the Rational edge
e-zine for the rational community

COVER STORY

The RUP: An Industry-wide Platform for Best Practices

– by Per Kroll

december 2001

enter issue | mission statement | archives | subscriptions

Editor's Notes

Blow Your Horn!

Imagine you're a professional musician, say, a classically trained trumpet player, with years of practice and public performance under your belt. Your horn is your tool, and all those stacks of printed musical scores are your projects -- past, present, and future. As you step onto the stage and find your place in the orchestra, you know your fellow musicians, the conductor, and the paying audience expect you to "get it right." No sweat. You're confident that each of your notes will be the correct ones, and that you'll deliver with perfect tempo and volume. But imagine ... you hear your part coming, and as you raise the instrument you suddenly realize: "This isn't my trumpet!"

This issue of *The Rational Edge* is dedicated to the performer, all of you, who must roll with the constant changes in information technology and, at the same time, deliver what's expected at the end of each software development project. Every day, new IDEs, deployment platforms, and emerging industry standards are adding to the pitch of your learning curve. Which is why the Rational Unified Process® -- the RUP® -- offers a proven method for taking one step at a time. Now in its sixth year as a generally available product, the RUP has become a robust, customizable platform for managing the phases of the software development lifecycle iteratively. In a nutshell, it helps you deliver successfully without requiring you to "get it right" the first time.

For a look at the RUP's history and how the latest release offers a new, customized approach to software lifecycle management, see this month's cover story; there's also a nifty breakdown of the RUP's essential elements in "The Spirit of the RUP." And Murray Cantor is back with the second installment of his technique for applying the RUP to systems engineering projects. For the project manager or others in the lifecycle who want more project tracking capabilities, Paul R. Wyrick and Doug Ishigaki describe the new Rational Project Console tool, now shipping with every copy of Rational Suite.

Need to learn a new programming language? Joe Marasco offers some guidance in Franklin's Kite that will get you up to speed and give you a basis for comparing one language to another as well. And there's more for you techies out there: a great technique for keeping documentation up to speed with the code during a project; a strategy for automating risk management with Rational RequisitePro®; and Dr. Use Case returns with a

close look at word choice in your use-case descriptions.

Speaking of performers, this month's book section includes a review of the new autobiography, *Jack: Straight from the Gut*. As review Sid Fuchs puts it, "Whether you love him or hate him, there's no disputing that Jack Welch delivered some pretty impressive results..." For the rest of us players looking for the ideal route to our industry's equivalent of Carnegie Hall, Rational offers some friendly advice: "Best practice, best practice, best practice!"

Happy iterations,
Mike Perrow
Editor-in-Chief

The RUP: An Industry-wide Platform for Best Practices

by **Per Kroll**
Director, Rational Unified Process Development
and Product Management Teams

download pdf (326 K)

The Rational Unified Process® (RUP®) is a comprehensive, Web-enabled set of software engineering best practices that provide customers with guidance for streamlining their teams' development activities. Earlier this month, Rational announced its collaboration with a broad range of key industry leaders, including IBM, Microsoft Corporation, and Sun Microsystems, to expand the Rational Unified Process into an industry-wide process platform. This article presents the Rational Unified Process as an evolving platform that facilitates software development, and it examines the new capabilities in the latest version of the RUP.

The Rational Unified Process product has evolved from a software engineering process deployed as a Web site providing Rational's guidance on best practices, to an industry-wide platform for best practices. But what do we mean by an industry-wide platform? Essentially, we mean a platform that facilitates software development for any size organization, and which provides -- or allows the addition of -- specialized content for a wide variety of software markets, without overwhelming a practitioner with irrelevant procedures. Today, the software industry's most innovative pioneers including IBM, Microsoft, and HP use the RUP process platform to capture and document their know-how and best practices. They rely on the RUP platform as a distribution mechanism for these best practices; it is a vehicle through which thousands of projects and end users consume and adopt them. In this article we will:

- Discuss the business and technology drivers that have defined an accepted framework of best practices.
- Look at the emergence and evolution of the RUP.
- Examine the underlying components of the RUP platform.

Business and Technology Drivers

Several industry trends have driven the development of Rational's industry-wide platform for best practices. Some of the most influential are:

- Rapid introduction of new technology and evolution of existing technology.
- Standardization and commercialization of tools, methods, and process.
- The trend by consulting companies to no longer include a proprietary process as a competitive differentiator.
- Industry skepticism regarding traditional and heavy-weight processes that are dogmatic and manager focused.

These trends are discussed in the following sections.

Rapid Evolution of Technology

It is very difficult for developers to keep up with constantly evolving technology, and lack of knowledge is becoming a major impediment for the adoption of new technology. To address this issue, vendors have increased their focus on guidance and best practices in the marketplace, with the goal of effectively distributing their advanced tools and technologies onto a developer's desktop. Examples of such solutions include MSDN, Oracle Technology Network, and Vignette Global Marketplace. The goal is to put the knowledge that developers need just a mouse click away on their desktop.

Standardization and Commercialization of Tools, Methods, and Process

Over the last 30 years, the software engineering industry has continuously moved toward standardization and commercialization of technology and knowledge. This meant that in the 1960s and 1970s, commercial compilers become available; in the 1980s, CASE tools and databases; and in the 1990s, advanced configuration management systems and IDEs. In the mid 90s, we also saw standardization in the methods and modeling language area. The Unified Modeling Language (UML) was originally developed by Rational and its partners, and later adopted and managed by OMG as an industry standard. By the late 90s, companies were ready for standardization of process, and many companies abandoned their homegrown process and began looking for a commercially available one. This allowed the large investments necessary to build an enterprise-wide process, such as the Rational Unified Process, to be shared by many companies. Companies that have standardized on the RUP, for example, include CGE&Y and Merrill Lynch.

Consulting Companies Less Reliant on Process as a Competitive Differentiator

Similarly, while standardization has resulted from the commercial availability of computing technology, the competitive differentiation based on proprietary processes that consulting companies once relied on has also become much less significant as a business driver. Process used to be a selling factor for any consulting company worth its name, but as customers got tired of project overruns and the poor quality of delivered applications, they started to demand some assurance that the practices used by system integrators were proven. Since it is extremely difficult to assess the applicability and value of a homegrown process, customers started to demand commercially available processes with a proven track

record, and which independent reviewers could evaluate more easily. As a result, there has been a rapid move among system integrators away from homegrown processes to commercially available processes. The RUP has been the process of choice for many of these companies, including CGE&Y, Deloitte Consulting, and IconMedialab.

Industry Skepticism Regarding Existing Processes

At the same time, the software development industry became skeptical toward traditional and heavyweight processes that are prescriptive and have a strong primary focus on the needs of managers. These processes typically promoted a "waterfall"[1] sequence of development, functional decomposition, and a document-centric approach. These processes had been popular in the late 80s and the 90s, and proved to be ineffective for several reasons:

- *Processes were of little value to developers*. Since the guidelines primarily focused on describing *what* should be done, and not *how* it should be done, developers saw little value in them. Since it did not help them do a better job, developers resented this type of process.
- *The waterfall approach did not allow effective risk management*. Even worse, since these processes typically followed a waterfall approach, which is ineffective in addressing key risks early in the project (regardless of whether the risks are technical or business related), the project success rate declined as technology evolved and projects became increasing complex and risky. The industry was ready for a change.
- *The process was hard to access*. Processes were often published in thick binders, and the content was not integrated with desktop tools. This made it hard to find the information being sought, so process binders simply collected dust.
- *The process did not focus on delivering value to customers and other stakeholders*. The waterfall-based and dogmatic nature of the processes focused on many artifacts, on heavy documents and activities that did not help deliver real business value to end users and other stakeholders.

The Emergence and Evolution of the RUP

It was in this business climate that the Rational Unified Process gained popularity. From a product development standpoint, the RUP product has gone through three complete product phases and is now moving into its fourth phase (see Figure 1).

Phase 1: 1987-1996

The RUP product was originally developed by Objectory AB under the name of the Objectory Process[2]. The process was formally modeled using object-oriented modeling techniques, which many years later allowed us to componentize RUP, see section Componentizing the RUP below. The Objectory Process focused on Business Engineering, Requirements, and Analysis & Design, and promoted a use-case driven approach to software development to ensure that user requirements were not only captured, but also designed, implemented and tested. The Objectory Process also introduced use cases and object modeling to support user interface design.

Another strength of the Objectory process is the close ties between business modeling and software development, allowing you to link software features to the business processes they support. This coupling allows you to optimize IT investments to maximizing business needs.

Phase 2: 1996-1999

In 1996, Rational Software acquired Objectory AB, and decided to use the Objectory Process product as a baseline for its future process work, including the process modeling technique that now has evolved into the OMG standard SPEM. We now integrated the best practices captured in the Objectory Process with the best practices from several other processes including the Rational Approach[3], with the objective to produce one unified process. It was essential that the process should be iterative, architecture-centric, and use-case driven, since these were some of the main best practices that Rational had identified from its work with its customers and partners. Because Rational has always also had an extremely strong focus on the needs of the practitioner, the RUP process development team believed that the process had to be non-intrusive to the practitioner, and that the average analyst, developer, and tester had to find it was easier to do their jobs with the RUP than without it. These benefits may seem obvious, but prior to the RUP's introduction, most processes could not be described in these terms. The RUP team also realized that the technical process had to be anchored by a solid project management (macro) process to guarantee project success. The macro process needed to ensure that risks were addressed early, that projects were run cost effectively, and that overall business objectives were met.

Having set these objectives, we took on the task of augmenting the Objectory Process by harvesting and adding Rational's know-how into one consistent knowledge base during the years 1996 through 1999, available among others through acquisitions of a wide array of companies. The resulting process was named the Rational Unified Process, or RUP. We merged the concepts of iterative development, architecture-centric development and risk management from the Rational Approach into the RUP, as well as merged the RUP with additional process experience regarding requirements management[4], testing[5], and other areas.

By 1999, the Rational Unified Process covered the full lifecycle, from Business Modeling through Deployment, as well as areas such as Configuration Management and Project Management. Furthermore, the RUP combined a technical process that added value for practitioners -- rather than impeding their ability to do their jobs -- with a macro process that ensured that overall business objectives were met.

Figure 1: RUP Product Development Has Gone Through Three Distinct Phases. *The RUP product has gone through three phases so far. In Phase 1, we created a process focusing on the front-end of the lifecycle; and in Phase 2, we integrated content within Rational to create a consistent process covering the full lifecycle. In Phase 3, we collaborated with a variety of industry leaders to provide in-depth guidance around technology such as leading development platforms.*

To ensure practitioner value, the RUP was also tightly integrated with practitioner tools. Sales soared, and the RUP become the most widely used process for iterative and component-based development. Its success could be attributed to the fact that the RUP was Web-enabled, easy-to-use, and nonintrusive.

Phase 3: 2000-2001

By 2000 the Rational process development team started to look elsewhere for more specialized expertise to address customer's need for content around J2EE, WinDNA, application servers and other technologies. We already had a full-lifecycle process and were looking to add more targeted, specific value for developers. After discussions with many developers, it was obvious that we needed to provide more detailed guidance on developing specific types of applications. We found that developers wanted to know how to develop applications on the J2EE or WinDNA platforms. They wanted to know how to use the RUP to develop e-business applications. But we needed outside help to provide best practices in these areas.

The solution was to collaborate with other companies. We worked with IBM, Sun, and BEA to gain expertise on how best to build applications on the Java platform and build e-business applications. We worked with Microsoft and AIS (Applied Information Sciences) to understand how best to build applications on the WinDNA platform.

Dilemma and Opportunity: The Move Toward Phase 4

RUP sales continued to accelerate, and the rate of adoption was greater than we ever had dreamed possible. But along with our success, we had a problem accommodating the enormous interest the RUP was generating. After one year into Phase 3, we had more partners and customers wanting to collaborate with us on developing new content than we had the bandwidth to handle.

At the same time, two other issues arose. On one hand, RUP users were looking for a variety of very high-level specialized knowledge and wanted Rational to add more content in areas such as commercial, off-the-shelf software development, creative design, content management -- and the list just got longer as we added new content. On the other hand, as we added more content, it became harder for some users to know where to start, given the volume of material now included in the RUP. And the architecture of RUP did not allow us to address both of these problems at the same time.

To resolve this dilemma, we started plans late in 2000 for Phase 4, which would develop the RUP into an industry-wide platform for best practices -- a process that could meet the needs of even the most specialized software development organizations, yet remain accessible to the many types of practitioners working on various phases of the software development lifecycle.

A Closer Look at the Industry-wide Best Practices Platform

The initiative to make the RUP a broadly accepted process platform had the following objectives:

- *Right-sizing:* Allow projects to "right-size" the process they use; that is, allow projects to use "just enough" process. Users should be able to start with a small process, and as project needs expand, to grow the process to address those needs.
- *Process guidance:* Make a wide array of process guidance available to RUP users, including specific guidance on how to develop software for a variety of architectures, target devices, and hardware/software platforms. The best way to achieve this is to make it easy for platform and tool vendors, system integrators, and other industry leaders, to make their know-how available in RUP format.

Today's RUP platform consists of four major components:

1. An infrastructure that allows Rational partners and customers to build and package best practices into process components.
2. A distribution and marketing channel for process components.
3. Deployment mechanisms for process components allowing you to "right-size" the process you use

   .
4. A framework for accessing best practices, which is tightly integrated with practitioner tools.

Let's take a closer look at each of these components.

An Infrastructure for Building and Packaging Best Practices

One of our objectives is to increase the amount of specialized content available to RUP customers. To accomplish this, we are taking advantage of two phenomena that we have observed over the RUP product's history. In the first place, Rational partners typically have an interest in developing specialized content to make their customers more successful in using their tools, or to advance their thought leadership in a certain technology or domain of expertise. Normally, it is in their interest to distribute this content at no cost, but some may want to charge for it. Second, RUP customers typically want to develop specialized content to address their specific, internal needs. Some of this content may be of special interest only to a given company; but in some cases the content may be of interest to other companies as well, and they may choose to distribute it to RUP users outside the company either free of charge or for a fee.

Componentizing the RUP

To enable partners and customers to develop RUP content independent from one another, we developed a component-based architecture for the RUP. To describe this architecture, we needed to introduce some new concepts:

- A ***RUP Base*** contains process elements that are likely to be useful for all projects, and which capture some of the fundamental principles of the RUP, such as iterative, use-case driven, and architecture-centric development. A RUP Base can be extended with RUP Plug-ins.
- A ***RUP Plug-in*** is the deployable unit for one or several process components that can be readily "dropped" onto a RUP Base to extend it.
- The ***RUP Process Framework*** is an extensible architecture for process definition. It provides:
    - A systematic means for decomposing and capturing process knowledge into well-defined (typed) process-definition elements, such as role, artifact, activity, guidelines, concepts, and so on.
    - A set of rules and policies to assemble and organize these elements into a cohesive customized process.
    - An extensible process template to serve as a basis for process authoring.

  The architecture of the RUP process framework is based on the Software Process Engineering Meta-model (SPEM, an OMG specification and UML domain model[6]), and its extension is supported by a set of tools: Rational Process Workbench® and RUP Builder. It includes a RUP Base.
- The ***RUP process platform*** comprises the RUP Process Framework, the supporting toolset, and a set of ready-made process plug-ins.
- ***Core RUP*** consists of the RUP Process Framework plus the RUP Plug-ins that are fundamental to software engineering and within the expertise of Rational Software. Core RUP is what we typically ship on the RUP product CD.

This division of the RUP into a *RUP Process Framework* (containing a *RUP Base*) and a number of *RUP Plug-ins* makes the product far more flexible. The very

nature of component-based architecture, along with the well-defined guidelines provided by the RUP Process Framework, allows a variety of companies to develop RUP content independently of each other.

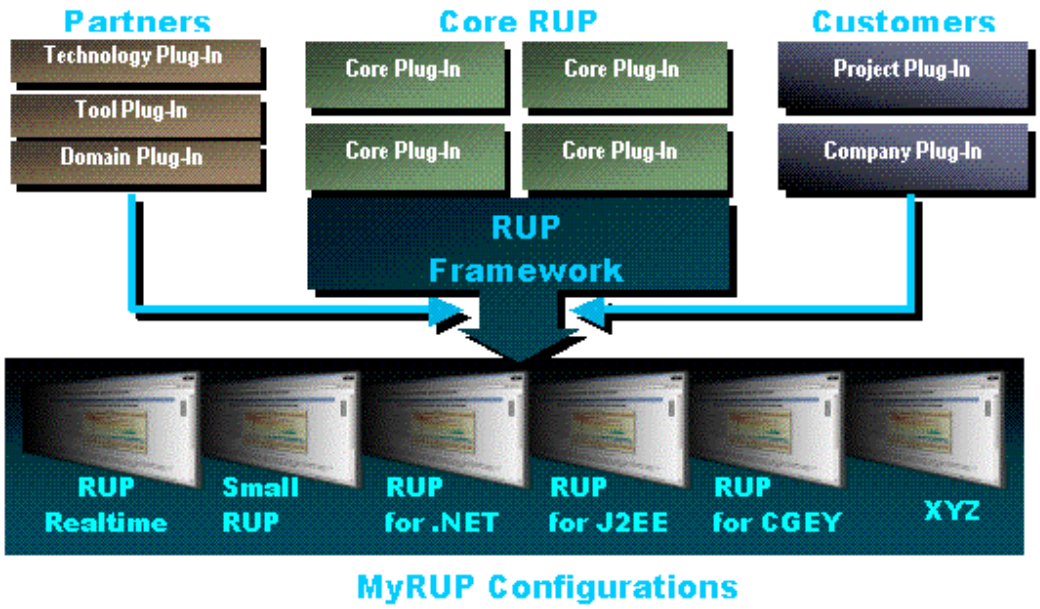


Figure 2: RUP's Component-based Architecture Facilitates Independent Plug-in Development. *The component-based architecture of the RUP allows partners and customers to independently package their know-how into plug-ins. Customers can now choose from a wide variety of plug-ins and deploy those that are appropriate for their project.*

As shown in Figure 2, partners can now package their know-how of a certain technology, tool, or domain into a RUP Plug-in. Customers can also take advantage of this technology to produce plug-ins that are specific for a project, a division, or their entire organization, thus further leveraging their investments in .NET, J2EE, or other development and deployment platforms. This allows companies with large numbers of software developers to put their vast know-how at the fingertips of all their software engineers.

Later we will see how a RUP user can determine which plug-ins to deploy for a certain project.

An Industry Standard for Process Authoring

About two years ago, we started to work with IBM on a standard for process authoring. The starting point was the meta-model for RUP and IBM Global Service's processes. We later brought this work to the Object Management Group (OMG), a standards body that owns, among other things, the Unified Modeling Language (UML). Through close collaboration with many other companies, this work evolved by June 2001 into the SPEM OMG standard for process authoring noted earlier.

SPEM is supported by the RUP and Rational Process Workbench, our own process-authoring tool. SPEM provides an intellectual foundation for the capabilities of Rational Process Workbench. And as an industry standard, SPEM offers users a common, underlying structure and terminology for processes, making it easier for people to build RUP Plug-ins.

Building RUP Plug-ins

With SPEM as the theoretical foundation and Rational Process Workbench as the process-authoring tool, partners and customers can now package their know-how into RUP Plug-ins. Rational Process Workbench is a Rational Rose add-in that allows you to take advantage of the power of UML and visual modeling to model the various process elements you have. New capabilities in the Process Workbench allow you to define how your process elements should extend or override process elements in the RUP Framework. The Process Workbench also allows you to package your process elements, and the definition of how they extend and override process elements in the RUP Framework, into a RUP Plug-in.

Process engineers who use Rational Process Workbench appreciate its power, but it should be made clear that process authoring and using the Process Workbench require a certain level of expertise. You need to be familiar with object-oriented modeling, Rational Rose, and the RUP to build a RUP Plug-in.

