projects. Thanks to these capabilities, there's no need to abandon the model and move into "uncharted territory" when the implementation diverges from the original model. Once you've created a model and a framework, it's a simple matter to reverse engineer the implementation code back into the model, within the framework. In fact, by simplifying the inclusion of ongoing changes into an updated design, reverse engineering can often become part of the day-to-day process of software development, not just a way to deal with legacy code.

For more information on the products or services discussed in this article, please click [here](https://www.rationalsoftware.com) and follow the instructions provided. Thank you!
Dear Dr. Use Case
What About Arrowheads on Communication-Association Lines Between Actors and Use Cases?

by Leslee Probasco
Rational Software Canada

Dear Dr. Use Case,
We're about to begin a big project, and in looking at sample use-case diagrams in other sources, I've noticed a lack of consistency with respect to the presence or absence of arrowheads on communication-association relationships between actors and use cases in use-case diagrams.

After a rather vocal and agitated discussion among my colleagues, we realized that we were not able to come to a conclusion about this issue ourselves. Can you provide a UML-compliant "Rational" standard for diagramming communication-association relationships between actors and use cases?

Thanks for "pointing us in the right direction" on this issue.

Signed,
In What Direction Should We Go?

Dear What Direction,
This question was actually put to me a couple of years ago by some Rational folks. Since UML is not prescriptive in this case, and I (of course!) did not want to make any "rash" decisions (being a doctor and all...), I had to toss aside my stethoscope, consult the law (i.e., the OMG UML 1.4 Specification), and join my esteemed colleagues on the Rational Supreme Court to obtain a clear, fair, and consistent ruling.

My Proposal
I presented a proposal, along with some other options that I believed were acceptable and UML compliant (see references below), as well as a few less-savory options. You'll see all of these below.

My personal favorite, and therefore my number one proposal:

1. Show an arrowhead **only** on the communication-association line from the actor initiating the use case to the initiated use case, to show the direction of the initiation (actor-to-use-case only). See Figure 1. All other communication-association lines have no arrowheads.

![Figure 1: Option #1 -- Arrowhead ONLY on Communication-Association Lines from Actor-to-Use-Case](image)

Other possible options:

2. Show **no** arrowheads on communication-association lines between actors and use cases. See Figure 2. Strictly speaking, this is the convention in UML 1.3. However, arrowheads are allowed by the standard, if needed, for clarity. I believe this makes the diagram look too static and does not give the additional information provided by the arrowhead in option #1, which better communicates the value obtained, and to which actor.
3. Include an arrowhead on all communication-association lines between actors and use cases, to show the direction of the initiation (actor-to-use-case or use-case-to-actor). See Figure 3.
4. *Show an arrowhead only on one-way communication that has no intended response.* An example for this case might be output to a printer, although most printers also have a function to communicate print status.

I could list more options, but I think these are the main ones.

Some further notes and cautions:

- Make sure an arrowhead won’t be perceived to show data flow -- it should only be used to show (initiation of) communication.
- I believe double-headed arrows only clutter the diagram. Two-way communication is the default assumption.

That was the text of my proposal. I welcomed discussion and comments from all the use-case gurus I queried. I gave them one week -- assuming they would all be too busy to answer within that time. Then, of course, I could just issue the high-court ruling (at my whim).

What ensued was actually a very lively discussion, which would take up more space than this article allows. Although I found the discussion quite enlightening (and it's always good to hear that folks are all still alive out there!), it was not at all conclusive. So I still had some work to do.

**The Ruling**

After assimilating all the arguments presented, I decided to issue the following discussion and ruling:

> Association relationships between actor classes and use-case classes are used to indicate that the actor participates and communicates with the system containing the use case. Association relationships are denoted as solid lines or paths. Arrowheads may be used to indicate who initiates communication in the interaction. If an arrowhead points to a use case, the actor at the other end of the association initiates the interaction with the system. If the arrowhead points to an actor, the system initiates the interaction with the actor at the other end of the association.

By convention (and high-court ruling):

- **RU courses will show the initiating actor-to-use-case arrowhead only.**
- The use-case-to-actor arrowhead **may** be used if necessary, for clarification.
- **No** double-headed arrows are to be used (two-way communication is assumed).
Note that the ruling was made to ensure consistency in our courseware and to provide a rationale for instructors responding to students' questions. In practice, each project should set its own conventions (and document these in the Use-Case Modeling Guidelines).