To assist the process engineer in building RUP Plug-ins, we offer tutorials and training material, as well as referrals to partners willing to build specialized Plug-ins for a fee.[7]

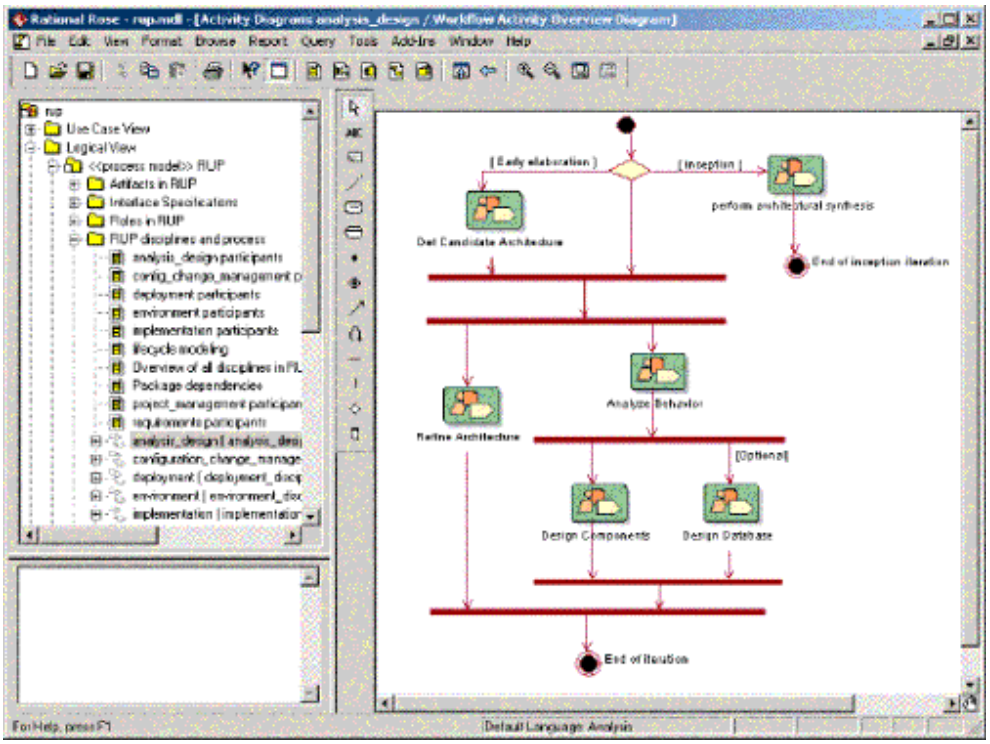


Figure 3: Rational Process Workbench Exploits the Power of Visual Modeling. *Rational Process Workbench is a process-authoring tool built as a Rational Rose add-in. It brings the power of visual modeling and UML to process authoring, and allows you to build RUP Plug-ins.*

Distribution and Marketing Channel for Process Components: The RUP Exchange

The Rational Developer Network[8] is an online community and knowledge resource for Rational customers worldwide. One of the Rational Developer Network's services is the RUP Exchange, which functions as a distribution channel for RUP Plug-ins, as well as an information source for building Plug-ins.

The RUP Exchange contains a hyperlinked list of currently available RUP Plug-ins. This list is continually updated as Plug-ins become available from Rational, our

partners, and our customers. Clicking on a link to a Plug-in gives you its description and allows you to download it to your desktop.

Developers can find the latest guidelines for designing and building Plug-ins on the RUP Exchange. Once a Plug-in is built, it can be included on the RUP Exchange by using the RUP Plug-in upload page. Making your RUP Plug-ins available to other RUP users can be of significant value to the RUP community, and it can give you and your organization exposure as experts in a certain technology, tool, or domain.

The RUP Exchange also lists companies that will help you package your knowledge into a RUP Plug-in, for a fee. You can get help in a wide array of areas -- for example, using Rational Process Workbench for modeling your Plug-in, or transforming Word files into the Plug-in format.

You can find the RUP Exchange in the RUP Knowledge Center within the Rational Developer Network. There you will find RUP-related discussions, various software engineering articles, and other material of interest to project members.

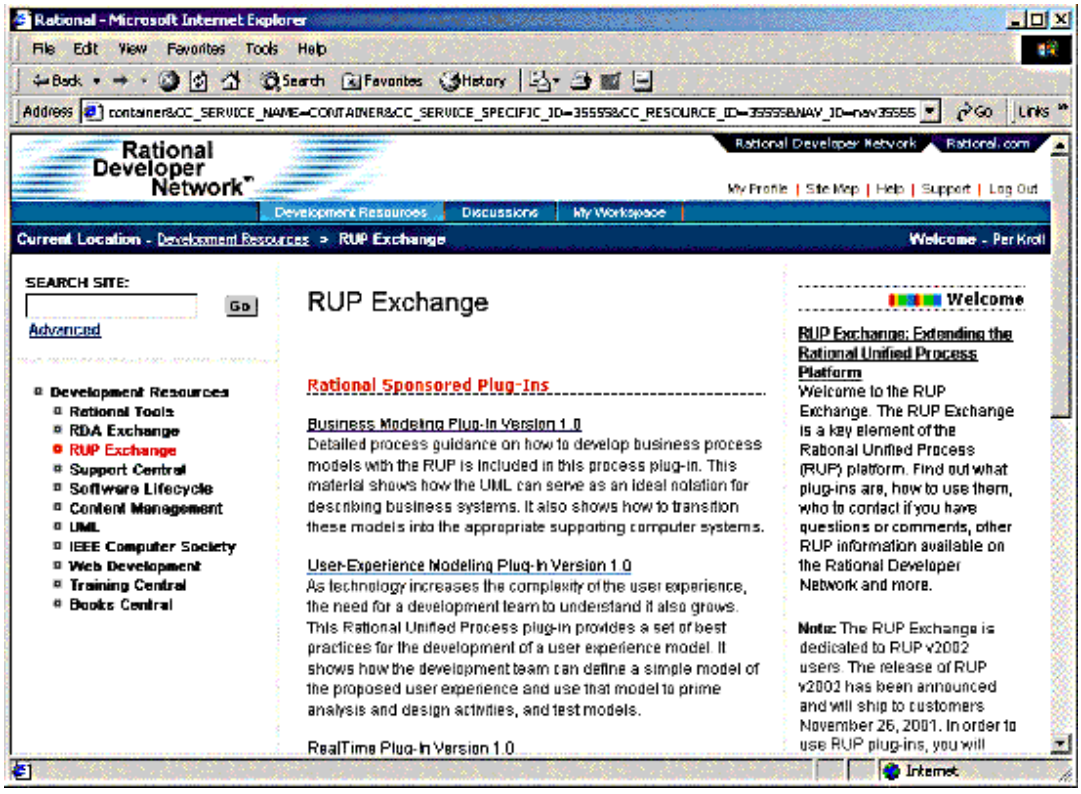


Figure 4: The RUP Exchange on the Rational Developer Network Provides The Latest Information about RUP Plug-ins. *The RUP Exchange allows you to find new or updated Plug-ins from Rational, and from Rational partners and customers. You can also find guidelines on building Plug-ins, and you can upload Plug-ins you have built yourself.*

Deployment Mechanisms for Process Components: RUP Builder

With RUP Builder, a new tool in the RUP product, you can select which Plug-ins to deploy for your project. First, let's introduce some new concepts:

- A ***RUP Configuration*** consists of a RUP Base, and a selected set of RUP Plug-ins. RUP Configurations are assembled using RUP Builder.
- A ***RUP Web site*** is a generated RUP Configuration.

The RUP ships with RUP Builder, the RUP Framework (which contains a RUP Base), and a number of Plug-ins. And as Rational, our partners, and our customers place more Plug-ins on the RUP Exchange, you can download these and use them in your RUP Builder.

RUP Builder organizes your Plug-ins into "configurations." It ships with a number of predefined configurations, and you can create additional configurations as you require. Once you have defined which Plug-ins belong to a configuration, RUP Builder validates that the selected Plug-ins are compatible. Once you have selected a set of Plug-ins that do not conflict with each other, RUP Builder allows you to generate a RUP Web site from your configuration. This Web site has the look and feel of the RUP as you know it today. It has the tree control, navigation buttons, and search capabilities you are used to, but the actual content is based on the particular Plug-ins you have selected for your configuration.

A key feature of RUP Builder is that it allows you to right-size your process. This means that you can choose a small, medium size, or large process, by selecting and de-selecting Plug-ins. Rational is committed to allowing your project to define the specific process you need, and we strive for continuous improvements in this area.

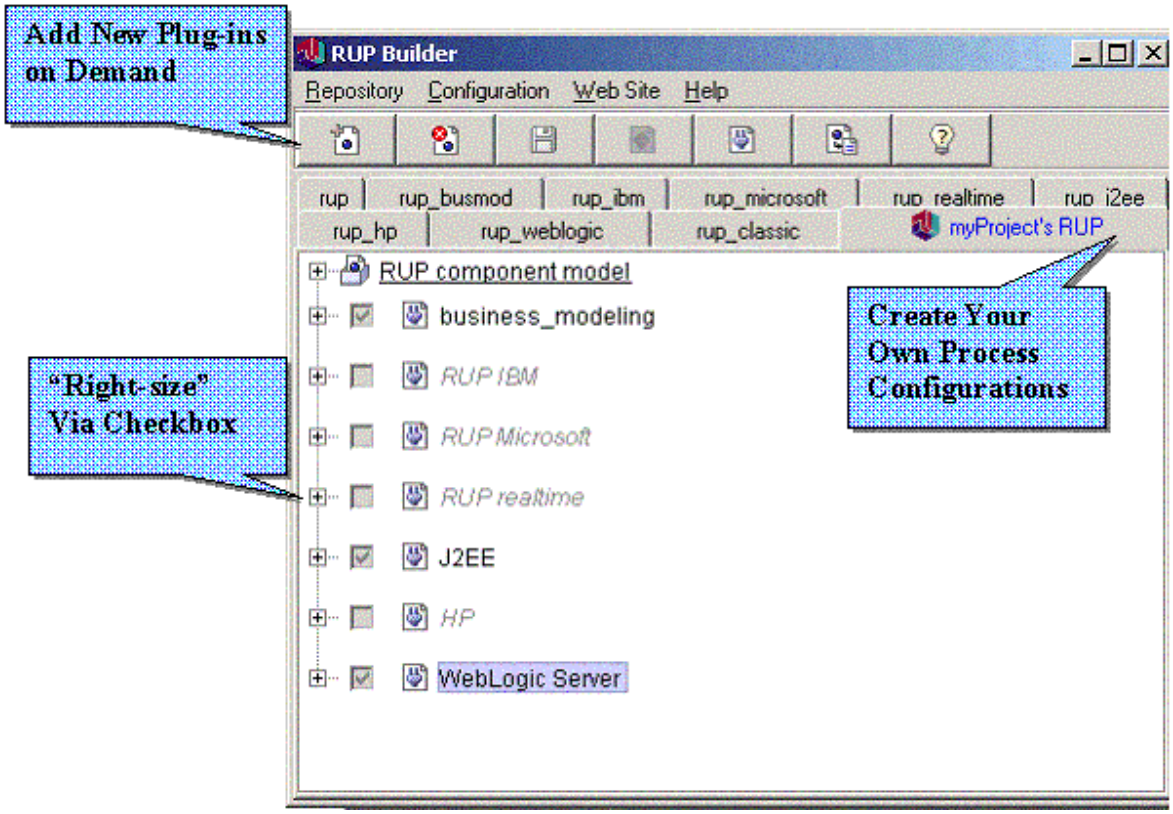


Figure 5: RUP Builder Allows "Right-sizing" to Customize Your Process. *RUP Builder allows you to "right-size" the process you use in a project, by making your process smaller or larger. This is done by selecting which Plug-ins should be included in a RUP Configuration, and generating a RUP Web site from your Configuration.*

The generated Web site, or instance of RUP, can be further customized to address specific project needs. This is typically done by producing a *Development Case*, which guides team members as to which parts of the RUP to use and how to use them. More guidelines for customizing the RUP are in the Process Engineering Toolkit included in the Environment discipline of the RUP.

Accessing Best Practices

All of the capabilities described above are only valuable if analysts, developers, testers, database administrators, configuration managers, project managers, deployment managers, and other team members can effectively use the know-how captured in the Plug-ins. Therefore, the last piece in our improved process framework is the new capabilities of the generated RUP Web site.

You can view the RUP Web site through your favorite Web browser. You will find the following features that help you access pertinent information from the RUP knowledge base:

- A *Getting Started tour* to acquaint you with the RUP product.
- A *search engine and index* to make it easy to find information.
- A *role-based presentation of material* to allow you to rapidly access all relevant parts of the process for the responsibilities of that role.
- *Graphical navigation and extensive hyperlinking* that allow you to drill down into details in the areas of most interest to you.

RUP Integration with Tools: Tool Mentors and Extended Help

The bulk of the RUP is tool independent, but at the end of the day, practitioners need to understand how to implement the process with the tools at hand. *Tool mentors* provide step-by-step guidelines for implementing the various RUP activities using the tools on your desktop. Tool mentors describe which menus to select, what to enter in dialog boxes, and how to draw diagrams to accomplish the tasks specified in the RUP.

Tool mentors are available for Rational tools, as well as for partners' tools such as IBM WebSphere Application Server and BEA WebLogic. Customers and partners can write additional tool mentors, and tool mentors can be included in RUP plug-ins, as can any process element in RUP.

Extended Help provides context-sensitive process guidance within the various tools. For example, if you are trying to use the Class Diagram editor in Rational Rose and you do not know what to do next, you can open "Extended Help" from the Rose tools menu to get a list of the most relevant topics within the RUP, depending on the context -- in this case, a Class Diagram in Rose.

Extended Help is available not only from Rational tools, but it can also be integrated with any tools through a simple API, allowing partners to integrate their tools with the Rational Unified Process. This further promotes the notion of the Rational Unified Process platform assisting you in developing software using a variety of tools, on a variety of hardware and software platforms.

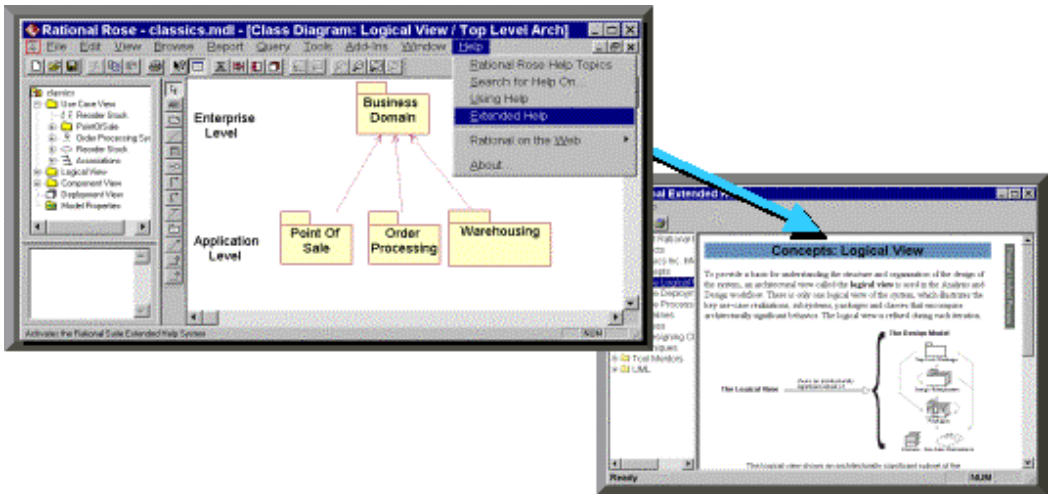


Figure 6: RUP Context-Sensitive Extended Help. *Extended Help provides context-sensitive help from the tools you are using. When launched, it presents a list of the most relevant topics in the RUP.*

The combination of Tool Mentors and Extended Help provides a powerful two-way integration between the RUP and the tools at your desktop. This integration helps practitioners make more effective use of their tools, allowing them to get more value out of their tool investment, and facilitating effective implementation of the process.

Conclusion

The Rational Unified Process is an industry-wide platform for software best practices. It allows Rational, its partners, and its customers a better way to package their know-how into process components, encapsulate them as RUP Plug-ins, and distribute them through the RUP Exchange on the Rational Developer Network. Plug-ins currently provide content from IBM, Microsoft, BEA, Sun, HP, and other companies.

RUP users can find out about, and download, RUP Plug-ins from the RUP Exchange on the Rational Developer Network.[9] Using RUP Builder, they can "right-size" the process they use for their specific project by selecting the Plug-ins they want to use, and based on that selection they can generate a project-specific RUP Web site. The know-how captured in Plug-ins is now easily accessible for the practitioner through a Web browser. Extended Help and Tool Mentors provide a two-way integration with the tools enabling effective implementation of the process.

Notes

[1] For a review of the waterfall approach, see "Going Over the Waterfall with the RUP" by Philippe Kruchten in the September 2001 issue of *The Rational Edge*: http://www.therationaledge.com/content/sep_01/t_waterfall_pk.html

[2] The Objectory Process was developed in Sweden with Ivar Jacobson as a principal author.

[3] The Rational Approach was developed 1985-95 with Grady Booch, Philippe Kruchten, and Walker Royce as principal authors. It represents an evolution of Barry Boehme's "spiral" model for iterative software development.

[4] The requirements content was developed by Requisite Inc. with Dean Leffingwell as principal author.

[5] The SQA Process was developed by SQA Inc.

[6] See OMG Document ad/01-06-05 for more info: http://cgi.omg.org/cgi-bin/doc?ad/01-06-05

[7] More information about this can be found at RUP Exchange within the Rational Developer Network. See below.

[8] www.rational.net (registration required for access)

[9] Ibid.

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

Business and Technology Drivers

Several industry trends have driven the development of Rational's industry-wide platform for best practices. Some of the most influential are:

- Rapid introduction of new technology and evolution of existing technology.
- Standardization and commercialization of tools, methods, and process.
- The trend by consulting companies to no longer include a proprietary process as a competitive differentiator.
- Industry skepticism regarding traditional and heavy-weight processes that are dogmatic and manager focused.

These trends are discussed in the following sections.

Rapid Evolution of Technology

It is very difficult for developers to keep up with constantly evolving technology, and lack of knowledge is becoming a major impediment for the adoption of new technology. To address this issue, vendors have increased their focus on guidance and best practices in the marketplace, with the goal of effectively distributing their advanced tools and technologies onto a developer's desktop. Examples of such solutions include MSDN, Oracle Technology Network, and Vignette Global Marketplace. The goal is to put the knowledge that developers need just a mouse click away on their desktop.

Standardization and Commercialization of Tools, Methods, and Process

Over the last 30 years, the software engineering industry has continuously moved toward standardization and commercialization of technology and knowledge. This meant that in the 1960s and 1970s, commercial compilers become available; in the 1980s, CASE tools and databases; and in the 1990s, advanced configuration management systems and IDEs. In the mid 90s, we also saw standardization in the methods and modeling language area. The Unified Modeling Language (UML) was originally developed by Rational and its partners, and later adopted and managed by OMG as an industry standard. By the late 90s, companies were ready for standardization of process, and many companies abandoned their homegrown process and began looking for a commercially available one. This allowed the large investments necessary to build an enterprise-wide process, such as the Rational Unified Process, to be shared by many companies. Companies that have standardized on the RUP, for example, include CGE&Y and Merrill Lynch.

Consulting Companies Less Reliant on Process as a Competitive Differentiator

Similarly, while standardization has resulted from the commercial availability of computing technology, the competitive differentiation based on proprietary processes that consulting companies once relied on has also become much less

significant as a business driver. Process used to be a selling factor for any consulting company worth its name, but as customers got tired of project overruns and the poor quality of delivered applications, they started to demand some assurance that the practices used by system integrators were proven. Since it is extremely difficult to assess the applicability and value of a homegrown process, customers started to demand commercially available processes with a proven track record, and which could be evaluated more easily by independent reviewers. As a result, there has been a rapid move among system integrators away from homegrown processes to commercially available processes. The RUP has been the process of choice for many of these companies, including CGE&Y, Deloitte Consulting, and IconMedialab.

Industry Skepticism Regarding Existing Processes

At the same time, the software development industry became skeptical toward traditional and heavyweight processes that are prescriptive and have a strong primary focus on the needs of managers. These processes typically promoted a "waterfall"[1] sequence of development, functional decomposition, and a document-centric approach. These processes had been popular in the late 80s and the 90s, and proved to be ineffective for several reasons:

- *Processes were of little value to developers.* Since the guidelines primarily focused on describing *what* should be done, and not *how* it should be done, developers saw little value in them. Since it did not help them do a better job, developers resented this type of process.
- *The waterfall approach did not allow effective risk management.* Even worse, since these processes typically followed a waterfall approach, which is ineffective in addressing key risks early in the project (regardless of whether the risks are technical or business related), the project success rate declined as technology evolved and projects became increasing complex and risky. The industry was ready for a change.
- *The process was hard to access.* Processes were often published in thick binders, and the content was not integrated with desktop tools. This made it hard to find the information being sought, so process binders simply collected dust.
- *The process did not focus on delivering value to customers and other stakeholders.* The waterfall-based and dogmatic nature of the processes focused on many artifacts, on heavy documents and activities that did not help deliver real business value to end users and other stakeholders.

The Emergence and Evolution of the RUP

It was in this business climate that the Rational Unified Process gained popularity. From a product development standpoint, the RUP has gone through two complete product phases and is now moving into its third phase (see Figure 1).