Remember the Golden Rule of Use Cases:

> Effectively communicate the requirements to ALL stakeholders.

Hope this helps.

Usefully yours,

Dr. Use Case

P.S. The aforementioned historic decision was made when UML 1.4 was in the proposal stage. For you UML-heads (I know you will be keeping me honest!), I just took a look at the now-adopted (as of May 2001) UML 1.4 specification, and we are still okay. The excerpts I considered are as follows:

"An association between an actor and a use case is shown as a solid line between the actor and the use case. It may have end adornments such as multiplicity..."

"Binary associations are shown as lines connecting two classifier symbols. The lines may have a variety of adornments to show their properties..."

"The (association) path may have graphical adornments at each end where the path connects to the classifier symbol. These adornments indicate properties of the association related to the classifier. The adornments are part of the association symbol, not part of the classifier symbol. The end adornments are either attached to the end of the line, or near the end of the line, and must drag with it..."

"An arrow may be attached to the end of the path to indicate that navigation is supported toward the classifier attached to the arrow. Arrows may be attached to zero, one, or two ends of the path. To be totally explicit, arrows may be shown whenever navigation is supported in a given direction. In practice, it is often convenient to suppress some of the arrows and just show exceptional situations..."

For more information on the UML, please refer to the OMG's UML Resource page.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
Business Process Modeling and Simulation with UML

Part I: Defining a UML Transaction Model That Maps to RUP Business Models

by Pan-Wei Ng
Software Engineering Specialist
Rational Software Singapore

Business modeling is an important activity in software development: For software to be useful, it must directly support the business. Once a sound business model is developed, it is possible to explore different business improvement options such as eliminating redundant tasks, and automating repetitive and mundane tasks or those prone to errors. Although the Rational Unified Process® (RUP®) provides a systematic approach to visually representing a business model, it does not provide a way to derive simulation models that support quantitative comparisons between various improvement options. In this first part of a two-part series, we will define a Unified Modeling Language (UML) profile for a transaction model that fills this gap, and show how it can be mapped from business models in the RUP. Next month, Part II will discuss how to execute the transaction model in a simulation, using the Rational Rose® Process Simulator Add-In.

The business modeling discipline in the Rational Unified Process (RUP) provides a way for project teams to:

- Understand the structure and dynamics of the organization in which a system is to be deployed (the target organization).
- Understand current problems in the target organization and identify potential areas for improvement.
- Ensure that customers, end users, and developers have a common understanding of the target organization.
Derive the system requirements needed to support the target organization.

To achieve these goals, the business modeling discipline describes how to develop a vision of the new target organization, and, based on this vision, define the processes, roles, and responsibilities and their associated organizations within the business models. A key strength of the RUP approach is that business modeling follows the same principles as object modeling for software development; the use of common principles makes the resulting business model easy for stakeholders to understand.

What Is Use-Case-Driven Simulation?

Although business models in UML have a standardized notation that facilitates discussion of existing problems in target organizations and how these problems can be solved, the models lack the quantitative dimension that permit stakeholders to make objective comparisons between the current process and proposed changes, or between different proposals. The RUP counters this inadequacy by recommending Activity Based Costing (ABC) to provide the needed mathematical support. The problem with mathematical computation is its unwieldiness when the model grows in size and complexity, a by-product of the stochastic nature of business processes (i.e., these processes are subject to random variables and guesswork).

Simulation, specifically the technique of Discrete Event Simulation (DEVS), poses an attractive alternative because it is less mathematically demanding in its specification, but it still offers sufficient analytical power to compare different improvement options, such as adding more computing resources or modifying current work processes. In addition, DEVS follows modeling principles similar to those in UML, such as abstraction, modularity, and hierarchy. Instead of manipulating mathematical formulae, one only needs to manipulate simulation entities (i.e., classes) and their collaborations.

One inadequacy of DEVS modeling is the lack of a "use-case-driven" approach that emphasizes business goals and elaborates how collaborating participants contribute to these goals. Instead, DEVS emphasizes organizational structure and individual participant behavior, which consequently produces a larger-than-required model, because even those activities not related to the business goal under investigation are included.

Consequently, it would be highly advantageous to have a UML-based business-process simulation methodology that is "use-case-driven." The benefits would include:

- Common notation based on UML.
- Analytical capability of DEVS.
- Common modeling principles as per the RUP.