Phase 1: 1996-1999

The RUP process development team[2] was launched in February 1996. It was

essential that the process should be iterative, architecture-centric, and use-case driven, since these were some of the main best practices that Rational had identified from its work with its customers and partners. Because Rational has always also had an extremely strong focus on the needs of the practitioner, the RUP process development team believed that the process had to be nonintrusive to the practitioner, and that the average analyst, developer, and tester had to find it was easier to do their jobs with the RUP than without it. These benefits may seem obvious, but prior to the RUP's introduction, most processes could not be described in these terms. The RUP team also realized that the technical process had to be anchored by a solid project management (macro) process to guarantee project success. The macro process needed to ensure that risks were addressed early, that projects were run cost effectively, and that overall business objectives were met.

Having set these objectives, we then took on the task of harvesting and documenting Rational's know-how into one consistent knowledge base during the years 1996 through 1999. Rational already had a (mainly unpublished) process called the Rational Approach, and through acquisitions of other companies we had gained access to the Objectory Process[3] as well as additional processes centered on requirements management, testing, and other areas.

By 1999, the Rational Unified Process covered the full lifecycle, from Business Modeling through Deployment, as well as areas such as Configuration Management and Project Management. Furthermore, the RUP combined a technical process that added value for practitioners -- rather than impeding their ability to do their jobs -- with a macro process that ensured that overall business objectives were met.

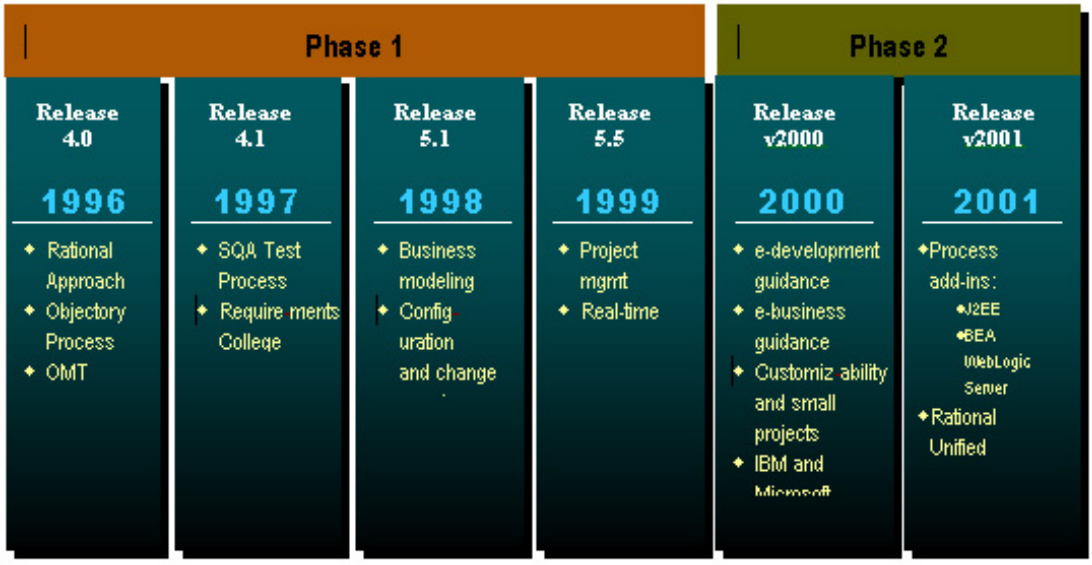


Figure 1. RUP Product Development Has Gone Through Two Distinct Phases. *In Phase 1, we integrated content within Rational to create a consistent process covering the full lifecycle. In Phase 2, we collaborated with a variety of industry leaders to provide in-depth guidance around technology such as leading development platforms.*

To ensure practitioner value, the RUP was also tightly integrated with practitioner tools. Sales soared, and the RUP become the most widely used process for iterative and component-based development. Its success could be attributed to the fact that the RUP was Web-enabled, easy-to- use, and

nonintrusive.

Phase 2: 2000-2001

By 2000, the Rational process development team had captured all its internal know-how and had started to look elsewhere for more specialized expertise. We already had a full-lifecycle process and were looking to add more targeted, specific value for developers. After discussions with many developers, it was obvious that we needed to provide more detailed guidance on developing specific types of applications. We found that developers wanted to know how to develop applications on the J2EE or WinDNA platforms. They wanted to know how to use the RUP to develop e-business applications. But we needed outside help to provide best practices in these areas.

The solution was to collaborate with other companies. We worked with IBM, Sun, and BEA to gain expertise on how best to build applications on the Java platform and build e-business applications. We worked with Microsoft and AIS (Applied Information Sciences) to understand how best to build applications on the WinDNA platform.

Dilemma and Opportunity: The Move Toward Phase 3

RUP sales continued to accelerate, and the rate of adoption was greater than we ever had dreamed possible. But along with our success, we had a problem accommodating the enormous interest the RUP was generating. After one year into Phase 2, we had more partners and customers wanting to collaborate with us on developing new content than we had the bandwidth to handle.

At the same time, two other issues arose. On one hand, RUP users were looking for a variety of very high-level specialized knowledge and wanted Rational to add more content in areas such as commercial, off-the-shelf software development, creative design, content management -- and the list just got longer as we added new content. On the other hand, as we added more content, it became harder for some users to know where to start, given the volume of material now included in the RUP. And the architecture of RUP did not allow us to address both of these problems at the same time.

To resolve this dilemma, we started plans late in 2000 for Phase 3, which would develop the RUP into an industry-wide platform for best practices -- a process that could meet the needs of even the most specialized software development organizations, yet remain accessible to the many types of practitioners working on various phases of the software development life cycle.

A Closer Look at the Industry-wide Best Practices Platform

The initiative to make the RUP a broadly accepted process platform had the following objectives:

- *Right-sizing*: Allow projects to "right-size" the process they use; that is, allow projects to use "just enough" process. Users should be able to start with a small process, and as project needs expand, to grow the

process to address those needs.

- *Process guidance*: Make a wide array of process guidance available to RUP users, including specific guidance on how to develop software for a variety of architectures, target devices, and hardware/software platforms. The best way to achieve this is to make it easy for platform and tool vendors, system integrators, and other industry leaders, to make their know-how available in RUP format.

Today's RUP platform consists of four major components:

1. An infrastructure that allows Rational partners and customers to build and package best practices into process components
2. A distribution and marketing channel for process components
3. Deployment mechanisms for process components allowing you to "right-size" the process you use
4. A framework for accessing best practices, and which is tightly integrated with practitioner tools

Let's take a closer look at each of these components.

An Infrastructure for Building and Packaging Best Practices

One of our objectives is to increase the amount of specialized content available to RUP customers. To accomplish this, we are taking advantage of two phenomena that we have observed over the RUP's six-year history. In the first place, Rational partners typically have an interest in developing specialized content to make their customers more successful in using their tools, or to advance their thought leadership in a certain technology or domain of expertise. Normally, it is in their interest to distribute this content at no cost, but some may want to charge for it. Second, RUP customers typically want to develop specialized content to address their specific, internal needs. Some of this content may be of special interest only to a given company; but in some cases the content may be of interest to other companies as well, and they may choose to distribute it to RUP users outside the company either free of charge or for a fee.

Componentizing the RUP

To enable partners and customers to develop RUP content independent from one another, we developed a component-based architecture for the RUP. To describe this architecture, we needed to introduce some new concepts:

- A ***RUP Base*** contains process elements that are likely to be useful for all projects, and which capture some of the fundamental principles of the RUP, such as iterative, use-case driven, and architecture-centric development. A RUP Base can be extended with RUP Plug-ins.
- A ***RUP Plug-in*** is the deployable unit for one or several process components that can be readily "dropped" onto a RUP Base to extend it.
- The ***RUP Process Framework*** is an extensible architecture for process

definition. It provides:

- A systematic means for decomposing and capturing process knowledge into well-defined (typed) process-definition elements, such as role, artifact, activity, guidelines, concepts, and so on.
- A set of rules and policies to assemble and organize these elements into a cohesive customized process.
- An extensible process template to serve as a basis for process authoring.

The architecture of the RUP process framework is based on the Software Process Engineering Meta-model (SPEM, an OMG specification and UML domain model[4]), and its extension is supported by a set of tools: Rational Process WorkbenchÒ and RUP Builder. It includes a RUP Base.

- The ***RUP process platform*** comprises the RUP Process Framework, the supporting toolset, and a set of ready-made process plug-ins.
- ***Core RUP*** consists of the RUP Process Framework plus the RUP Plug-ins that are fundamental to software engineering and within the expertise of Rational Software. Core RUP is what we typically ship on the RUP product CD.

This division of the RUP into a *RUP Process Framework* (containing a *RUP Base*) and a number of *RUP Plug-ins* makes the product far more flexible. The very nature of component-based architecture, along with the well-defined guidelines provided by the RUP Process Framework, allows a variety of companies to develop RUP content independently of each other.

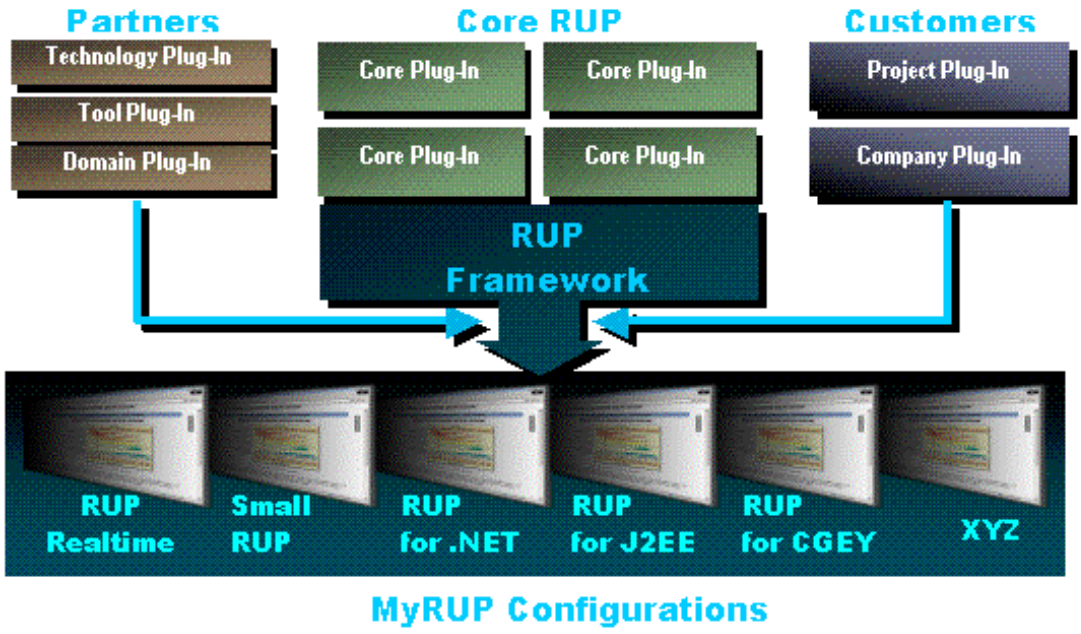


Figure 2. RUP's Component-based Architecture Facilitates Independent Plug-in Development. *The component-based architecture of the RUP allows partners and customers to independently package their know-how into plug-ins. Customers can now choose from a wide variety of plug-ins and deploy those that are appropriate for their project.*

As shown in Figure 2, partners can now package their know-how of a certain technology, tool, or domain into a RUP Plug-in. Customers can also take advantage of this technology to produce plug-ins that are specific for a project, a division, or their entire organization, thus further leveraging their investments in .NET, J2EE, or other development and deployment platforms. This allows companies with large numbers of software developers to put their vast know-how at the fingertips of all their software engineers.

Later we will see how a RUP user can determine which plug-ins to deploy for a certain project.

An Industry Standard for Process Authoring

About two years ago, we started to work with IBM on a standard for process authoring. The starting point was the meta-model for RUP and IBM Global Service's processes. We later brought this work to the Object Management Group (OMG), a standards body that owns, among other things, the Unified Modeling Language (UML). Through close collaboration with many other companies, this work evolved by June 2001 into the SPEM OMG standard for process authoring noted earlier.

SPEM is supported by the RUP and Rational Process Workbench, our own process-authoring tool. SPEM provides an intellectual foundation for the capabilities of Rational Process Workbench. And as an industry standard, SPEM offers users a common, underlying structure and terminology for processes, making it easier for people to build RUP Plug-ins.

Building RUP Plug-ins

With SPEM as the theoretical foundation and Rational Process Workbench as the process-authoring tool, partners and customers can now package their know-how into RUP Plug-ins. Rational Process Workbench is a Rational Rose add-in that allows you to take advantage of the power of UML and visual modeling to model the various process elements you have. New capabilities in the Process Workbench allow you to define how your process elements should extend or override process elements in the RUP Framework. The Process Workbench also allows you to package your process elements, and the definition of how they extend and override process elements in the RUP Framework, into a RUP Plug-in.

Process engineers who use Rational Process Workbench appreciate its power, but it should be made clear that process authoring and using the Process Workbench require a certain level of expertise. You need to be familiar with Object-Oriented modeling, Rational Rose, and the RUP to build a RUP Plug-in.

To assist the process engineer in building RUP Plug-ins, we offer tutorials and training material, as well as referrals to partners willing to build specialized Plug-ins for a fee.[5]

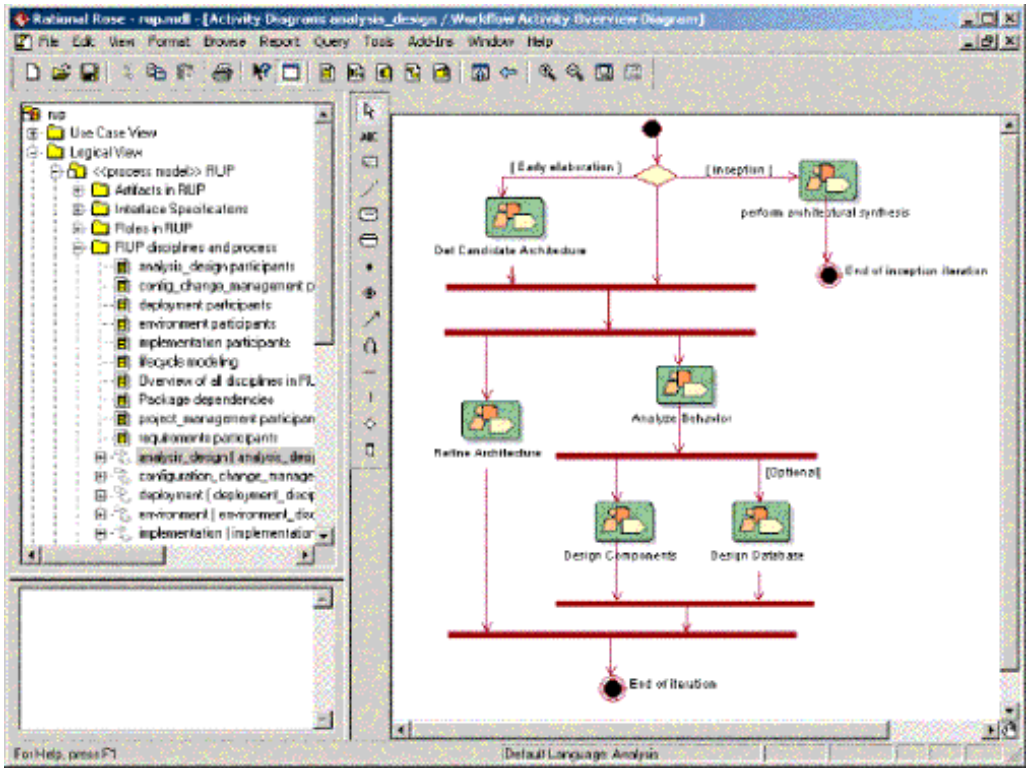


Figure 3. Rational Process Workbench Exploits the Power of Visual Modeling. *Rational Process Workbench is a process-authoring tool built as a Rational Rose add-in. It brings the power of visual modeling and UML to process authoring, and allows you to build RUP Plug-ins.*

Distribution and Marketing Channel for Process Components: The RUP Exchange

The Rational Developer Network[6] is an online community and knowledge resource for Rational customers worldwide. One of the Rational Developer Network's services is the RUP Exchange, which functions as a distribution channel for RUP Plug-ins, as well as an information source for building Plug-ins.

The RUP Exchange contains a hyperlinked list of currently available RUP Plug-ins. This list is continually updated as Plug-ins become available from Rational, our partners, and our customers. Clicking on a link to a Plug-in gives you its description and allows you to download it to your desktop.

Developers can find the latest guidelines for designing and building Plug-ins on the RUP Exchange. Once a Plug-in is built, it can be included on the RUP Exchange by using the RUP Plug-in upload page. Making your RUP Plug-ins available to other RUP users can be of significant value to the RUP community, and it can give you and your organization exposure as experts in a certain technology, tool, or domain.

The RUP Exchange also lists companies that will help you package your knowledge into a RUP Plug-in, for a fee. You can get help in a wide array of areas -- for example, using Rational Process Workbench for modeling your Plug-in, or transforming Word files into the Plug-in format.

You can find the RUP Exchange in the RUP Knowledge Center within the Rational Developer Network. There you will find RUP-related discussions,

various software engineering articles, and other material of interest to project members.

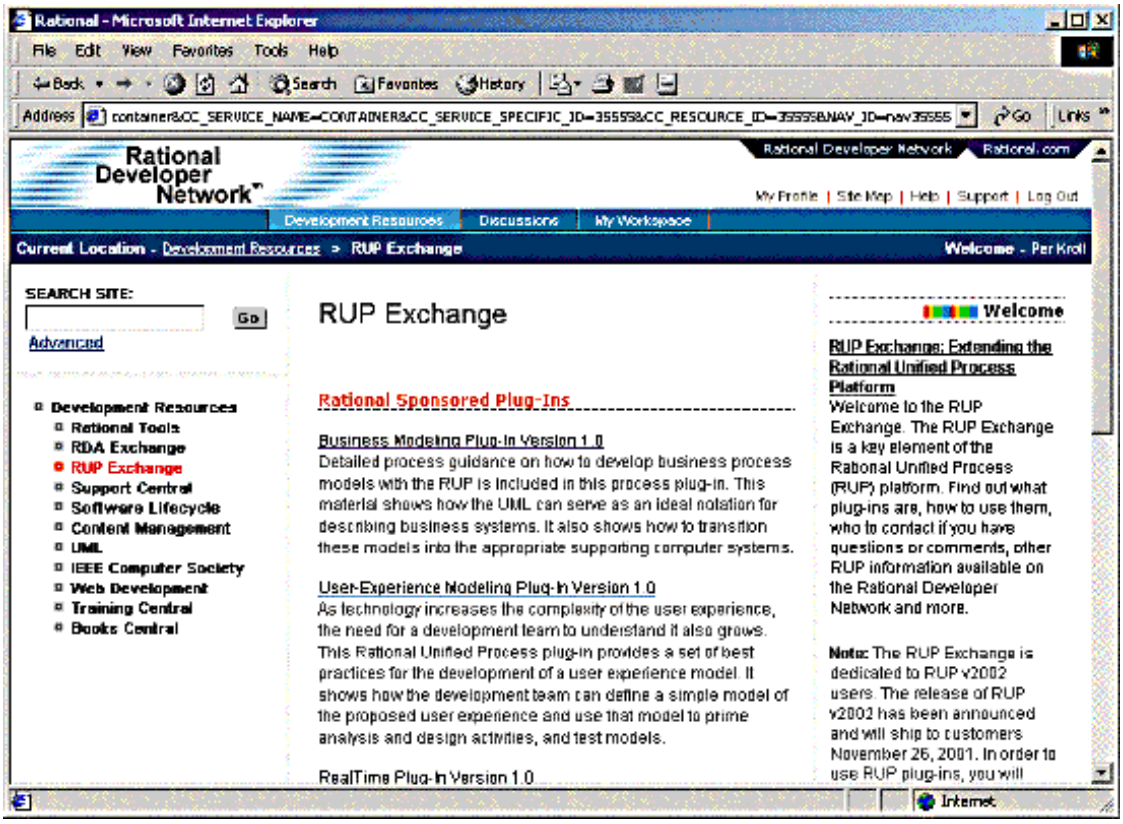


Figure 4. The RUP Exchange on the Rational Developer Network Provides The Latest Information about RUP Plug-ins. *The RUP Exchange allows you to find new or updated Plug-ins from Rational, and from Rational partners and customers. You can also find guidelines on building Plug-ins, and you can upload Plug-ins you have built yourself.*

Deployment Mechanisms for Process Components: RUP Builder

With RUP Builder, a new tool in the RUP product, you can select which Plug-ins to deploy for your project. First, let's introduce some new concepts:

- A ***RUP Configuration*** consists of a RUP Base, and a selected set of RUP Plug-ins. RUP Configurations are assembled using RUP Builder.
- A ***RUP Web site*** is a generated RUP Configuration.

The RUP ships with RUP Builder, the RUP Framework (which contains a RUP Base), and a number of Plug-ins. And as Rational, our partners, and our customers place more Plug-inson the RUP Exchange, you can download these and use them in your RUP Builder.