In the remainder of this article, we will derive a UML profile for business-
process simulation by walking through a simplified example of an outpatient hospital. First, we'll describe the hospital work processes through RUP business models and then map these models to a transaction model that can be simulated.

**Deriving a Transaction Model**

The RUP specifies two models for business modeling:

- **Business Use-Case Model.** Describes business processes from an external perspective.
- **Business Object Model.** Describes business processes from an internal perspective.

We will refer to these as the *RUP business models*. From these two models, we will derive a *transaction model* to describe the business process from a simulation perspective. We call it a *transaction model* rather than a *simulation model* because it addresses transactions that are handled by the business; simulation is a technique for analyzing these transactions. The transaction model operates at a different level of abstraction from the business object model and has more details to support quantitative analysis.

Transactions are traced from use cases and their realizations, and they are the basis for simulation. The relationship between the RUP business models and the transaction model are depicted in Figure 1.

![Figure 1: Relationship Between RUP Business Models and Transaction Model](image)

**RUP Business Models**

The RUP Business models are comprised of several stereotyped UML elements, namely:

- **Business Actor.** Represents a role played in relation to the business by someone or something in the business environment.
- **Business Use Case.** Defines a set of business use-case instances; each instance is a sequence of actions a business performs that yields an observable result of value to a particular business actor.
- **Business Worker.** Represents an abstraction of a human or
system that acts within the system. A business worker interacts with other business workers and manipulates business entities while participating in business use-case realizations.

- Business Entity. Represents passive objects that business workers access, inspect, manipulate, produce, and so on.

- Business Use-Case Realization. Describes how a particular business use case is realized within the business object model, in terms of collaborating objects (instances of business workers and business entities).

Our example, an outpatient hospital, illustrates the concepts prescribed in the business modeling stereotypes. The business use-case model comprises a set of business actors and business use cases. In our simplified example, we have a single business actor, the Patient, and a single business use case, Visit Doctor. See Figure 2.

![Figure 2: Visit Doctor Business Use-Case Model](image)

Within the hospital, we find several business workers, namely:

- Nurses who make simple clinical measurements before the patient sees the doctor.
- Doctors who make the actual diagnosis and prescribe drugs.
- Dispensary staff who collect payments and dispense drugs.

Figure 3 is a sequence diagram of a business use-case realization for a typical visit. The use case begins when the patient walks into the hospital. The nurse retrieves the patient's medical record and takes clinical measurements from the patient, such as temperature and blood pressure. Then the doctor starts to make a diagnosis and may prescribe suitable drugs. The patient subsequently goes to the dispensary to collect the prescribed drugs and pay for them after completing a subsidy form.
The Sequence Diagram in Figure 3 does not show any business entities. This is because the purpose of simulation is largely to compare transactional durations and human and system resource consumption, which are primarily dependent on active elements (business actors and business workers) in the business process. Business entities are more useful in enterprise modeling that captures business domain details and are often a basis for database design.

**The Transaction Model**

The active elements in the business models and their collaborations are used to derive elements in the transaction model. The transaction model comprises a set of transactions and participating transaction elements as depicted in Figure 4. Each transaction comprises a number of transaction flows and is visually represented in interaction diagrams, as exemplified in Figures 7 and 8, which show sequence and collaboration diagrams, respectively. Each step in a transaction flow is represented as a message.

Transaction model elements participate in transactions and are further categorized as generators and servers (commonly used terms in simulation literature). Generators are able to initiate transactions, whereas servers are service providers. Each time a service is rendered, a resource unit is held for a period of time and then released after the service is completed. For example, once a doctor has completed a diagnosis and has prescribed drugs, then he or she is available for the next patient. Generators are traced from business actors, whereas servers are traced from business workers. Transactions are traced from business use cases.

Each transaction model element has a set of operations, which are categorized as initiations, services, and pauses. As the name implies, initiations indicate the initiation of a transaction flow. Each step in the
transaction flow may require a transaction model element to provide a service, or it may pause for a while (to allow time to think).

The Visit Doctor business use case in the outpatient hospital example is realized by the Visit Doctor transaction depicted in Figure 5.