RUP Builder organizes your Plug-ins into "configurations." It ships with a number of predefined configurations, and you can create additional configurations as you require. Once you have defined which Plug-ins belong to a configuration, RUP Builder validates that the selected pPlug-ins are compatible. Once you have selected a set of Plug-ins that do not conflict with each other, RUP Builder allows you to generate a RUP Web site from your configuration. This Web site has the look and feel of the RUP as you know it today. It has the tree control, navigation buttons, and search capabilities you

are used to, but the actual content is based on the particular Plug-ins you have selected for your configuration.

A key feature of RUP Builder is that it allows you to right-size your process. This means that you can choose a small, medium size, or large process, by selecting and de-selecting Plug-ins. Rational is committed to allowing your project to define the specific process you need, and we strive for continuous improvements in this area.

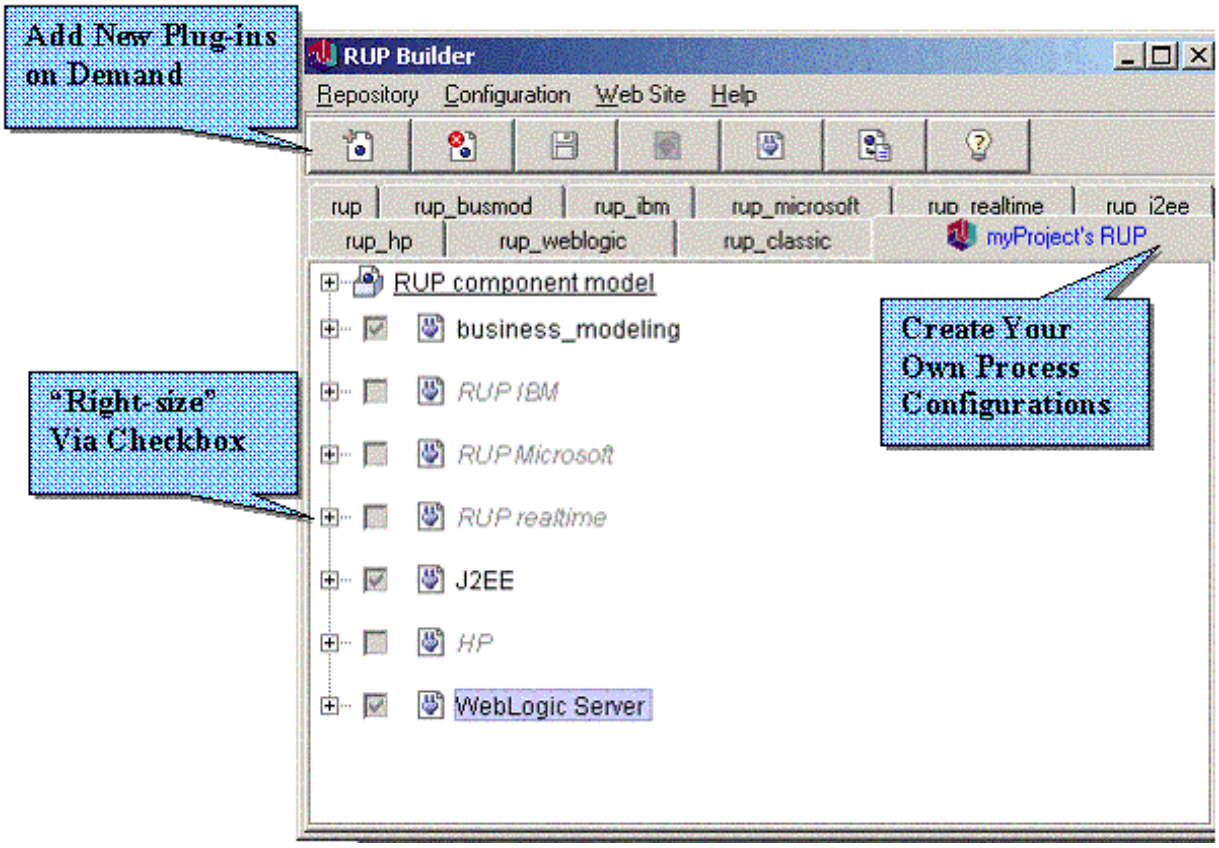


Figure 5. RUP Builder Allows "Right-sizing" to Customize Your Process. *RUP Builder allows you to "right-size" the process you use in a project, by making your process smaller or larger. This is done by selecting which Plug-ins should be included in a RUP Configuration, and generating a RUP Web site from your Configuration.*

The generated Web site, or instance of RUP, can be further customized to address specific project needs. This is typically done by producing a *Development Case*, which guides team members as to which parts of the RUP to use and how to use them. More guidelines for customizing the RUP are in the Process Engineering Toolkit included in the Environment discipline of the RUP.

Accessing Best Practices

All of the capabilities described above are only valuable if analysts, developers, testers, database administrators, configuration managers, project managers, deployment managers, and other team members can effectively use the know-how captured in the Plug-ins. Therefore, the last piece in our improved process framework is the new capabilities of the generated RUP Web site.

You can view the RUP Web site through your favorite Web browser. You will

find the following features that help you access pertinent information from the RUP knowledge base:

- A *Getting Started tour* to acquaint you with the RUP product.
- A *search engine and index* to make it easy to find information.
- A *role-based presentation of material* to allow you to rapidly access all relevant parts of the process for the responsibilities of that role.
- *Graphical navigation and extensive hyperlinking* that allow you to drill down into details in the areas of most interest to you.

RUP Integration With Tools: Tool Mentors and Extended Help

The bulk of the RUP is tool independent, but at the end of the day, practitioners need to understand how to implement the process with the tools at hand. *Tool mentors* provide step-by-step guidelines for implementing the various RUP activities using the tools on your desktop. Tool mentors describe which menus to select, what to enter in dialog boxes, and how to draw diagrams to accomplish the tasks specified in the RUP.

Tool mentors are available for Rational tools, as well as for partners' tools such as IBM WebSphere Application Server and BEA WebLogic. Customers and partners can write additional tool mentors, and tool mentors can be included in RUP plug-ins, as can any process element in RUP.

Extended Help provides context-sensitive process guidance within the various tools. For example, if you are trying to use the Class Diagram editor in Rational Rose and you do not know what to do next, you can open "Extended Help" from the Rose tools menu to get a list of the most relevant topics within the RUP, depending on the context -- in this case, a Class Diagram in Rose.

Extended Help is available not only from Rational tools, but it can also be integrated with any tools through a simple API, allowing partners to integrate their tools with the Rational Unified Process. This further promotes the notion of the Rational Unified Process platform assisting you in developing software using a variety of tools, on a variety of hardware and software platforms.

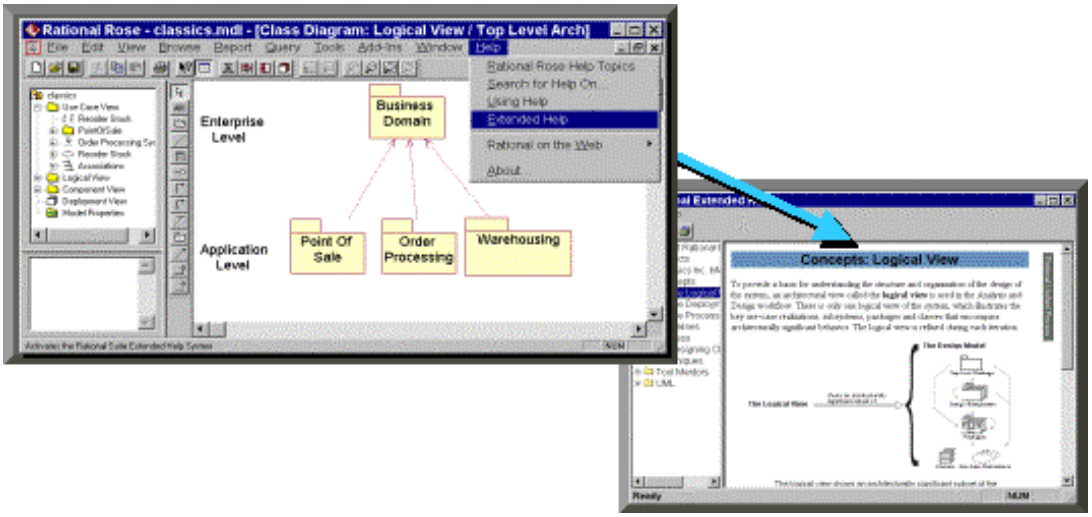


Figure 6. RUP Context-Sensitive Extended Help. *Extended Help provides context-sensitive help from the tools you are using. When launched, it presents a list of the most relevant topics in the RUP.*

The combination of Tool Mentors and Extended Help provides a powerful two-way integration between the RUP and the tools at your desktop. This integration helps practitioners make more effective use of their tools, allowing them to get more value out of their tool investment, and facilitating effective implementation of the process.

Conclusion

The Rational Unified Process is an industry-wide platform for software best practices. It allows Rational, its partners, and its customers a better way to package their know-how into process components, encapsulate them as RUP Plug-ins, and distribute them through the RUP Exchange on the Rational Developer Network. Plug-ins currently provide content from IBM, Microsoft, BEA, Sun, HP, and other companies.

RUP users can find out about, and download, RUP Plug-ins from the RUP Exchange on the Rational Developer Network.[7] Using RUP Builder, they can "right-size" the process they use for their specific project by selecting the Plug-ins they want to use, and based on that selection they can generate a project-specific RUP Web site. The know-how captured in Plug-ins is now easily accessible for the practitioner through a Web browser. Extended Help and Tool Mentors provide a two-way integration with the tools enabling effective implementation of the process.

Notes

[1] For a review of the waterfall approach, see "Going Over the Waterfall with the RUP" by Philippe Kruchten in the September 2001 issue of *The Rational Edge*: http://www.therationaledge.com/content/sep_01/t_waterfall_pk.html

[2] Note that the product was originally called the Rational Objectory Process, and was named the Rational Unified Process in 1998

[3] The Objectory Process was developed 1987 by Objective Systems, founded by Ivar Jacobson.

[4] See OMG Document ad/01-06-05 for more info: http://cgi.omg.org/cgi-bin/doc?ad/01-06-05

[5] More information about this can be found at RUP Exchange within the Rational Developer Network. See below.

[6] www.rational.net (registration required)

[7] Ibid.

For more information on the products or services discussed in this

article, please click here and follow the instructions provided. Thank you!

Unified Software Project Management with Rational ProjectConsole

by **Paul R. Wyrick**
Product Manager for Rational Project Console

Doug Ishigaki
Senior Technical Marketing Engineer

Rational Software

Rational® ProjectConsole, a new Rational tool that's included in the most recent release of Rational Suite®, can provide an entire development team, and all others invested in the development of a software product, with easy access to the information they need to keep a project on track from start to finish. In this article, we provide some guidelines and examples for using ProjectConsole to put an effective and easy-to-implement metrics program in place for a development team.

Although most everyone would agree that it's a great idea to track project metrics throughout the product development lifecycle, in practice, most project teams haven't implemented metrics programs because of the high overhead involved. When a team is operating under tight time and budget constraints, gathering, analyzing, and disseminating data continuously are generally not perceived as high-priority tasks. But unless a team has the power to assess project status and predict trends through reliable, up-to-date metrics, the project will suffer.[1]

The good news is that with Rational ProjectConsole, which comes bundled with all Rational Suite products, you can automate and unify metrics collection, analysis, and display. With a minimal investment of time and

effort on your team's part, ProjectConsole can collect data from the repositories of Rational Suite development tools and point products, as well as from third-party products such as Microsoft Project. It can display real-time results, in an easy-to-interpret graphical form, on a project Web site for access by an entire development team.

In this article, we provide some useful tips on how you can use ProjectConsole to easily put an effective metrics program in place. We describe a set of metrics a development team would find useful in monitoring a project through all the phases of a project developed using the Rational Unified Process®(RUP®) -- from Inception to Elaboration, through Construction, and finally, Transition.

There are more possible objects and attributes to measure in a software development project than we can (or care to) cover here. Most of the examples we show -- which represent just a small sample of the types of metrics ProjectConsole can generate -- are prepackaged and available in the sample metrics database that ships with Rational Suite products.[2]

The metrics charts are designed to help project stakeholders determine how much work remains to be done in each phase of development; below, we'll suggest guidelines on what to look at during each phase of a RUP project and how you might interpret what you see.

When you first start using ProjectConsole to track your project's progress, it's a good idea to start with a small, simple set of metrics. As you get comfortable with the technology, you'll want to add metrics tailored to your project and its changing needs.

Monitoring the Inception Phase

During the Inception phase of a RUP project, the primary objectives are to establish the project's scope and boundary conditions, define the critical use cases, define a candidate architecture, develop cost estimates and a schedule, identify potential risks, and prepare the supporting environment for the project.

Identifying business and system requirements are of paramount importance during Inception. Some of the identified requirements will be high level, and some will include more detailed descriptions. Throughout this phase, new requirements are added and refined. The charts in this section show data collected for the sample project over the first three weeks of an Inception phase scheduled to last one month.

Monitoring Total Requirements

Figure 1 shows the number of sample project requirements collected from RequisitePro in the first three weeks of Inception. Level 1 numbers represent high-level requirements, and Level 0 numbers represent detailed requirements.

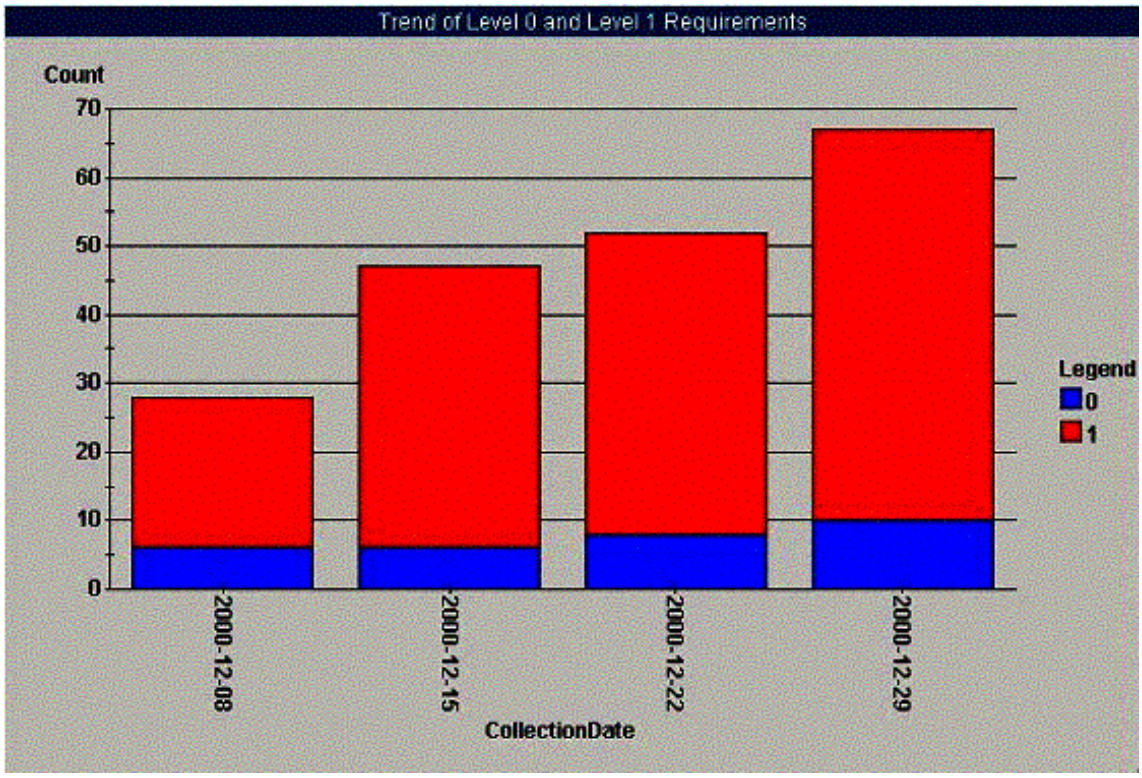


Figure 1: Sample Project Requirements Collected During First Three Weeks of Inception

The metrics depicted here indicate two noticeable trends. The first is that the total number of requirements is increasing. This suggests that project scope has not stabilized yet. Typically, the number of requirements stabilizes toward the end of Inception, so these results indicate that the Inception phase may have to be extended beyond the planned duration of one month.

The second noticeable trend is that, while the number of Level 1 requirements continues to increase, the number of Level 0 (high-level) requirements is stabilizing. This is what we'd expect to see, and it suggests that the project team has gained a good understanding of the project's defined scope. The high-level requirements are in place, with few new ones being added, and the requirements that flesh out the details are still coming in.

Assessing Changes in Requirement Type

Figure 2 shows the number of use-case requirements and features requirements defined for the project over the first three weeks of the Inception phase, based on data collected from Rational RequisitePro. As expected, as the Inception phase progresses and new requirements are identified, the numbers of new use cases and features are increasing.

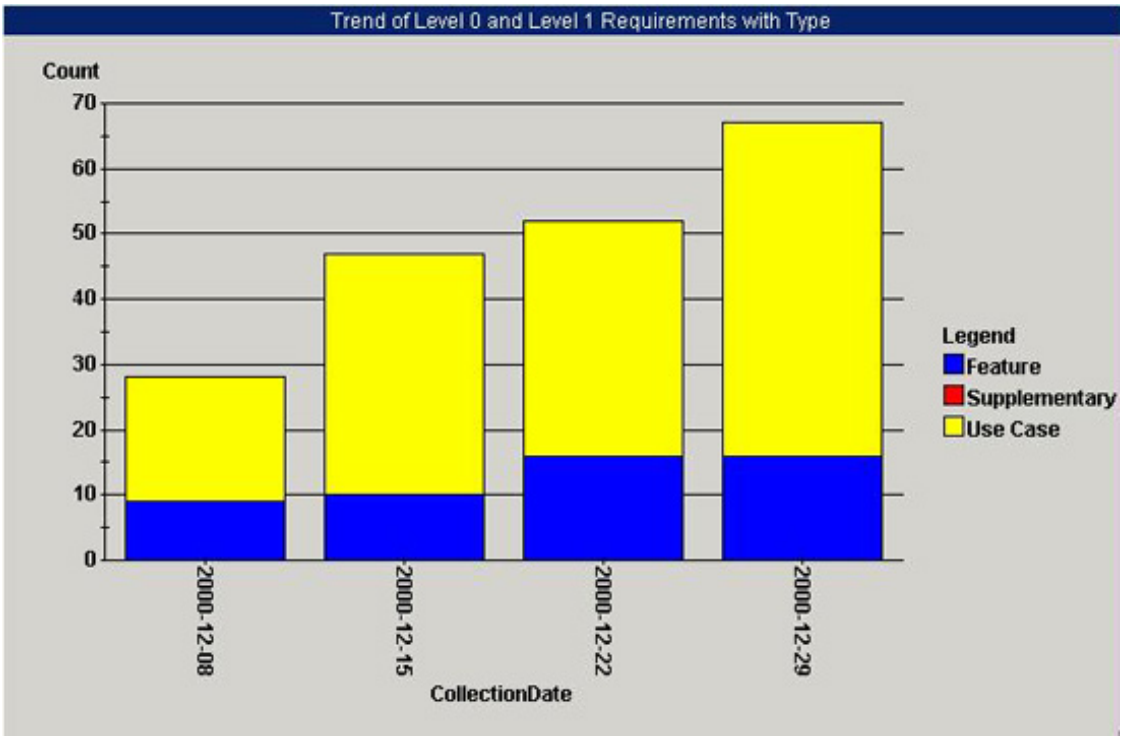


Figure 2: Use-Case and Features Requirements Defined During First Three Weeks of Inception

You can see evidence of two trends in this chart. First, the number of feature-related requirements is stabilizing, further supporting the impression that the team has nearly finished defining the project scope. The second trend is that the number of use-case requirements is increasing. To complete the Inception phase, the project scope must be stable. Although use-case requirements will continue to grow throughout Inception and Elaboration, features should stabilize at the end of the Inception phase.

Assessing Requirement Status

During an iteration in a RUP project, the status of a requirement changes from *proposed* to *approved* to *incorporated* (if approved) and finally, to *validated*. At the beginning of the Inception phase, the ratio of proposed requirements to approved requirements is high. By the end of Inception, however, the situation should be reversed, with approved requirements far outnumbering proposed requirements. Figure 3 shows the number of proposed and approved requirements for the sample project in the first three weeks of Inception.