Figure 5: Traceability from Business Use Case to Transaction

The traceability from the business model elements to the transaction model elements is depicted in Figure 6. In this example, we have a one-to-one relationship.
The transaction flow is derived from the Visit Doctor business use-case realization and is depicted in Figure 7. Note that, aside from a change in the stereotypes of the participating objects, there is no difference between Figure 7 and Figure 3. This highlights the fact that our transaction modeling approach follows the same paradigm as that of the RUP business modeling approach.

Collaboration Diagrams offer an alternate representation of the transaction flow. The Sequence Diagram in Figure 7 is represented as a Collaboration Diagram in Figure 8. This representation is more familiar to DEVS practitioners who frequently deal with queuing models, in which our icon stereotypes are founded. The icon representation for a server looks like a queue; the one for a generator looks like a signal generator. Links in the Collaboration Diagram appear as possible paths for information flow.
The view of participating classes (VOPC) for the Visit Doctor transaction depicted in Figure 9 is more interesting, because it is here that the differences between business modeling and transaction modeling start to appear. The simulation of the transaction model requires the specification of numerical quantities, such as timing and resource usage mechanisms.

**Server Multiplicity.** Each server has a multiplicity that indicates how many instances it can possibly have. In our example there are two doctors, three nurses, and one dispensary.

**<<initiates>> Operation Flow Multiplicity.** Each generator <> will generate a number of transaction flows indicated by its multiplicity. In our example 100 typical visits are generated. The notation "Transaction.Transaction Flow" is used to indicate the specific transaction flow initiated.

**<<initiates>> Operation Inter-Arrival Time.** The transaction flows have a specified inter-arrival rate. In our example the time between visits is fifteen minutes.
● **Operation Duration.** There is a finite waiting time. In our example it takes two minutes to complete the subsidy form.

● **Operation Duration.** Each service takes a finite time period to complete. In our example a nurse takes one minute to retrieve the patient record, and a doctor takes fifteen minutes per diagnosis.

● **Operation Resource Usage.** Each time a service is processed, it holds a number of resources. In our example, each time a patient is serviced, one resource unit (i.e., a doctor, nurse, or dispensary staff person) is held.

These parameters are represented as tag values in the transaction model classes, attributes, and operations.

Now that we have defined our transaction model, we are ready to execute it in a simulation. In Part II, which will appear in the May issue of *The Rational Edge*, we will show how to perform a simulation with the Rational Rose Process Simulator Add-In.

---

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

by Doug Foote
Project Management Specialist

During my employment at Page Technologies, I had the opportunity to introduce the concept of the RUP® (the Rational Unified Process®) to a variety of individuals in a variety of scenarios across a variety of business domains. In some scenarios, I was in the role of project manager, in some I had a sales role, in others I was conducting a seminar. In each case, the challenge was how to effectively communicate the robust capabilities that the RUP offers.

In a nutshell, I find that it's helpful to compare the RUP to a restaurant buffet. That's right.

Before I explain, though, I'd like to suggest that you might find this analogy useful if your job involves trying to sell or introduce RUP to an organization. The buffet comparison is most helpful in communicating the basic concepts of RUP: the four phases, iterative development, tailoring RUP to meet the business need, and so on. You may find it useful if:

1. You are a project manager trying to convince your team members or management that your organization should adopt RUP.

2. You are part of a process group that wants the broader development organization to adopt RUP.

3. You are trying to change the opinion of an organization or individuals who think RUP is an inflexible monolith that can't compete with processes such as XP, RAD, etc.

4. You are trying to convey the value of RUP to an organization or individuals who think process is just a make-work effort geared at large artifact thwomp factors (i.e., how loud of a sound a document
The goal for this article is not to provide ROI analysis or empirical data that will allow you to justify implementing RUP. Rather, it is to provide an analogy that will help your audience understand how RUP should be implemented.

The Basic Analogy

Let me start off by stating that I love buffets. I enjoy the periodic lunchtime jaunts to the local Chinese, Indian, Thai, and Mexican buffets that my development teams and I have taken over the past many years. It was after one of these forays to a local Minneapolis restaurant a few years ago that it occurred to me: The RUP is not unlike a visit to one of these buffets. You have requirements, and you have a potential solution. The requirements are driven by a need (hunger) and the solution (food) is provided by the buffet. How you deliver this solution to your stomach is the process. Chances are, if you are like me, that you will make several trips to the buffet (iterations) and use several tools and activities to assemble different foods from different courses (phases). Table 1 details this analogy.