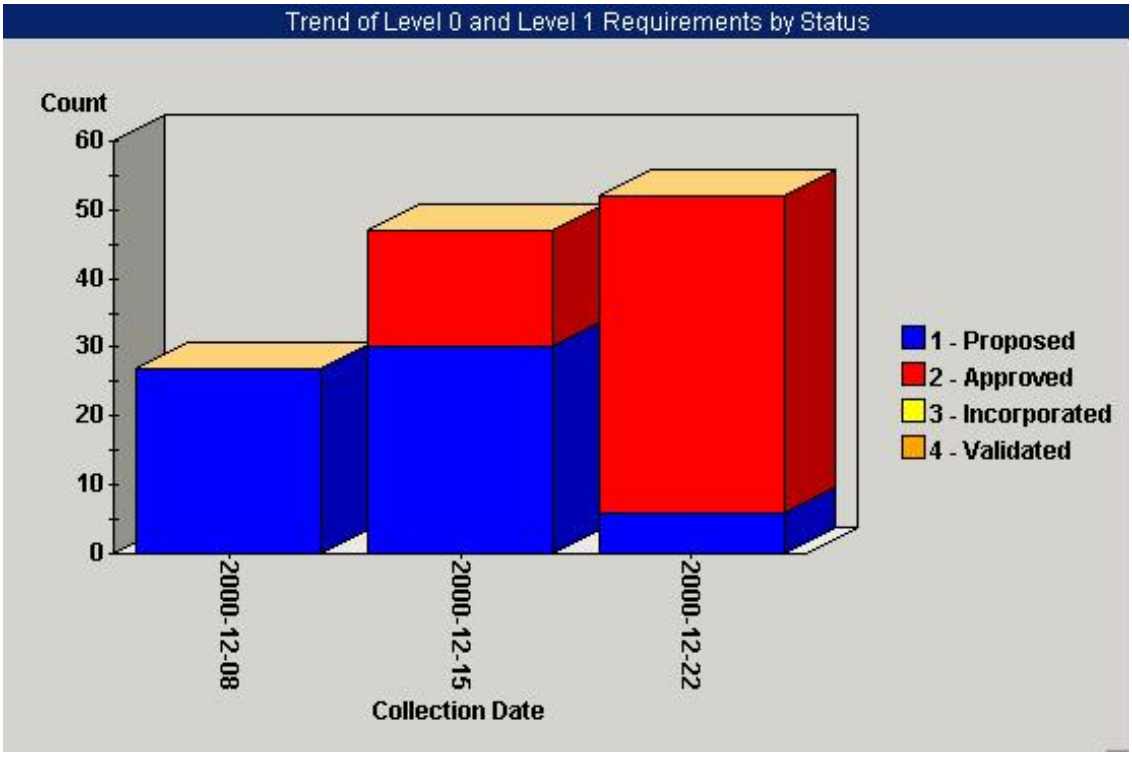


Figure 3: Proposed and Approved Requirements in the First Three Weeks of Inception

Figure 3 shows that, by the end of the third week, the vast majority of proposed requirements have been approved. This is a good sign that the project team is reaching consensus on project requirements. No prototyping has been done up to this point. This may be either a good or a bad thing, depending on the nature of the project. For some projects, the scope can't be identified without some sort of prototyping; others, depending on scope and risk, may not need a prototype.

Assessing Requirements by Priority

Toward the end of the Inception phase, the team should have a clear understanding of the requirements, and information about requirement priority should be fairly well documented. Figure 4 shows the total number of requirements, based on their assigned project priority (1, 2, or 3).

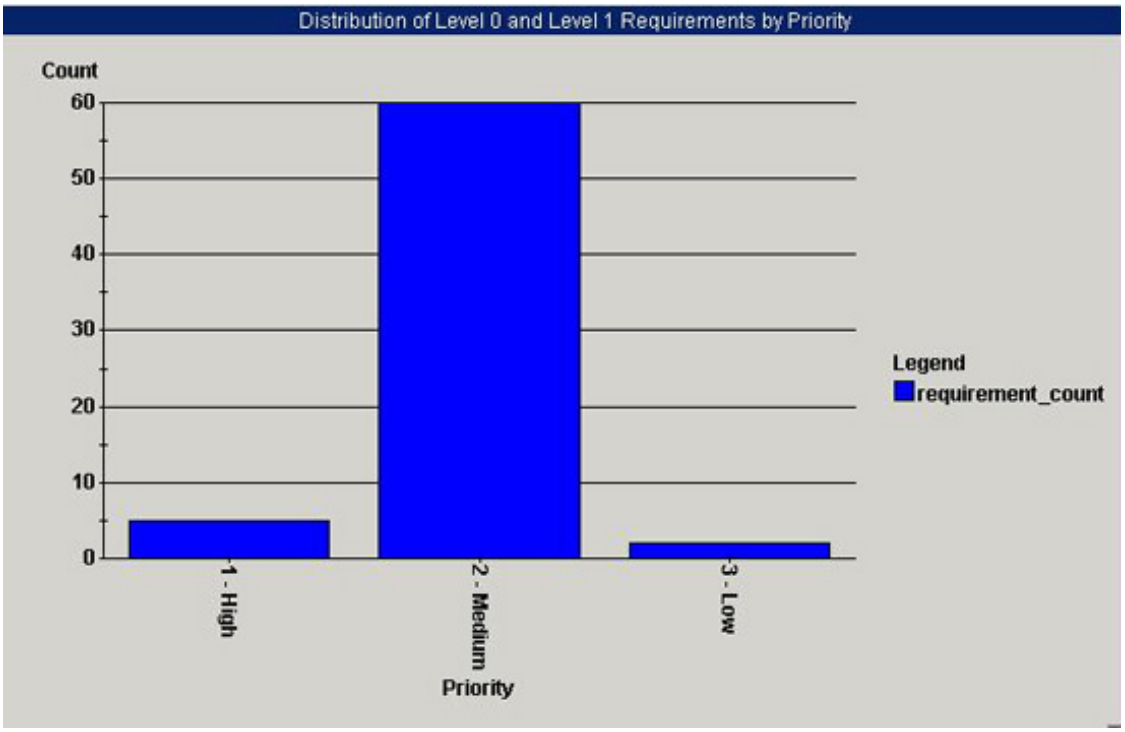


Figure 4: Total Number of Requirements, Based on Assigned Project Priority

At first glance, it's apparent that the distribution of requirements across the three priority groups is uneven. This may be an indication that no one has taken the time to make tough decisions about assigning priority. In this case, "medium" priority can be interpreted as "undecided."

Assessing Requirements by Iteration

Toward the end of the Inception phase, collective understanding of the requirements should be clearer, and information about planned iterations should be more complete. Figure 5 shows the proposed distribution of work for all requirements across planned iterations of development.

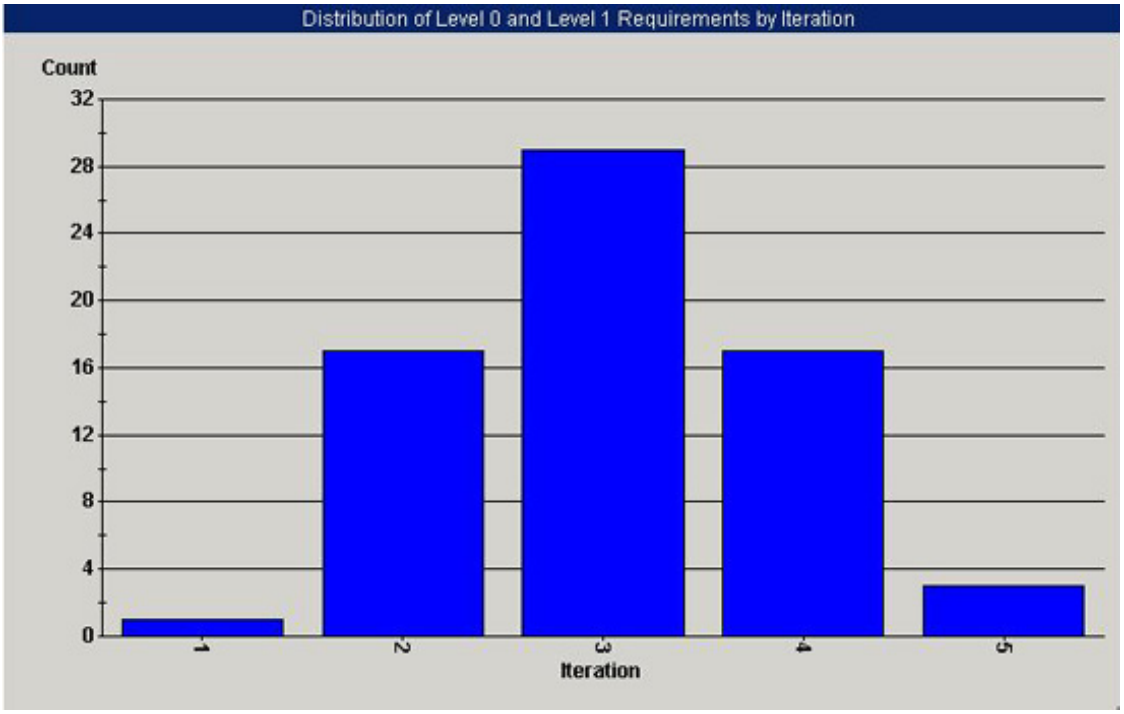


Figure 5: Distribution of Work for Requirements Across All Iterations

Figure 5 shows that the planned work is unevenly distributed across iterations. The relative sizes of the bars depend on the resources available for each iteration, the amount of work accomplished during previous iterations, and how difficult the requirements are to implement. In this case, we can see that there is a heavy load of requirements work toward the end of the Elaboration phase, which may not be realistic.

Additional Trends to Look for During Inception

As the Inception phase comes to a close, you can use ProjectConsole metrics charts to look for the following trends:

- Risks are being addressed and mitigated.
- The number of staff required to do the work does not exceed what you planned for.
- Expenditures are within budget (based on data from Microsoft Project).
- Tasks are being completed on schedule (based on data from Microsoft Project).

Monitoring the Elaboration Phase

During the Elaboration phase of a RUP project, a development team focuses on clarifying the architecture's scope, major functionality, and nonfunctional requirements such as performance requirements. Elaboration activities include establishing a sound architectural foundation, analyzing the problem domain, designing the solution, addressing the highest risk issues, and refining the software development plan.

To evaluate the success and completion of the Elaboration phase, stakeholders want to look at the stability of the product vision and architecture, the extent to which risks have been resolved, the sufficiency and credibility of the construction plan, and actual versus planned expenditures.

The charts in this section show data collected for the sample project over the first three weeks of an Elaboration phase scheduled to last one month.

Examining Use Cases That Affect System Architecture

One effective way to assess the progress of the Elaboration phase is to check the status of use cases that directly affect the project's architectural design. By the end of the Elaboration phase, all of these use cases should be verified. Figure 6 shows the total number of use cases, classified according to whether they do or do not affect the system architecture, three weeks into the Elaboration phase.

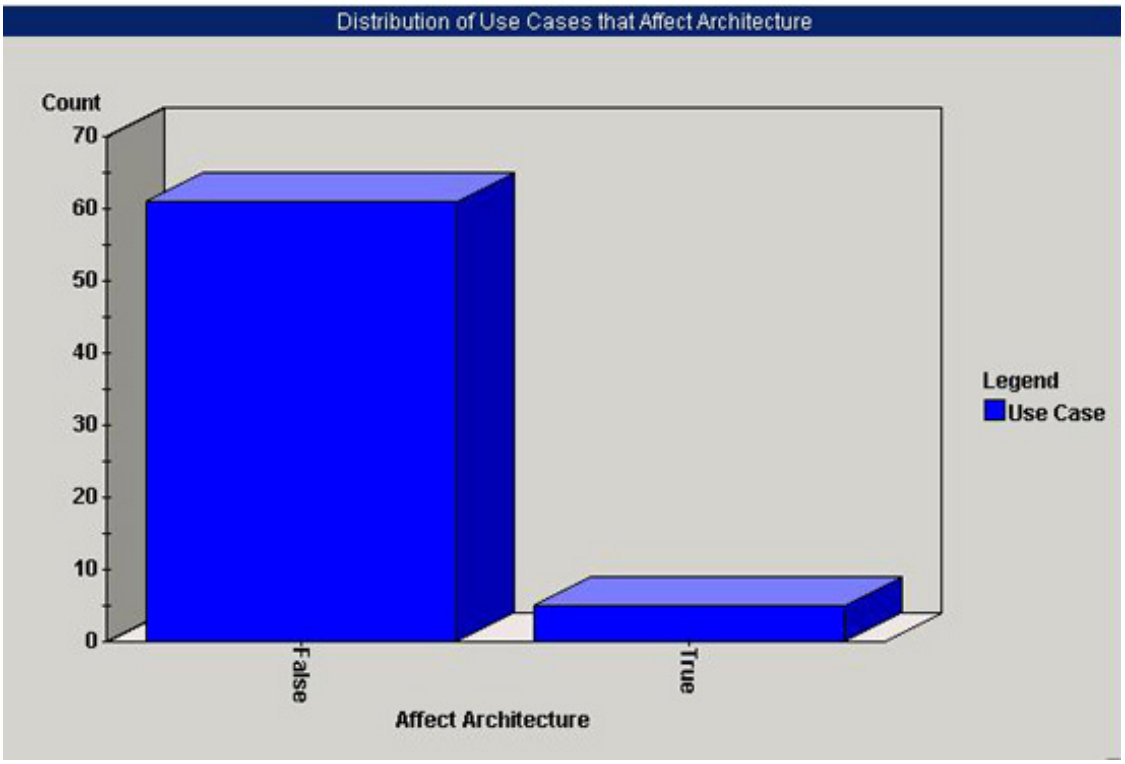


Figure 6: Number of Use Cases Three Weeks into Elaboration

Figure 6 shows that just five use cases require validation during Elaboration. If no use cases had been developed, it would not be possible to complete the Elaboration milestone and exit that phase.

Figure 7 shows the chart that's displayed if a ProjectConsole user "drills down" to see the details for the five use cases that affect the architecture. (To drill down for these details, a user can double-click the "True" column in the chart.)

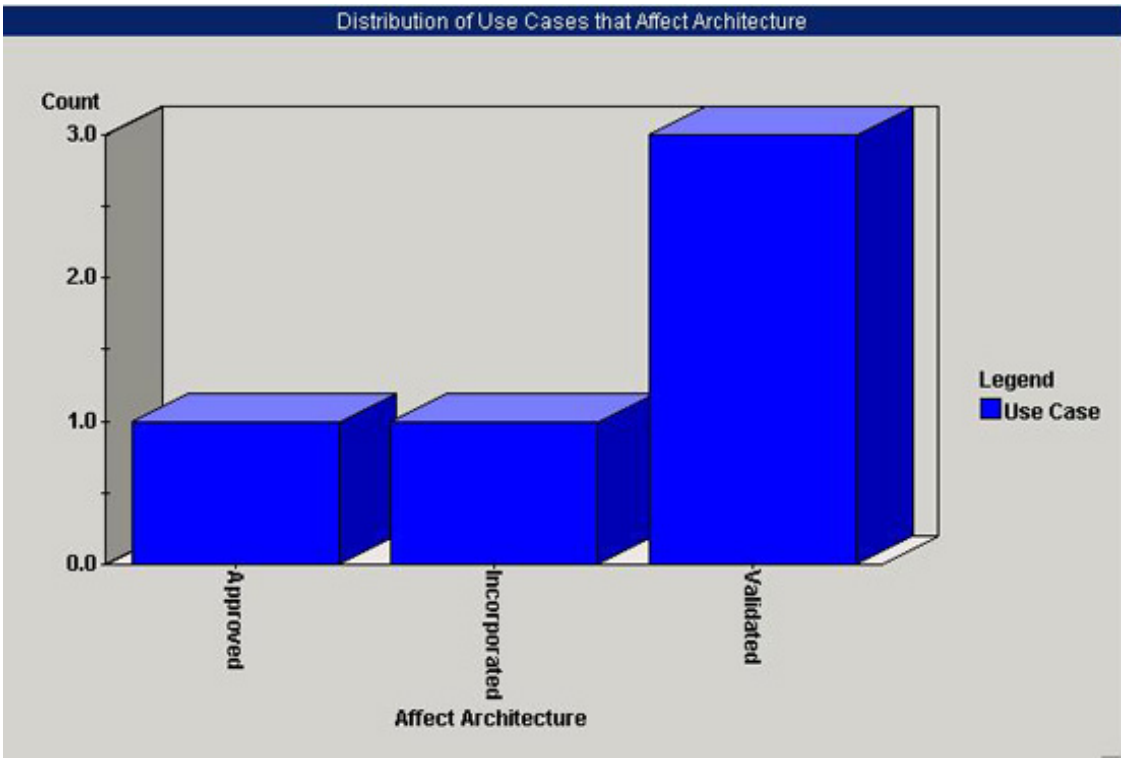


Figure 7: Detail for Elaboration Use Cases

Figure 7 shows that only two use cases remain to be validated before the end of Elaboration. One of these has already been incorporated, and the other has yet to be incorporated. We can see that, from an architectural design standpoint, some work is required before the Elaboration phase can come to a close.

Assessing the Level of Detail in Requirements

As Elaboration progresses, more requirements are defined in greater detail. As Elaboration nears a close, all high-level requirements should be fleshed out in preparation for the next phase: Construction. Figure 8 shows the distribution of requirements, based on level of detail, two weeks into the Elaboration phase. In this chart, the detail level increases with the number. A zero represents almost no detail, and a three represents the highest level of detail.

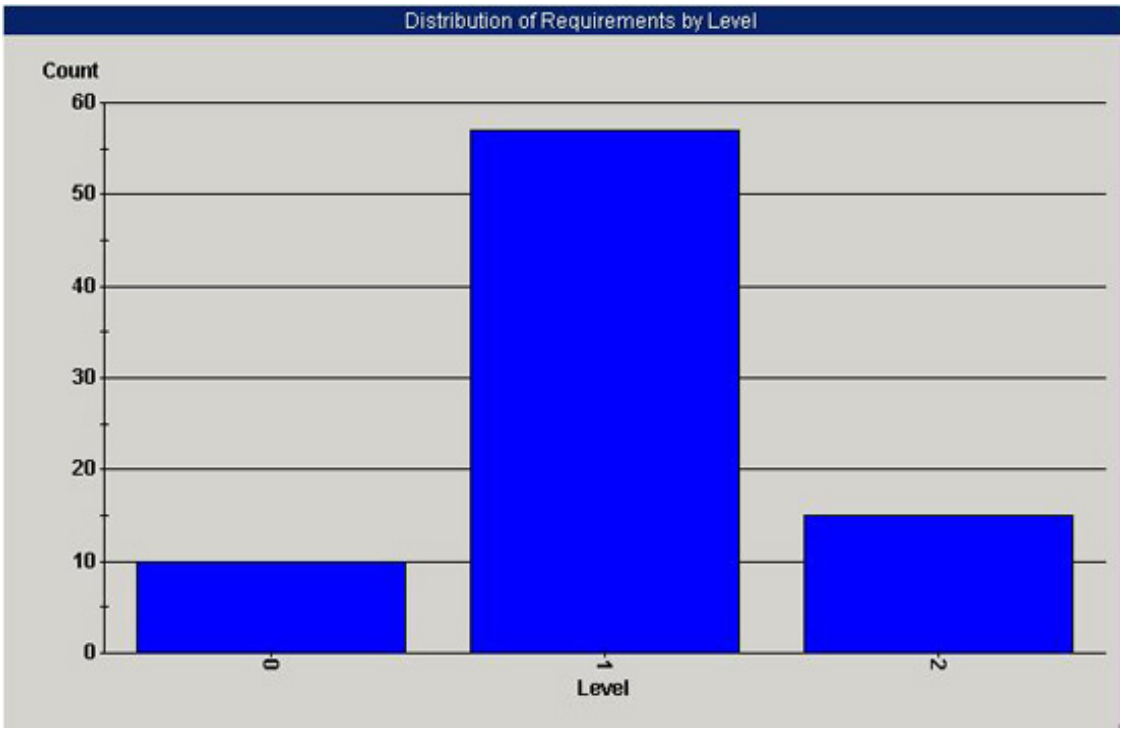


Figure 8: Distribution of Requirements, Based on Level of Detail Two Weeks into Elaboration

Halfway through the Elaboration phase, a higher proportion of requirements should be classified as "2"s, and few or none should be classified as "0"s. The results shown in Figure 8 suggest that more requirements need to be fleshed out, and that some prototyping might be needed.

Additional Trends to Look for During Elaboration

As the Elaboration phase nears its end, you can also use ProjectConsole metrics charts to check for the following trends:

- Approved requirements are beginning to get incorporated into the system under development.
- The number of approved requirements has significantly increased.
- Planning is almost complete, and the work planned for existing requirements is distributed fairly evenly across planned iterations.

Monitoring the Construction Phase

During the Construction phase of a RUP project, a project team focuses on developing and integrating all components and features into the product, and testing these features thoroughly. Successful completion of Construction is evaluated based on, among other things, the stability and maturity of product releases, and the readiness of stakeholders to transition the product to the user community.