Table 1: Similarities Between a Restaurant Buffet and RUP

<table>
<thead>
<tr>
<th>Point of Comparison</th>
<th>Buffet</th>
<th>RUP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Need</strong></td>
<td>Buffets are designed to satisfy hunger, a basic need that is fulfilled by food. But we should only eat until we are no longer hungry. Generally, it is not considered good practice to eat just for the sake of eating, without being driven by hunger.</td>
<td>Technical solutions are designed to fulfill business needs, and we should not try to create a solution without first defining the business need (first step in RUP). Can you imagine a software development team working on a project just for the sake of designing and writing code? (You've seen this happen? Well, let's hope that's the exception to the rule.)</td>
</tr>
<tr>
<td><strong>Process</strong></td>
<td>A buffet provides a framework (buffet table) and process (loading up your plate) that allows you to meet your hunger need.</td>
<td>RUP provides a framework and process that allows you to meet your business need and that supports your development effort.</td>
</tr>
<tr>
<td>Selecting Among Options</td>
<td>When you visit a buffet, you take what you want and need from the table to meet the hunger need. Although some gluttons will eat every single item at a buffet, their numbers are few. The premise of the buffet is that you will make thoughtful selections from many options to make your experience both enjoyable and nourishing. Every experience you have with a given dish informs how you will approach it in the future.</td>
<td>You tailor RUP to meet your business need. Although a few critical initiatives may actually require most of the features in the RUP, they are the exception. The premise of RUP is that you will make thoughtful selections from many options to make your experience enjoyable and successful. Every experience you have with the RUP determines how you will approach it in the future.</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Phases</td>
<td>Most buffets (in the US) are divided into four courses: appetizers, salad, entrée, and dessert. Although they all provide sustenance, most of us progress through them in the order listed. And there's reason behind this: A salad cleanses the palate between courses; sugary desserts suppress further appetite, so they're best at the end of the meal.</td>
<td>RUP is divided into four phases: Inception, Elaboration, Construction, and Transition. To succeed, development typically must proceed in this order: You can't build the right system without assembling requirements; you can't generate the right code unless you have a detailed system architecture, and so on.</td>
</tr>
<tr>
<td>Iterations</td>
<td>One advantage of a buffet is that you can take as many &quot;trips to the trough&quot; as you need. You can progress through the different courses gradually, over multiple trips.</td>
<td>One advantage of RUP is that it is an iterative process. You go through the entire process of formulating requirements to delivering an executable - multiple times -- in a structured manner. This is the key advantage of RUP over the traditional waterfall process, which</td>
</tr>
</tbody>
</table>
allows you to visit each phase only once.

<table>
<thead>
<tr>
<th>Assessment</th>
<th>After each trip to the buffet, you assess the food you eat. Does it hit the spot? Does it suit your taste buds? And if you're concerned with nutrition, this can get a little complicated. For instance, maybe it tastes good but isn't good for you. In any case, you can always return for more, or at least keep your assessment in mind for future visits to the restaurant.</th>
</tr>
</thead>
<tbody>
<tr>
<td>After each iteration in RUP, you perform an iteration assessment to determine what aspect of the iteration met the business need, what was successful and not successful, and what will be needed for the next iteration. You will return to work on some items and learn what to utilize for future projects. You will also discard those items that do not add value.</td>
<td></td>
</tr>
</tbody>
</table>

The point of this comparison is to drive home that the RUP is not an all-or-nothing proposition. The intent of the RUP is not for a team to utilize every possible activity, template, and artifact. The intent is for teams to tailor the process to meet their particular business need. Just as there are core tools associated with a buffet that you must use every time to be successful in eating -- plates and cutlery, for example -- so too are there core aspects of RUP that you should use every time (its four phases, iterative development, etc.) to make your project successful. What you decide to put on your plate at the buffet, and how many times you decide to return are up to you. Likewise, with the RUP, you decide what to implement in your project and what is required to meet the business need. Ultimately, you and your team are the ones who decide how to leverage the process.