The charts in this section show data collected for the sample project during a Construction phase scheduled to last three months.

Monitoring Open Defects

Figure 9 shows the number of open defects over a seven-week period.

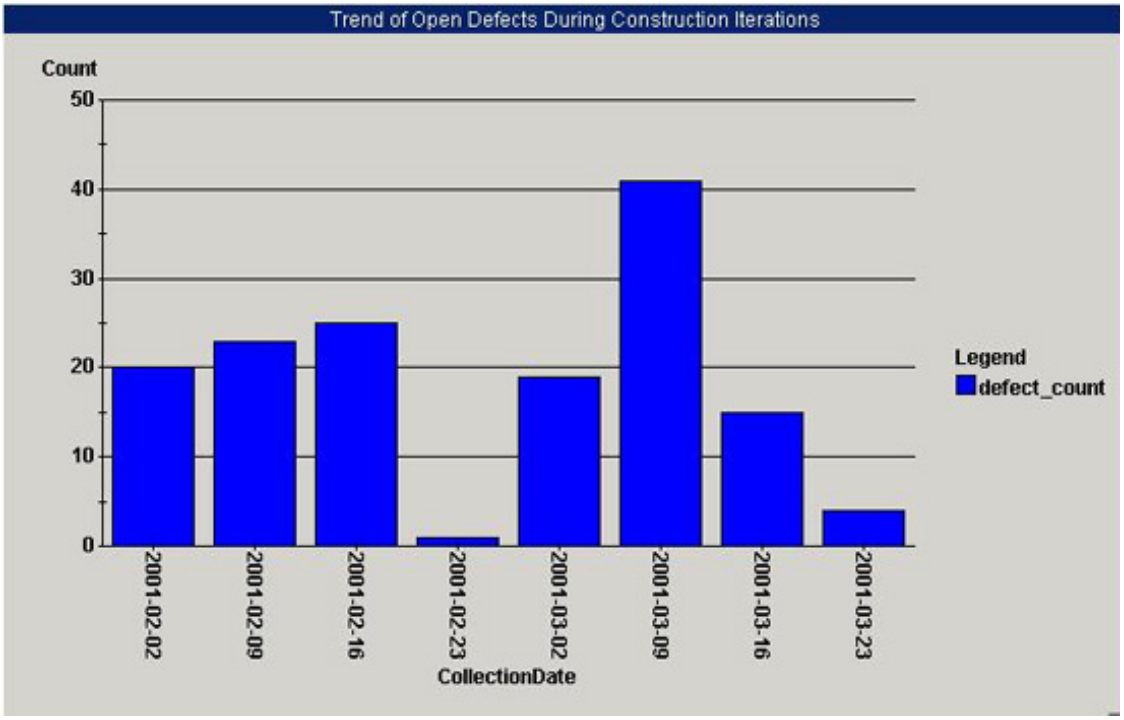


Figure 9: Open Defects Over a Seven-Week Period During Construction

One of the trends shown in Figure 9 approximates what you expect and want to see during the Construction phase: periods during which the number of open defects is high, followed by a dramatic decline in that number. This pattern suggests that the development team is stabilizing the product at the close of each Construction iteration.

Another noticeable trend is a continuous increase in the number of open defects, which probably means that the testing program in place is effective.

Additional Trends to Look for During Construction

Throughout the Construction phase, use ProjectConsole metrics charts to look for the following trends:

- Changes in the number of lines of code in the product under development. If the number of lines of code in a product count decreases while the number of classes stabilizes or increases, this is a healthy sign.
- Use-case validation. If use cases aren't getting validated until late in the Construction phase, it could mean that inadequate testing resources have been assigned, or that testers are reporting their results late.
- The number of staff required to do the work does not exceed what you planned for.

- Tasks are being completed on schedule (based on data from Microsoft Project).

Monitoring the Transition Phase

In the Transition phase of a RUP project, the development team focuses on moving the software product to the user community. The success of the Transition phase is evaluated based on user satisfaction with previous product versions and the balance between the actual resources expended versus planned expenditure.

The charts in this section show data collected for the sample project during a Transition phase scheduled to last one month.

Monitoring Open Defects for the Release

Figure 10 shows the number of open defects over the entire development lifecycle.

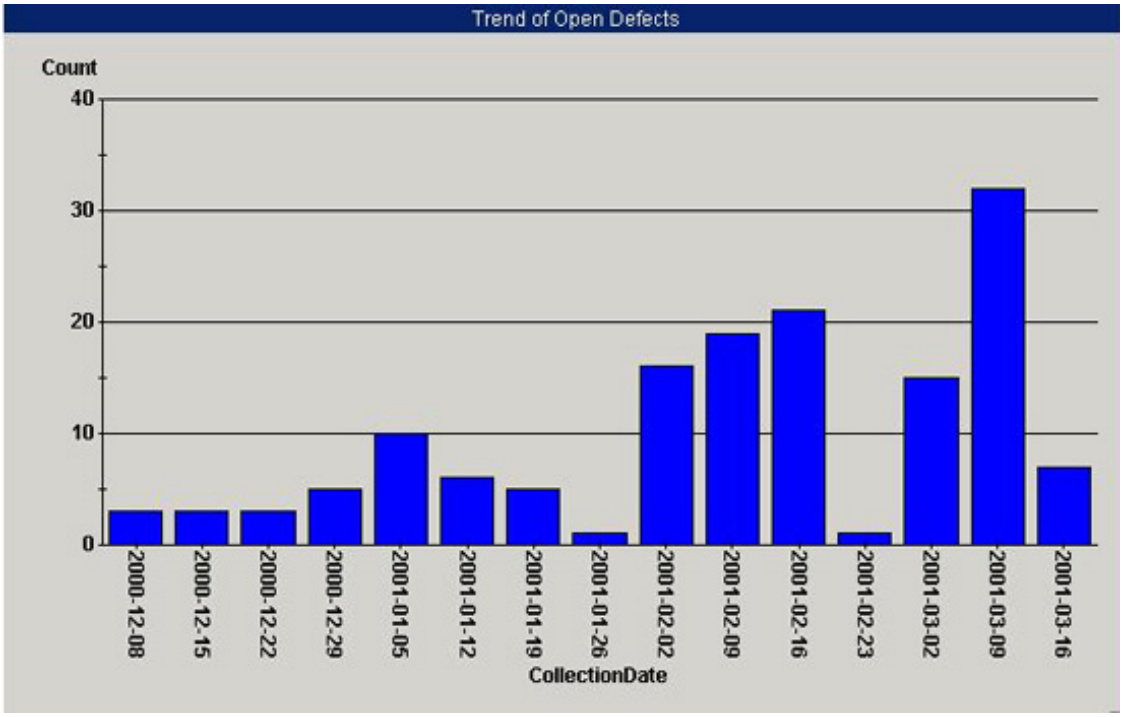


Figure 10: Number of Open Defects Over the Development Lifecycle

A given release can include multiple iterations. At the beginning of each iteration, the trend of open defects may increase, but as you near the end of each, and the release milestone, the number of open defects should decrease, indicating that the quality of the product is reaching an acceptable state.

Additional Trends to Look for During Transition

Throughout the transition phase, use ProjectConsole metrics charts to look for the following trends:

- The number of total defects across the entire development has leveled off, which suggests that the product has stabilized.
- The vast majority of change requests are closed.
- Code churn (lines added, modified, and deleted) has decreased to very low levels, supporting the assumption that the product is almost ready for release.
- Staffing requirements have not deviated from plan.
- Expenditures have been within the planned budget.
- The number of tasks completed late is very low.

Summary

In this article, we've introduced you to simple metrics that you can use to measure progress and assess quality during project development, using the Rational Unified Process and Rational Suite. The sample metric charts were produced using Rational ProjectConsole.

ProjectConsole allows a software development team to automatically quantify the current project status and assess development trends of their project with up-to-date metrics. ProjectConsole collects metrics data on a specified scheduled or on demand, from the Rational Suite's development environment and selected third-party tools. The results are presented visually in graphs, charts, and gauges.

With ProjectConsole charts and indicators, all project team members can analyze low-level details, planned-versus-actual metrics, historical data, and trend charts to get an overview of an entire project. This information enables a software development team to set realistic project expectations, more realistically assess potential risks, identify bottlenecks, realize the cause for late deliverables, take prompt corrective action, forecast future project milestones, and ultimately, put the entire team in a better position to objectively and accurately measure project progress and quality.

Notes

1 For an interesting perspective on predicting and monitoring project trends, see Joe Marasco's article, "A Physicist Looks at Project Progress" in the November issue of *The Rational Edge*.

2 The examples are based on data collected from Rational RequisitePro®, Rational ClearQuest®, and Rational ClearCase®. Rational Suite includes a tutorial that describes how to install and use ProjectConsole to deploy these metrics into your production environment.

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

Thank you!

The Spirit of the RUP

by **Per Kroll**
Director, Rational Unified Process Development
and Product Management Teams

At the core of the Rational Unified Process®, or RUP®, lie eight fundamental principles that represent the essential "Spirit of the RUP." These are the experiences gleaned from a huge number of successful projects distilled into a few simple guidelines. Although these principles represent neither a complete view of the RUP nor the full complexity of those many projects, understanding these principles will guide you in better applying the RUP to your own projects. These principles are:

1. *Attack major risks early and continuously... or they will attack you.*
2. *Ensure that you deliver value to your customer.*
3. *Stay focused on executable software.*
4. *Accommodate change early in the project.*
5. *Baseline an executable architecture early on.*
6. *Build your system with components.*
7. *Work together as one team.*
8. *Make quality a way of life, not an afterthought.*

You may find that one or several of these principles are incompatible with what you would like to apply to your project, and that's fine. Even at its essence, the RUP should be considered a smorgasbord from which you choose the dishes that fit your needs.

Introducing Iterative Development

The RUP proposes an iterative approach to software development, which means that a project is divided into several small projects that run sequentially, one directly after another. Each iteration has a well-defined set of objectives, and concludes by delivering an executable that is a step closer to the final product than the last one. Each iteration contains elements of requirements management; analysis and design; implementation; integration; and testing.

The RUP[1] provides a structured approach to iterative development that divides a *project* into four *phases*: Inception, Elaboration, Construction, and Transition. The objectives of the phases are:

- Inception: Understand what to build.
- Elaboration: Understand how to build it.
- Construction: Build a beta version of the product.
- Transition: Build the final version of the product.

Each *phase* contains one or more *iterations*, which focus on producing the technical deliverables necessary to achieve the business objectives of that phase. Each phase includes as many iterations as it takes to address the objectives of that phase, but no more.

This evolutionary approach to software development can be seen as an umbrella for all the principles of software development described in this article. Some of the principles can be applied outside the context of iterative development, and iterative development can be done without applying all of the principles. There is, however, a strong correlation between successful iterative development and the principles described in this article, and to optimize your implementation of an iterative approach, you should attempt to apply as many of the principles as is feasible for your project.

1. Attack Major Risks Early and Continuously...Or They Will Attack You

As Tom Gilb said, "If you don't actively attack the risks, they will actively attack you."[2] As Figure 1 shows, one of the prime benefits of the iterative approach is that it allows you to identify and address major risks *early* in the project.

Why address top risks early on? Unaddressed risks mean that you are potentially investing in a faulty architecture and/or a nonoptimal set of requirements. This is bad software economics. In addition, the amount of risk is directly correlated to the difference between the upper and lower estimates of how long it will take to complete a project. To come up with accurate estimations, you need to identify and address risks up front.

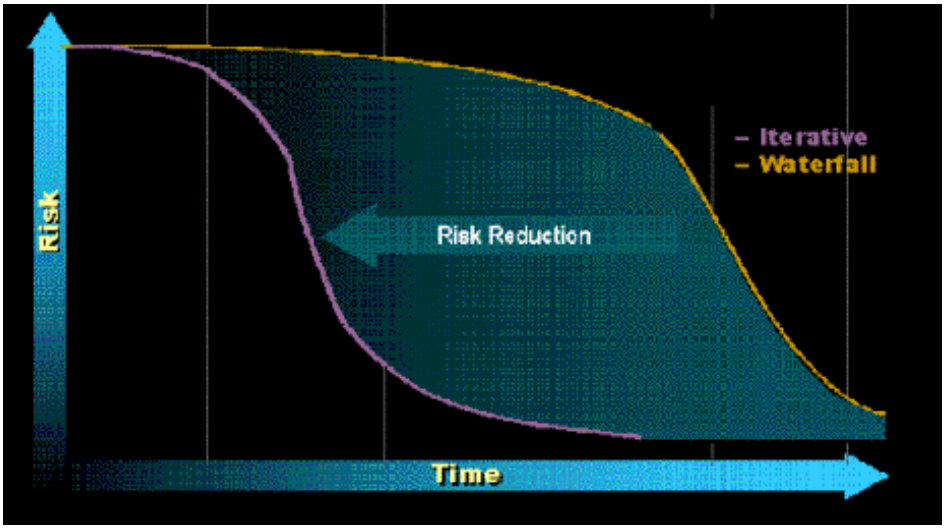


Figure 1: Risk Reduction Profiles for Waterfall and Iterative Developments. *A major goal with iterative development is to reduce risk early on. This is done by analyzing, prioritizing, and attacking top risks in each iteration.*

How do you deal with risks early on? At the beginning of each iteration, the RUP advises you to make, or revise, a list of top risks. Prioritize the risk list, then decide what you need to do to address, typically, the top three to five risks. For example, the risks may look as follows:

- **Risk 1:** We are concerned, based on past experience, that Department X will not understand what requirements we plan to provide, and as a result will request changes *after* beta software is delivered.
- **Risk 2:** We do not understand how we can integrate with legacy system Y.
- **Risk 3:** We have no experience in developing on the Microsoft .NET platform or in using Rational Rose.
- **Risk 4:** ...etc.

Now, how will this risk list be used? Addressing risks should be a priority for everyone throughout the project. Look at what you would "normally" do for the iteration at hand, then slightly modify the plan to make sure that you deal with your risks. Typically, risks need to be addressed from multiple perspectives, such as requirements, design, and test. For each of these perspectives, start with a coarse solution, and successively detail it to diminish the risk. You may, for example, add the following actions to address the above-identified risks:

- **Risk 1:** As the use cases related to Department X are developed, complement them with a UI prototype. Set up a meeting with Department X, and walk them through each use case, using the UI prototype as a storyboard. Get a formal sign-off on the requirements. Throughout the project, keep Department X in the loop on progress, and provide them with early prototypes and/or alpha releases.
- **Risk 2:** Have a "tiger team" with one or two skilled developers build

an actual prototype that shows how to integrate with legacy system Y. The integration may be a throwaway, but the prototype should prove that you actually can integrate with the legacy system. Throughout the project, ensure appropriate testing of the integration with legacy system Y.

- An alternative would be to cut the scope of the project, so you do not have to integrate with legacy system Y. This strategy is called "risk avoidance" and is typically highly effective. Note that all other examples in this list are of the strategy type "risk mitigation."
- **Risk 3:** Send a couple of people for training on Microsoft .NET and Rational Rose, respectively, and find the budget to bring in a Rational Rose mentor two days per week for the first three weeks of the Elaboration phase. Recruit a team member with an understanding of the .NET platform.
- **Risk 4:**...etc.

Many project risks are associated with the architecture. This is why the RUP's primary objective in the Elaboration phase is getting the architecture right. To do this you not only design the architecture, but you also implement and test it. (Find out more about this in "Baseline an Executable Architecture Early On" below.)

One thing to keep in mind is that the risk list continuously changes. Attacking risk is a constant battle -- in each iteration you will be able to reduce or remove some risks, while others grow and new ones appear.

Summary

The RUP provides a structured approach to addressing top risks early on, which decreases overall costs, and allows you to earlier make realistic and accurate estimations of how much time it will take to complete the project. Remember that risk management is a dynamic and ongoing process.

2. Ensure that You Deliver Value to Your Customer

Delivering value to your customer is a pretty obvious goal, but how is it done? Our recommendation is closely related to iterative development and the "Use-Case-Driven Approach."[3] So, what are use cases? Use cases are a way of capturing functional requirements. Since they describe how a user will interact with the system, they are easy for a user to relate to. And since they describe the interaction in a time-sequential order, it is easy for both users and analysts to identify any holes in the use case. A use case can almost be considered a section in the future user manual for the system under development, but written with no knowledge about the specific user interface. You do not want to document the user interface in the use case; instead, you complement the use-case descriptions with a UI prototype, for example, in the form of screen shots.

Many people have learned to love use cases for their simplicity and their

ability to facilitate rapid agreement with stakeholders on what the system should do. But after ten years of use-case consulting, I feel that this is not their primary benefit. Rather, I think their major advantage is that they allow each team member to work very closely to the requirements when designing, implementing, testing, and finally writing user manuals. Use cases force you to stay externally focused on the user's perspective, and they allow you to validate the design and implementation with respect to the user requirements. They even allow you to carefully consider user needs when planning the project and managing scope (see Figure 2).

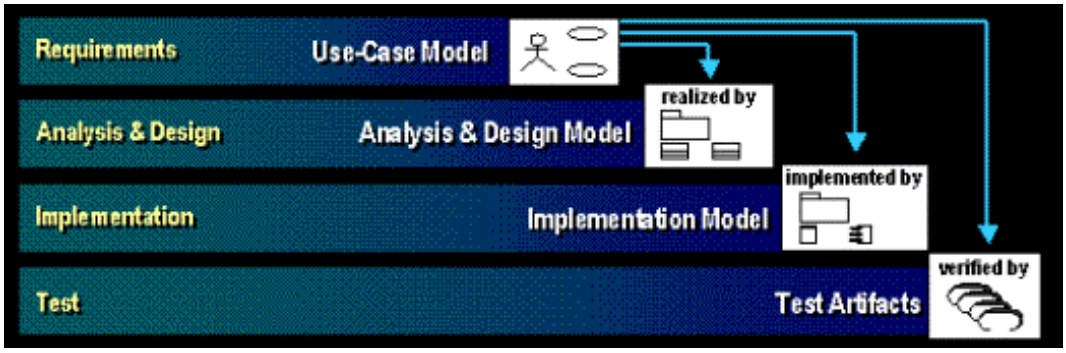


Figure 2: How Use Cases Relate to Other Software Engineering Models. *Use cases are a highly effective vehicle for capturing requirements. They also allow you to work closely to the requirements when doing design, implementation, and testing, ensuring that you deliver the requirements.*

Since a use case describes how a user will interact with the system, you can use UML[4] diagrams such as Sequence Diagrams or Collaboration Diagrams to show how this interaction will be implemented by your design elements. You can also identify test cases from a use case. This ensures that the services the users expect from the system really are provided. Since the use cases are "*user manuals without knowledge of the UI*," they make a solid starting point for writing user manuals. [**Editor's note**: for more on the relationship between use cases and the writing of user manuals, see Robert Pierce's article, "Keep Documentation Up to Date Throughout Development with Rational Rose" in this issue of *The Rational Edge*.] And when prioritizing which capabilities to deliver when managing scope, you choose which use cases to implement. And as we noted above, use cases allow you to work closely to the requirements throughout the development lifecycle. An expression that captures the essence of use cases is "*Documenting customer needs is great; implementing them is better*."

Summary

Use cases make it easy to document user requirements and to help all stakeholders understand the capabilities that are to be delivered. More essentially, use cases allow you to work closely to the requirements when doing design, implementation, and testing, thereby ensuring that you not only document, but also deliver, the user requirements.

3. Stay Focused on Executable Software

This third essential RUP principle has several facets. First, you should, to

the extent possible, measure progress by measuring your *executable* software. It is great to know that ten out of twenty use cases have been described, but that does not necessarily mean that 50 percent of the requirements are completed. What if you later find that ten of those use cases require major rewrites because you did not properly understand the user requirements? That could mean that you were only 25 percent complete, right? So what you can really say when you have completed half of the use cases is that you are probably not more than 50 percent done with the requirements.

The best way of measuring progress is by measuring what software is up and running. This allows you to do testing and, based on testing and defect rates, assess the true progress that has been made. When the typical developer states, "I am 90 percent done," you can ask, "Great, but can you please demo what is up and running?" and then get a solid idea of what has actually been accomplished. As an architect/team leader/manager, you should always strive to have working software demonstrated and to look at test coverage and test results, rather than be fooled by the often-false reality of completed documents. This does not mean that you should disregard the information in completed documents, but when considered in isolation they provide a poor measure of true progress.

Second, a clear focus on executable software also promotes right thinking among your team; you run less risk of overanalyzing and theorizing, and instead get down to work to prove whether solution A or B is the better. Forcing closure by producing executable software is often the fastest way of mitigating risk.