**Preparation for Introducing RUP**

**Know Thyself**

There is no "perfect" way to implement RUP, except to assess the organization and situation in which you will be rolling it out, follow the best practices, develop iteratively, and do your best to get your team fully involved. The first step to success is to "know thyself." If you are trying to sell or introduce the RUP to an organization without prior experience, then you will have an uphill battle. Educate yourself through training and mentoring. Attend your local Rational User Groups and meet with others in your industry who have RUP rollout experience. Check out the various discussion forums available at http://www.rational.com/support/usergroups/index.jsp, including the new Rational Unified Process discussion. Education and experience will make you much more effective in selling and/or introducing the RUP to your team.
Know the Organization

It's also important to acknowledge that people in your organization have varying levels of process experience, and you should know the organization well before performing the RUP's organizational assessment. Some people will have development backgrounds, some will have business backgrounds, some will have domain experience in the commercial arena, some in government, and so forth. Most will come with preconceived notions about development process, and some with preconceived notions about the RUP. Be prepared to answer any questions they might have, and don't get surprised by individuals who might be anti-RUP or anti-process. Prior to your meeting, write down questions that a person with a particular background (business, technical, etc.) might ask, and then develop an informative answer that will help in their decision-making.

Know About Prior RUP Implementations

Find out if individuals in your organization have had prior experience with the RUP, especially unpleasant experiences. Most of these result from improper implementations, in which the RUP was presented:

1. As a monolith.
2. As a waterfall without iterations.
3. As an all-or-nothing proposition, so the team felt compelled to use all templates, workflows, and so on, whether they were necessary or not.
4. As part of a Rational tool purchase with little or no education on the value of process.
5. Out of the box with no training, mentoring, education, or prior experience.

(See also "The RUP: An Industry-wide Platform for Best Practices," in the December 2002 issue of The Rational Edge.) Talk ahead of time with the team that will be ultimately responsible for the success of the project; determine what their process experience is, and what caused problems for them. Head off the possibility that someone will put the kibosh on your RUP initiative based on a bad experience created by improper implementation.

The rule is the same for people who have had good experiences with RUP. Find out what was successful about their implementation (preferably from iteration assessments or lessons learned), document the information, and then leverage it to support your position with decision makers.

Conclusion

Some people will grasp the concept of the RUP immediately, but others will have a more challenging time. Try starting with the buffet analogy to establish a framework for understanding. This has been effective every time I've used it, especially with people who are new to process. Get your
audience engaged with the analogy, and see if they can create examples and draw more parallels themselves. The exercise need not take more than a couple minutes, and the very act of doing this will get them more engaged and help further their understanding.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
Bridging the CM Gaps: Use Case Analysis of a New Configuration Management System

by Christian Buckley, e2open
and Darren Pulsipher, Cadence Design Systems

Editor's Note: Each month, we will feature one or two articles from the Rational Developer Network, just to give you a sense of the content you can find there. If you're a licensed Rational customer, you should join the Rational Developer Network now!

Building a bridge is one of the most fundamental ascents of mankind over nature. Few endeavors over the last 100 years have posed such risk while also providing such great utility: linking regions and communities otherwise locked away behind distance and, usually, water. Without the vast systems of roads and bridges, man would be forced to travel less efficiently, and society would be disjointed.

In the United States, for example, the development and success of two major geographic regions came about largely due to the building of bridges.

Manhattan is an island, with the East River to the east, the Hudson River to the west, New York Harbor to the south, and the Harlem River to the north. Boat and ferry service was unreliable, particularly in winter, when ice jams often prevented boats from crossing. The construction of the Brooklyn Bridge was the first major step in linking Manhattan Island and the surrounding areas with a system of bridges.

On the opposite side of the country, the dream of spanning the Golden Gate Strait had been around for well over a century before the Golden Gate Bridge opened to traffic. The Golden Gate Strait is the entrance to the San Francisco Bay from the Pacific Ocean. The strait is approximately three-miles long by one-mile wide with currents ranging from 4.5 to 7.5 knots. Construction commenced on January 5, 1933 and the Bridge was open to vehicle traffic on May 28, 1937.
There are bridges in software development as well. Building a configuration management system can be a daunting task for any organization, and yet many teams, in the rush to solve immediate problems and focus their time on "billable" development activities, will push some kind of CM solution out the door without properly analyzing the organizational needs, short-term and long-term.

As we're sure you've all seen, far too many teams are tackling problems without understanding the full scope of the project in front of them. And yet time after time, that's how projects get underway ý with arbitrary deadlines ý and usually behind schedule from the start. Customer demands and project deliverables require a speedy implementation. But a speedy deployment and well planned and developed system do not have to be mutually exclusive.

One common management dream is to start on a project at the beginning of the lifecycle, rolling out process and procedure at the inception phase of development. However, this is rarely the case. Luckily, configuration management systems are available to aid in your development efforts. They can be a powerful asset to increasing communication, productivity, and quality through process automation and integration of the tools that most engineering groups use today. You could call CM the "bridge" that links your company's development teams together. Implementing some kind of CM solution will organize your development efforts around solid and repeatable processes ý and by helping your team prioritize and manage the development lifecycle, you are more likely to meet your customers' needs.

Configuration Management and the teams that manage these tools are in a unique position because they are the glue between Engineering and the rest of the product development structure in your organization. The product cannot move forward in a timely manner unless the CM team coordinates with the build and release teams, and, generally, manufacturing. And unless your project is proceeding quickly, it's a safe bet that your customers are not happy.

Now...back to the steps that lead you to the solution. For those of you just joining us, the process of determining the path and structure of the system as it is applied against your organization is called Business Modeling. The purpose of business modeling is to determine who and what the customer is ý but not necessarily to outline the requirements of the project. The point is to seek to understand the client's perspective, and not make any judgments about possible solutions or what the customer thinks he or she needs.

The purpose of this article is not to walk through the do's and don't of business modeling ý there are plenty of articles available on the Rational Developer Network on the subject. (For an excellent review of the why's and how's of business modeling, we recommend Business Modeling with UML: The Light at the End of the Tunnel by Bryon Baker.) It's probably
safe to say that business modeling is the most undervalued part of the software development process. With that said, we feel it is important to remind everyone that we are not trying to identify the requirements at this stage. The business model is the "problem domain" while the requirements are the "solution domain", which come later in the development process. Instead, it is critical to know what you are building and why.

"Models help a software development project team visualize the system they need to build that will satisfy the various requirements imposed by the project's stakeholders. The aphorism "A picture is worth a thousand words" holds especially true when it comes to work involved in developing software: much of what's interesting about a given system lends itself to visual modeling better than plan test does. The UML is specifically designed to facilitate communication among participants in a project." (From UML Explained, Kendall Scott, Addison Wesley, 2001)

**Identifying Actors**

When analyzing a system, we first like to draw a box around the system or define the boundaries of the system. This is best done by defining the actors (the people, software, hardware, etc.) that interact with the system. In the configuration management world, we can usually write down a list of actors that participate in a CM system fairly quickly. Remember ý think of the CM system as a black box. Just make sure you define all of the outside parts of the black box.

Here is a diagram of the actors within a basic CM system, and a brief description of each:

- **Software Engineer**
  - The Software Engineer is responsible for the design and

![Diagram of actors within a basic CM system](image)

**Figure 1**

- **Software Engineer**
  - The Software Engineer is responsible for the design and
development of the software that makes up your product. The software engineer will use the Configuration Management system to store designs, code, and sometimes tests, so that they can be integrated into a releasable product.

- **Configuration Management**
  For those of us who come from the CM world, you might consider the Configuration Manager as the most important actor in any CM system. It is typically their job to control the "crown jewels" of the company ý the source code. We typically see the CM people not only administrating ClearCase and the process of software development, but these individuals are also found performing the builds and releases of the product, as well. This is especially true in smaller organizations where resources are tight.

- **Quality Assurance (Product Validation)**
  We prefer the term "Product Validation" instead of "Quality Assurance", because these individuals do much more than check the product to find bugs the software engineers created. These teams play a very important role as the final validation of the product use cases before going to the customer. They will use the CM system to store their tests and test suites, and get controlled release of the product from the Configuration Manager.

- **Management**
  Whether we like it or not, management is a necessary evil. We deal with management on a daily basis (some days better than others), and, as the providers of the company’s CM solution, they recognize that we provide them with the information they need to make decisions about product direction and strategy. The CM system contains the information that Management needs, but it is our job to present the data in a manner that is useful to them.

- **Customer Support, or a CRM system**
  Another necessary actor is the Customer Support organization. Customers just might have problems with your software ý no matter how well it was designed ý and you need to be able to give information to your customer support organization about product releases, customer requests, and bugs that have been fixed in product releases. This information can either flow directly to the support team, or be tracked within a CRM solution.

- **Defect Tracking system**
  This can be part of the system or not. If you do not define this as an actor, you should at least be ready to include it in your system later down the road. If you are not integrating your CM solution with a Defect Tracking system, you should seriously consider this option. The joining of the defect data and the source code can help you make improvements in project management and the overall quality of your product.