A third attribute of this focus on executable software is that *artifacts other than the actual software are viewed as supporting artifacts*. They are there to allow you to produce better software. By staying focused on executable software, you are better prepared to assess whether producing other artifacts -- such as requirement management plans, configuration management plans, use cases, test plans, and so on -- will really lead to software that works better and/or is easier to maintain. In many cases the answer is yes, but not always. You need to weigh the cost of producing and maintaining an artifact against the benefit of producing it. The benefit of producing many artifacts typically increases as your project grows larger, as you have more complicated stakeholder relations, as your team becomes distributed, as the cost of quality issues increases, and as the software is more critical to the business. All these factors drive toward producing more artifacts, and treating them more formally. But for every project, you should strive to minimize the number of artifacts produced, to reduce overhead.

A good guideline is that if you are in doubt as to whether or not to produce an artifact, don't produce it. But do not use this guideline as an excuse to skip essential activities such as setting a vision, documenting requirements, having a design, and planning the test effort. Each of these activities produces artifacts of obvious value. If the cost of producing an artifact is going to be higher than the return on investment, however, then you should skip it.

One of the most common mistakes RUP users make is to produce artifacts just because the RUP describes how to produce them. Remember, the RUP is a smorgasbord, and it is typically unwise to eat every dish at a table like the one in Figure 3.

Figure 3: Consider the RUP as a Smorgasbord. *You can think of the RUP as a smorgasbord of best practices. Rather than eat everything, eat only your favorite dishes -- the ones that make sense for your specific project.*

Summary

Working software is the best indicator of true progress. When assessing progress, as much as possible look at what code is up and running, and which test cases have been properly executed. A strong focus on working software also enables you to minimize overhead by producing only those artifacts that add more value to your project than they cost to produce.

4. Accommodate Change Early in the Project

Change is good. Actually, change is great. Why? Because most modern systems are too complex to allow you to get the requirements and the design right the first time. Change allows you to improve upon a solution. If there is no change, then you will deliver a defective solution -- possibly so defective that the application has no business value. That is why you should welcome and encourage change. And the iterative approach has been optimized to do exactly that.

But change can also have severe consequences. Constant change will prevent project completion. Certain types of change late in the project typically mean a lot of rework, increased cost, reduced quality, and probable delays -- all things you want to avoid. To optimize your change management strategy, you need to understand the relative cost of introducing different types of changes at different stages of the project lifecycle (see Figure 4). For simplicity, I group changes into four categories.

- **Cost of change to the business solution.** Change to the business solution involves major rework of requirements to address a different set of users or user needs. There is a fair amount of flexibility in making this type of modification during the Inception phase, but costs escalate as you move into the Elaboration phase. This is why you force an agreement on the vision for the system in Inception.
- **Cost of change to the architecture.** When following the RUP, you can make fairly significant architectural changes until the end of Elaboration at low cost. After that, significant architectural changes become increasingly costly, which is why the architecture must be baselined at the end of Elaboration.
- **Cost of change to the design and implementation.** Due to the component-based approach, these types of changes are typically localized, and can hence be made at fairly low cost through the Construction phase. These types of changes are, however, increasingly expensive in the Transition phase, which is why you typically introduce feature freeze at the end of Construction.
- **Cost of change to the scope.** The cost of cutting scope -- and hence postponing features to the next release -- is relatively inexpensive throughout the project, if done within limits. Scope cutting is a key tool project managers should use to ensure on-time project delivery.

The above cost considerations mean that you need to manage change. You need to:

- Have procedures in place for approving whether to introduce a change.
- Be able to assess the impact of change.
- Minimize the cost of change.

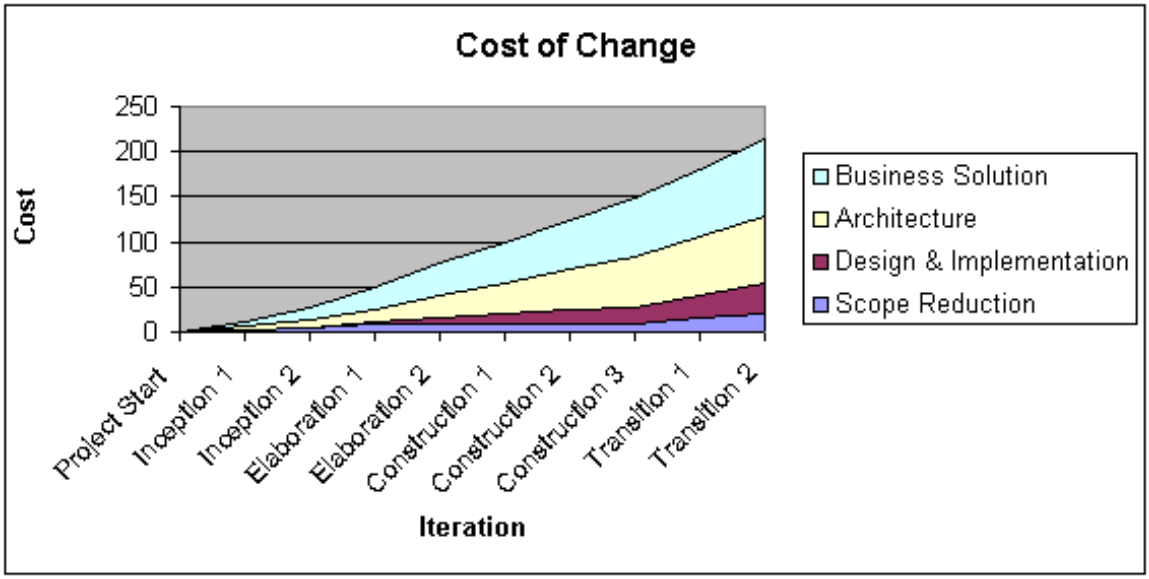


Figure 4: The Cost of Introducing Change Varies Through the Lifecycle. *The cost of*

introducing a change varies according to the lifecycle phase, and each type of change has its own cost profile. The RUP milestones at the end of each phase are optimized to minimize overall cost of change, while maximizing freedom to make changes. In general, as the project progresses, you should be more careful about introducing change, especially to the overall business solution and the architecture.

Change approvals are done through a combination of manual procedures (for example, through Change Control Boards), and through tools such as Configuration and Change Management software. Assessing the impact of change is primarily done through traceability between tools for requirements, design, code, and test. When changing a class or a requirement, you should be able to understand what other things are potentially affected; this is where traceability saves a lot of grief.

The cost of change is minimized by having procedures in place for managing changes and also by assessing their impact. Again, the right tool support can have a tremendous effect. Round-trip engineering between requirements and design, and between design and code, are examples of automation reducing the cost of change.

Summary

The cost of change increases the further you are into a project, and different types of changes have different cost profiles. The RUP's phases have been optimized to minimize overall cost of change, while maximizing the ability to allow for change. This is why the RUP forces agreement on the overall vision at the end of the Inception phase, a baselined architecture at the end of the Elaboration phase, and feature freeze at the end of the Construction phase. Using the right tools can play a powerful role in managing change and minimizing its cost.

5. Baseline an Executable Architecture Early On

A lot of project risks are associated with the architecture. That is why you want to get the architecture right. In fact, the ability to baseline a functioning architecture -- that is, to design, implement, and test the architecture -- early in the project is considered so essential to a successful project that the RUP treats this as the primary objective of the Elaboration phase, which is phase two of this four-phase process.

First, what do we mean by architecture?[5] The architecture comprises the software system's most important building blocks and their interfaces -- that is, the subsystem, the interfaces of the subsystems, and the most important components and their interfaces (see Figure 5). The architecture provides a skeleton structure of the system, comprising perhaps 10 to 20 percent of the final amount of code. The architecture also consists of so-called "architectural mechanisms." These are common solutions to common problems, such as how to deal with persistency or garbage collection. Getting the architecture right is difficult, which is why you typically use your most experienced people for this task.

Having the skeleton structure in place provides a sound understanding of

the building blocks or components needed for the final product. And, having followed the RUP's iterative process, your team will already have gained some valuable experience in doing analysis, design, implementation, and testing, so you will usually have a firm grasp of what it will take to complete the system. Baselining an executable architecture also lays the groundwork for accurate assessments of how many resources are needed, and how long it will take to complete the project. Early understanding of this enables you to optimize your resource profile and manage scope to best address your business needs.

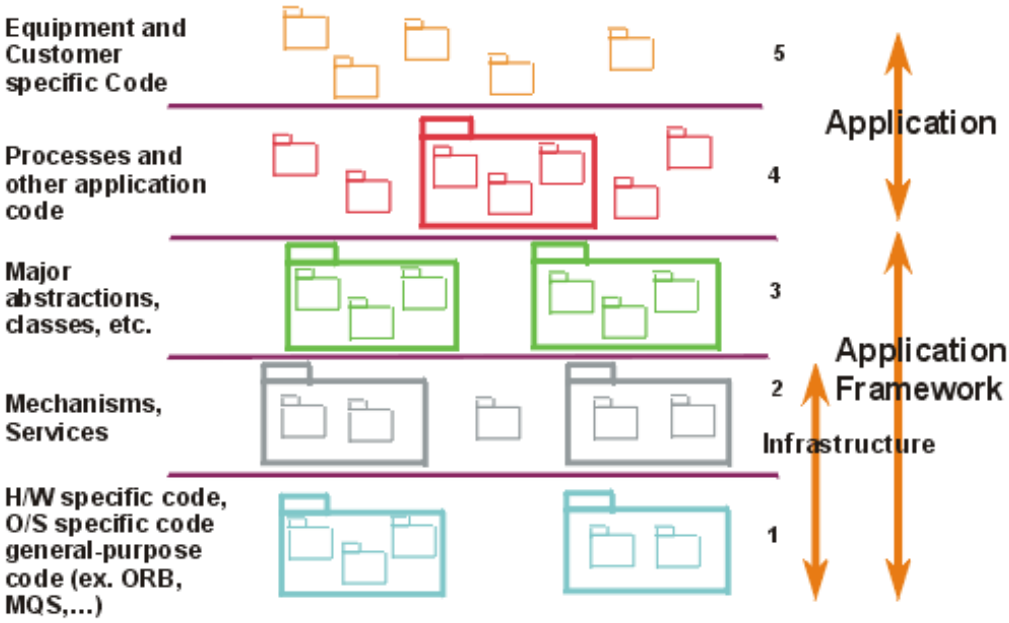


Figure 5: A System Architecture. *The architecture provides an understanding of the overall system and a skeleton structure for the application. It includes, among other things, the subsystems and their interfaces, and the most common components and their interfaces.*

When the architecture is in place, you have addressed many of the most difficult parts of building the system. It is now much easier to introduce new members to the project; boundaries are provided for additional code through the definition of key components and baselining of interfaces, and the architectural mechanisms are ready to be used, providing ready-made solutions to common problems.

Summary

The architecture is the system's skeleton structure. By designing, implementing, and testing the architecture early in the project, you address major risks and make it easier to scale up the team and to introduce less-experienced team members. Finally, since the architecture defines the system's building blocks or components, it enables you to accurately assess the effort needed to complete the project.

6. Build Your System with Components

One aspect of functional decomposition[6] is that it separates data from functions. One of the drawbacks with this separation is that it becomes expensive to maintain or modify a system. For example, a change to how

data is stored may impact any number of functions, and it is generally hard to know which functions throughout a given system may be affected (see Figure 6). This is the major reason the Y2K issue was so difficult to address.

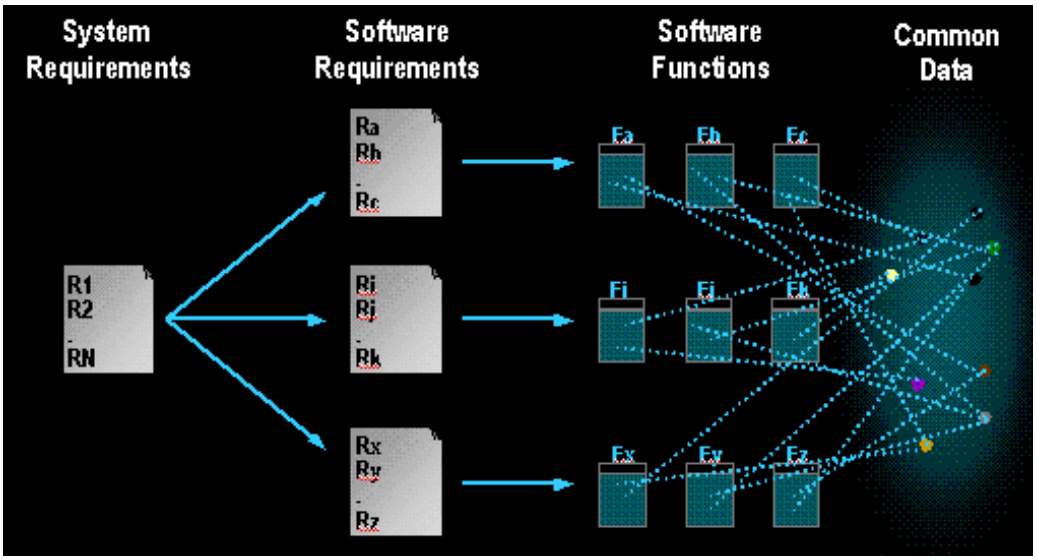


Figure 6: A Functional Decomposition Architecture. *Functional decomposition has the disadvantage that changes (to how data is stored, for example) may impact many functions, leading to systems that are both highly difficult and expensive to maintain.*

On the other hand, component-based development encapsulates data and the functionality operating upon that data into a component. When you need to change what data is stored, or how the data can be manipulated, those changes can be isolated to one component. This makes the system much more resilient to change (see Figure 7).

To communicate with a component, and hence take advantage of all its capabilities and code, you only need to know the component's *interface*. You don't need to worry about its internal workings. Even better, a component can be completely rewritten without impacting the system or system code, as long as its interface does not change. This is an important feature of component-based development called encapsulation, which makes components easy to reuse.

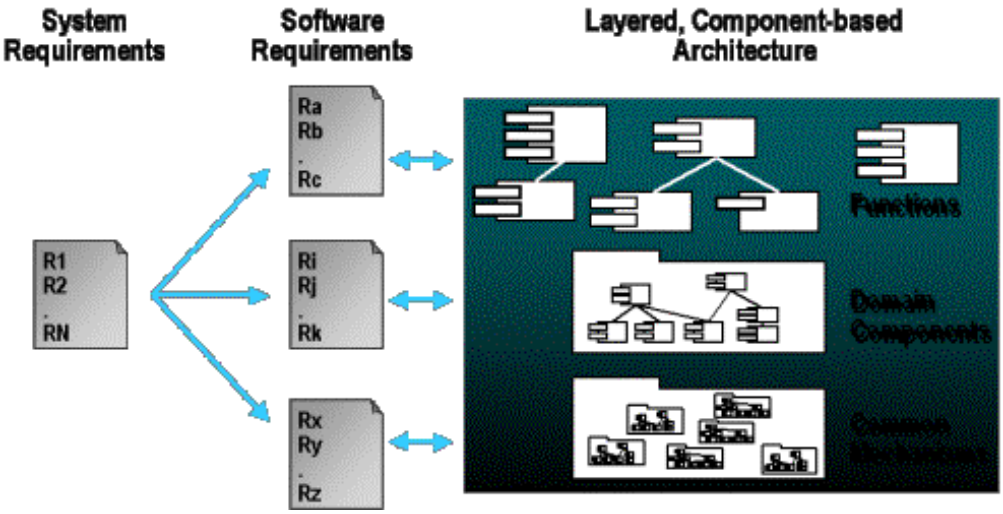


Figure 7: A Component-based Architecture. *Component-based development leads to systems that are more resilient to changes in requirements, technology, and data.*

A component can also be assembled by other components, thereby allowing it to provide many advanced capabilities. The combination of encapsulation and the availability of large components radically increases the productivity associated with reuse when developing applications.

Component technology is also the basis for the Web services initiatives recently launched by all the major platform vendors on both J2EE and .NET platforms, and it is why Web services may well be "the next big thing" in software development. In short, Web services are "Internet-enabled" components. Think of them as components on steroids. Like all components, they have a well-defined interface, and you can take advantage of all their capabilities simply by knowing that interface. As long as the interface does not change, you are unaffected by changes to a Web service. The major difference is that while a normal component typically limits you to communicating with components developed on the same platform, and sometimes only with components compiled on the same system, a Web services architecture allows you to communicate independently of the platform by exposing component interfaces over the Internet.[7]

Summary

Component-based development relies on the object-oriented principle of *encapsulation*, and enables you to build applications that are more resilient to change. Components also enable a higher degree of reuse, allowing you to build higher quality applications faster. This can radically decrease system maintenance costs. Component-based technology is the basis for Web services offered on J2EE and .NET platforms.

7. Work Together as One Team

People are the project's most important asset. Software development has become a team sport, and an iterative approach to software development impacts the ways in which you organize your team, the tools your team needs, and the values of each team member.

Traditionally, many companies have had a functional organization: All the analysts are in one group, the designers are in another group, and testers are in yet another group -- maybe even in another building. Although this organizational structure builds competency centers, the drawback is that effective communication among the three groups becomes compromised. Requirements specifications produced by analysts are not used as input by developers or testers, for example. This leads to miscommunication, extra work, and missed deadlines.

Functional organizations may be acceptable for long-term waterfall projects, perhaps as long as eighteen months or more. But as you move toward iterative development and shorter projects of nine months, or even

two to three months, you need a much higher bandwidth of communication between teams. To achieve this, you must:

- Organize your projects around cross-functional teams containing analysts, developers, and testers.
- Provide teams with the tools required for effective collaboration across functions.
- Ensure that team members have the attitude of "I'll do what it takes to get high-quality working software," rather than "I'll do my small part of the lifecycle."

Let's take a closer look at each of these points.

The project team should consist of analysts, developers, testers, a project manager, architects, and so on. You might say that this works for small projects, but what happens when projects become bigger with, say, fifty people involved? The answer is to organize around the architecture,[8] to group what we call "teams of teams." (See Figure 8.) Have a team of architects that own the architecture; they decide on the subsystems and the interfaces between them. Then, for each subsystem, have a cross-functional team consisting of analysts, developers, and testers who work closely to ensure high-bandwidth communication and fast decisions. They communicate with other teams primarily through the architecture and the architecture team.

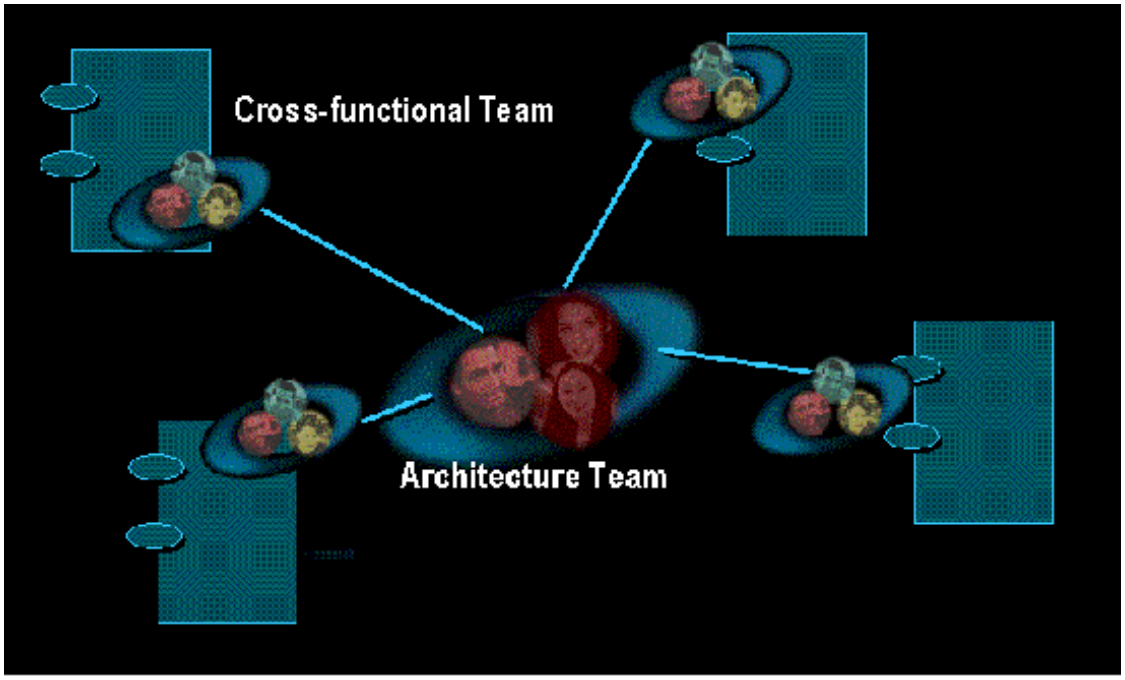


Figure 8: Teams Organized Around Architecture. *If the project is too big to have everyone on one team, organize teams around the architecture in "teams of teams." An architecture team owns the subsystems and their interfaces, and a cross-functional team is responsible for each of the subsystems.*

For a team of analysts, developers, and testers to work closely together, they need the right infrastructure. You need to ensure that all team members have access to the requirements, defects, test status, and so on.