- **Technical Writers**
  Make sure you include everyone that will develop artifacts for your product. There is nothing worse than getting documentation that is two versions behind the shipped software. If your Tech Writers are not using ClearCase, then teach them how to use it. If it is too hard, then write the scripts that they can use to make their jobs easier.
Everything will run much more smoothly if they at least know the basics. Just keep it simple.

- **System Administrator**
  Your Configuration Manager is not a System Administrator. Of course, in a small organization it could be the same, but these roles are very different. The system administrator is there as a support for your hardware and potential software problems. They should be making sure that the systems are configured according to product specifications, and fine tuned to the requirements of the CM team.

**Use Cases**

Now you need to look at how the actors use the system. For each actor, write down how the actor will use the system or how the system will use the actor. Draw an arrow in the direction of information flow. Don't forget the administration use cases, such as "Backup the Source Code", "Restore from Backup", etc. Don't include use cases that do not involve the CM system. For example, "Reboot Machine" is probably not a good use case, because it does not directly involve the CM system.

Pick the most used use cases and start working from there. Look at how the actor will use the system and how the system will guide the user. The CM world has some well defined scenarios, so try to focus on what is different about what your team does compared to other teams.

![Activity Diagrams](image)

**Figure 2**

**Activity Diagrams**

The next step is to look at activity diagrams and process analysis. The first thing you will find out is that your organization most likely already has some process in place (whether you like it or not). Whether or not there are formal guidelines for development, people typically work the same way over and over again. They get into a mode of working so they don't have to think about what comes next. They would rather spend their time working on the new things instead of worrying about what the next step in the process is. What you need to do is figure out what your team is currently doing and model it. Then you can find ways to optimize it.
Don't fall into the trap that process engineers run into time and time again: "I am going to develop a new process that will make everyone more productive." In reality, it usually has nothing to do with the team culture or habits, and hinders more than helps the organization. First, find out what your team is doing, and then try to change with little steps of improvement.

Here is an example of an activity flow of a typical CM system:

![Activity Flow Diagram](Click to enlarge)

**Figure 3**

**Component Diagrams**

One aspect of designing a Configuration Management system that is constantly missed is the definition of components. While you won't know all of the components of your system are at the beginning, with your knowledge of the system you should still be able to group things pretty well. You should also consider the actors of your CM system when defining components. Don't forget about the product's supporting files, such as Licenses, Documentation, Releases, and so forth.

Consider the following small project named "Kish":

- Kish_adm ý Administration VOB, for labels, branches, triggers, and supporting cm scripts
- Kish_process ý This is a process VOB, UCM likes to use this to store information about the project in the VOB
- Kish_doc ý Documentation VOB, your technical writers need a place to put their information
- Kish_src ý Most products have some kind of source that gets compiled
- Kish_release ý We prefer to store the releases in a separate VOB
from the src. It gives us the opportunity to multi-site the VOBs separately, and scrub them differently.

- Kish_test ý Most test harnesses become larger than the code base itself. This is great again for multi-site purposes.
- These VOBs can be broken down into more VOBs, according to the project size. We will talk about that in the next article.

**Figure 4**

**Deployment Diagrams**

For the context of this CM effort, deployment diagrams can be used for three things: First, to show the different machines you need to support the size of your team, VOB servers, view servers, build machines, test machines, and workstations. Second, to show the different platforms that you need to support. This is overlooked far too many times, and can have an impact on the way you set up your VOB structure and your multi-site strategy. And third, to show the different locations that will be working on the product and the VOBs that they need. This helps you with your multi-site strategy dramatically.

**Bridge Building**

Well, we've attempted to clearly illustrate the steps to designing your new CM solution, and hopefully you have figured out how to get started on this planning phase. As outlined in their book *Use Case Driven Object Modeling with UML* (Addison-Wesley, 1999), Doug Rosenberg and Kendall Scott summarized the goals of use case modeling:

- You've built use cases that together account for all of the desired functionality of the system.
- You've produced clear and concise written descriptions of the basic course of action, along with appropriate alternate courses of action, for each use case.
- You've factored out scenarios common to more than one use case.
Once you have accomplished these things, you should be ready to design the next stage of your CM system.

Remember — bridges don't just happen. They take years of planning, followed by systematic implementation. The result is a resource that can be used to improve the lives of those who use them. The same can be said for a solid configuration management system.

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