This, in turn, puts requirements on tooling. Communication between the different team members must be facilitated through integration between their tools. Among other things, this increases the ROI on tools, allowing round-trip engineering and synchronization of requirements, design elements, and test artifacts.

Working together as a team also enforces joint ownership of the final product. It eliminates finger pointing and assertions such as "*Your* requirements were incomplete" or "*My* code has no bugs."[9] Everyone shares project responsibility and should work together to solve issues as they arise.

Summary

An iterative approach increases the need for working closely as a team. Avoid functional organizations and instead use cross-functional teams of generalists, analysts, developers, and testers. Ensure that the tool infrastructure provides each team member with the right information, and promotes synchronization and round-trip engineering of artifacts across disciplines. Finally, make sure that team members take joint ownership of the project results.

8. Make Quality a Way Of Life, Not an Afterthought

One of the major benefits of iterative development is that it allows you to initiate testing much earlier than is possible in waterfall development. Already in the second phase, Elaboration, executable software is up and running, implementing the architecture (see Figure 9). This means you can start testing to verify that the architecture really works. You can, for example, do some simple load and performance testing of the architecture. Gaining early feedback on this (perhaps one third of the way into the project) may result in significant time and cost savings down the road.

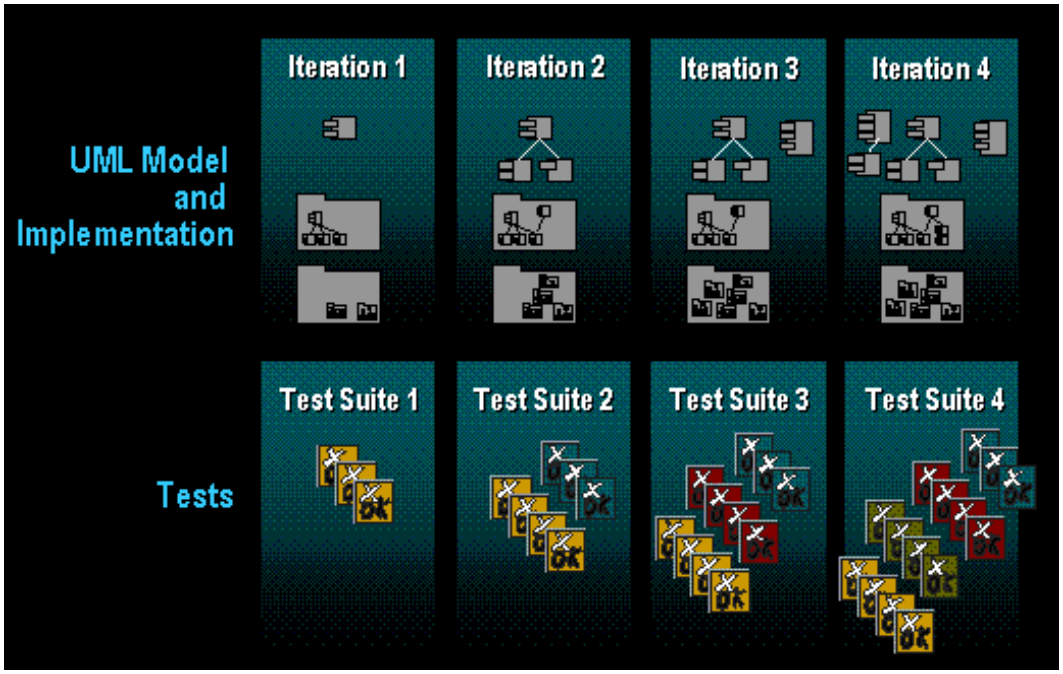


Figure 9: Testing Is Initiated Early and Expanded Upon in Each Iteration. *The RUP*

promotes early testing. Software is built in every iteration and tested as it is built. Regression testing ensures that new defects are not introduced as new iterations add functionality.

In general, the RUP requires you to test capabilities as you implement them. Since the most important capabilities are implemented early in the project, by the time you get to the end, the most essential software may have been up and running for months, and is likely to have been tested for months. It is not a surprise that most projects adopting the RUP claim that an increase in quality is the first tangible result of the improved process.

Another enabler is what we call "Quality By Design." This means coupling testing more closely with design. Considering how the system should be tested when you design it can lead to greatly improved test automation, because test-code can be generated directly from the design models. This saves time, provides incentives for early testing, and increases the quality of testing by minimizing the number of bugs in the test software. (After all, test scripts are software and typically contain a lot of defects, just like any other software.)

Iterative development not only allows earlier testing; it also forces you to test more often. On one hand, this is good because you keep testing existing software (so-called regression testing) to ensure that new errors are not introduced. On the other hand, the drawback is that regression testing may become expensive. To minimize costs, try to automate as much testing as possible. This often radically reduces costs.

Finally, quality is something that concerns every team member, and it needs to be built into all parts of the process. You need to review artifacts as they are produced, think of how to test requirements when you identify them, design for testability, and so on.

Summary

Ensuring high quality requires more than just the participation of the testing team. It involves *all* team members and *all* parts of the lifecycle. By using an iterative approach you can do earlier testing, and the quality-by-design concept allows software designers to automate test code generation for earlier and higher quality testing, thus reducing the number of defects in the test code.

Conclusion

We have examined the fundamental principles that represent the "Spirit of the RUP." Understanding these principles will make it easier for you to properly adopt the RUP. Don't get lost in the wide set of available activities and artifacts that the RUP offers; instead, let the "spirit" guide you, and you will more quickly find which activities and artifacts are appropriate for your specific project.

Notes

[1] Philippe Kruchten, *The Rational Unified Process: An Introduction*, 2nd ed. Addison-Wesley, 2000.

[2] Tom Gilb, *Principles of Software Engineering Management*. Addison-Wesley, 1988.

[3] For more information, see Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard, *Object-Oriented Software Engineering: A Use-Case Driven Approach*. Addison-Wesley, 1992.

[4] The UML, or Unified Modeling Language, is a standard managed by the Object Management Group. Rational Software developed the initial proposal for this standard.

[5] For a good discussion on this topic, see Philippe Kruchten, *The Rational Unified Process: An Introduction*, 2nd ed. Addison-Wesley, 2000.

[6] In functional decomposition, complex functions are recursively broken down into smaller and simpler functions, until you have functions that can easily be implemented. You also separate the data from the functions, which gives you a many-to-many dependency mapping between functions and data.

[7] For more information on Web Services for .NET platform, see Thuan Thai & Hoang Q. Lam, *.NET Framework Essentials*. O'Reilly, 2001.

[8] For more information on how to organize your team, and other management issues, see Walker Royce, *Software Project Management: A Unified Framework*. Addison-Wesley, 1998.

[9] For more on this topic, see Philippe Kruchten, "From Waterfall to Iterative Development: A Challenging Transition for Project Managers." *The Rational Edge*, December 2000. http://www.therationaledge.com/content/dec_00/m_iterative.html

---

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

Business Modeling with UML: The Light at the End of the Tunnel

by **Bryon Baker**
Product Manager
Requirements Management
Curriculum
Rational University

In the current economic climate, no software development project can afford to solve the wrong problem. Especially in companies for which a software system is *the business, the cost of failure could be the solvency of the business itself.*

How to ensure that you're solving the right *problem? In its influential* CHAOS Report, *the Standish Group*[1] *specifies a firm, basic requirements definition and a clear statement of business objectives as essentials for project success. Without a business model as a focal point for insight and discussion around business objectives, it is difficult to develop an essential understanding of exactly what processes a software system must support. Just ask five different people in your organization how the business operates; you'll get five different answers. Hence the value of a business model: It provides consensus on the problem you're trying to solve.*

So, great, you're committed to modeling your business before you undertake that next big technology project. But why choose the Unified Modeling Language (UML) as the basis for your business model? We'll explore its virtues for this purpose in this article.

What *Is* Business Modeling?

In a generic sense, business modeling is a set of activities whose goal is to help you visualize and understand business processes. As applied to software systems or other systems, business models act as a blueprint that will guide you as you construct the system. In effect, the model

becomes an operational description of the business that can illuminate value/cost tradeoffs, priorities, and risks. This level of understanding is frequently essential to helping system analysts, designers, and developers make informed decisions about the processes they're automating and the technologies most appropriate for implementing them.

There are three basic reasons why you might need to model a business:

- **To re-engineer a business.** This involves analyzing and fundamentally re-thinking how the business operates and interacts with the outside world. For this highest-risk form of process and system design, business modeling is absolutely essential.
- **To improve a business process.** I think of this as "bite-sized" process re-engineering, in which you identify an area of the business with the most critical problems and target your analysis to that area. The goal here is usually to streamline how the business works, and/or to enhance its competitiveness. Its micro-focus means it usually entails lower risk and a higher success rate than re-engineering.
- **To automate a business process.** This is the level of endeavor we customarily associate with software development.[2] The goal here is to reduce the resource requirements associated with a process by enabling more of it to happen without human intervention. In this context, a model of your current business allows you to understand the environment in which a software system will function.

Whether you plan to re-engineer the business or just automate an existing process, business modeling is the first step toward defining a software system that will solve the precise business problem you want to address. If yours is an established small company with a simple organizational structure, or if your team is attempting to automate or improve a very well-understood business problem, then you may realize few benefits from business modeling. But say yours is a startup operation, or you're entering a new business area. Here, modeling can provide critical insight into where automation can deliver the greatest ROI. If you plan to work with legacy systems or manual processes, or if you work in a large enterprise that is attempting to automate a mission critical process, then business modeling is almost certainly among the best things you can do to support project success.

Who performs business modeling? If the goal is to re-engineer the business, then modeling is most often undertaken by business process analysts who will develop new business architectures. If the goal is process improvement, then modelers may be business designers who need to describe new business processes. For a typical software development project, responsibility for modeling may fall to requirements analysts, whose job is to validate system requirements

The Business Value of UML

UML is generally thought of as a language that helps you visualize,

specify, construct, and document software-intensive systems. However, the designers of the UML made it possible to enhance the language so that it can be applied to many different domains. This feature has enabled it to rapidly emerge as the premier language for both traditional business modeling and the system analysis and design that follow.

UML has been successfully applied to the modeling of just about any system you can think of, from data structures to embedded real-time systems, to XML (Extensible Markup Language) schemas, and to real-world organizations from family businesses to multinational enterprises. So it's not surprising that business analysts find UML very handy for visualizing organizational processes.

But UML is far more than just handy; it's the most powerful, flexible notation available for business modeling today. It helps you manage complexity, reduce development time, and improve system quality. And there are six main reasons why:

1. UML provides a common language for business analysts and developers.
2. UML is visual.
3. UML is object-oriented.
4. UML describes business processes both structurally and dynamically.
5. UML helps you focus on the customer.
6. UML helps you derive better system requirements.

Let's look at each of these reasons more closely.

1. UML Provides a Common Language for Business Analysts and Developers

UML enables you to model business processes using the same symbols, diagrams, and other forms of notation that software teams use to model the systems to create or automate those processes. This ability to work using a common language enables something that was not previously possible in software development: Business people and systems people can communicate!

The idea sounds simple enough, but until UML was applied to business modeling, there was always a disconnect between the design of the business and the design of the systems within it. The UML largely eliminates that separation in perspectives between developers, business management, and customers.[3] When you start with a UML-based model, key business considerations are more likely to be included in the system requirements, and that ultimately leads to a system that serves customers better.

2. UML Is Visual

In my opinion, the perspectives offered by flowcharts and spreadsheets are too linear to effectively model a business. A limited viewpoint can yield false impressions of what drives a process, where it is bottlenecked, or how information flows. To fully model a business, you need to answer time-honored questions about who, how, what, why, and when things are happening.

UML allows you to visually model how your business operates. The *who*, the *what*, and the *how* are all represented in terms of symbols and diagrams. In this context, relationships, activities, and the flow of information, goods, and services all become more obvious. These visual representations enable you to see bottlenecks, understand how information flows (or doesn't), and determine who does what with your business information. For example, a visual model makes it plain when too much information must flow through a single point, pointing to a potential need to redirect some of that flow to other processes.

A well-constructed visual model of a business -- such as those the UML makes possible -- will answer all of these fundamental questions for you:

- Who are your internal and external customers? (I.e., Who will benefit from this business system?)
- Where can a system best add value to your business?
- What events within and/or outside the organization trigger each business process?

OVERVIEW OF BUSINESS MODELING WITH UML

Let's take a quick look at how you construct a business model with UML, with an eye toward how the process delivers business benefits.[4] Though tremendously powerful, a UML-based business model is conceptually straightforward. It consists of two key elements:

- A business use-case model, which describes the actions (i.e., a workflow) that a particular business process performs in order to deliver value to a business customer.
- A business object model, which describes how a business process will accomplish the actions described in the business use-case model. This model helps you see how people and things (goods, information, etc.) are related, and how they interact to perform the process in question.

Other key concepts include the *business worker* and the *business entity* (Figure 1). A business worker represents a role or set of roles in the business. A business worker interacts with other business workers and manipulates business entities, while participating in business use-case realizations. A business entity represents a "thing" handled or used by the business.

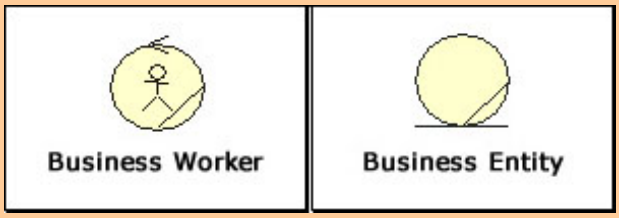


Figure 1: Business Worker and Business Entity in UML Notation

In general, a business use-case describes what the business does (i.e.,

- What end products do those business processes produce?
- What internal deliverables do those business processes entail?
- What is the organizational structure that supports the business?
- What roles and responsibilities within that organizational structure should be part of the system?

Having answers to these questions -- and the ability to follow up on related concerns -- will help you gain valuable insight into whether the changes you're about to make will solve the right problem, and do it in the way that delivers greatest value to the customer.

3. UML is Object-Oriented

UML is used to diagram software systems from an object-oriented perspective. When you model a business with UML, your business concerns are clearly outlined relative to the appropriate *business objects*, in the form of a *business object model* (see Sidebar).

Objects are entities that parallel objects in the "real world": They have some properties (such as name and address), relate in various ways to other things around them, and will exhibit some sort of behavior when acted upon. Object-oriented models can therefore very closely approximate actual business objects and systems, even to the extent of portraying how different parts of the system work together dynamically. This is true, in a nutshell, because what it delivers to its customers); a business object model describes how it does it. You specify business use cases first, and then use these to derive the business object model. A business use-case specification takes the form of a text description, along with one or more UML diagrams. A business object model can include class diagrams, activity diagrams, and business interaction diagrams -- all of which depict relationships and interactions among the entities that perform the activities of the business.

Figures 2 and 3 illustrate business use-case and business object models as applied to a business process automation problem. Note that they include two types of actors: Business actors (black stick figures with a yellow face crossed by a line) show interaction with business use cases; system actors (red stick figures) show interaction with system use cases.

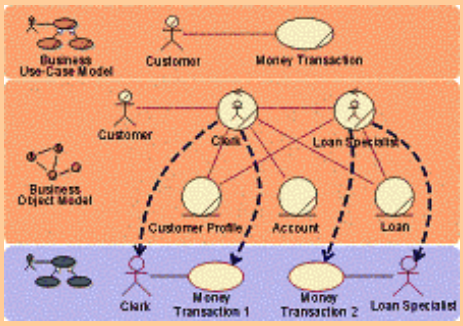


Figure 2: Business Models and Actors to the System

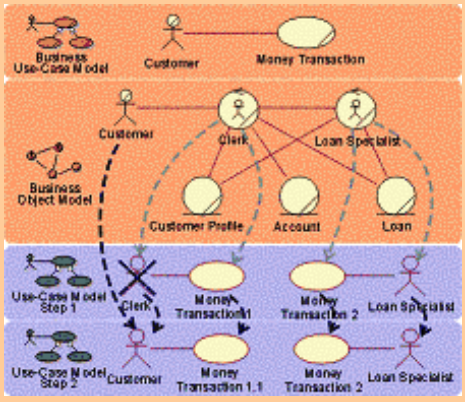


Figure 3: Automating a Business Worker

Here you can clearly see how the work you invest in producing a business

model will help you develop your system requirements.

the objects that comprise UML models reflect *real-world* entities very closely. This means object-oriented models are therefore more tangible and well described, and overall more flexible and intuitive.

Take, for example, an object-oriented description of a worker's role. It would likely include information about behavior, such as job responsibilities with respect to other workers in the business; state, such as how the role relates to other objects and to the process; and properties, such as quality checklist items or a job description.

4. UML Describes Business Processes Both Structurally and Dynamically

The more automated a business becomes, the more its interrelated software systems form the core of the business value it delivers to customers. Understanding how those interwoven systems interact -- and how to guide their successful evolution in response to a changing business landscape -- is critical to success. The value of UML in this context is that it lets you look at things inside the business both structurally and dynamically. The UML has many different diagram types that enable you to represent information from many different angles. These orthogonal views will completely describe your business and provide *meaningful* information to the varying levels of stakeholders.

UML use cases (see Sidebar) describe business processes from the viewpoint of how actors (*customers* for our purposes) use the business to attain some goal. A use case will describe a complete business process from start to finish, from an external perspective. This is in direct contrast to many "traditional" business-modeling methodologies, which *decompose*, or break down, processes along functional lines.[5] In such a decomposition, the whole is not necessarily the sum of its parts. The descriptions of the processes become isolated from each other, and their relationships become less apparent. When this happens, the benefit these isolated descriptions have to the business are less quantifiable, and their value therefore starts to become subjective.

The value of UML becomes especially evident when you actually turn a business modeling process loose on a business problem. With conventional methodologies, key issues can be easily misinterpreted, because these methodologies often overlook dynamic relationships that impact the process.

UML, in contrast, provides rich alternatives for describing the dynamic, real-world connections that are part and parcel of modern business systems. The multiple views that a UML model provides really help you understand a problem from all sides. Also, because UML provides a common language and a shared basis for discussion among development and business staff, there's no need to translate the model into words and risk diluting its meaning.

5. UML Helps You Focus on the Customer

The UML business modeling methodology revolves around business use cases, which emphasize how a business process delivers value to the customer. This customer-centric perspective helps you conceive an external view of the system you're creating. This is an especially powerful and useful approach in e-business contexts, where an external focus is even more critical to success.

In UML, a business use case is defined as:

> A sequence of actions performed by the business that yields an observable result of value to a particular business customer.

This parallels the influential work of Hammer and Champy (1993),[6] who define a business process as:

> A collection of activities that takes in one or more kinds of input and creates an output that is of value to a customer.

Business processes are what drive automated business systems; they comprise the tangible behavior of the organization. Because business processes are the means for delivering value to customers, the success of an organization hinges on the performance of its business processes (whether they are automated or not).

UML business use cases are built from the ground up to tell a story about how a business process works and how a customer uses that process. What better basis upon which to design a software system for optimal process performance -- and ultimate business success?

6. UML Helps You Derive Better System Requirements

With its rich descriptions of relationships among components, business actors, and other entities, UML makes it far easier than other modeling approaches to identify where a system best fits within a business context. This, in turn, enables you to more readily validate your software requirements. The end result is a working business system that solves a specific business problem, meets the real needs of the business, and delivers optimal value to customers.

Moreover, the UML-based business models your team creates can serve as direct input to a requirements model. The Rational Unified Process® (RUP®) provides a direct mapping between a UML business model and a requirements analysis model. This level of integration "front loads" your system analysis and design efforts, and further compresses time to deployment.

The Bottom Line

The nature of software development is changing in response to new business demands. Everywhere you look, innovative organizations are re-thinking traditional business processes and extending key internal processes beyond organizational boundaries to reach suppliers, customers,

partners, and government agencies.

In a world in which the risks associated with the failure of key processes has never been greater, it's no wonder business modeling is catching on like never before. When business requirements are poorly defined, the resulting inadequacies compound themselves and reverberate throughout the integrated enterprise. With a clearly explicated model of the business, however, companies have a foundation upon which to build and deliver successful systems -- that ultimately deliver optimal value to customers.

UML surpasses all other alternatives for developing a solid business model that can be applied to the development of a successful software system:

- It unifies development and business teams by integrating the modeling and development languages across every phase of the effort.
- It facilitates the specification of a robust, component-based architecture whose design and development can be controlled, managed, and verified, leading to reduced costs and shorter development cycles.
- It helps, at every step in the process, to keep you focused not just on technical innovation -- or even greater efficiency -- but also on the value you're delivering to customers.

Notes

1 www.standishgroup.com

2 Business process automation is also the focus of the business modeling discipline in the Rational Unified Process®(RUP®).

3 A *customer* is someone or something that interacts with the business, via a business process, for the purpose of obtaining something of value from that business. This can be an end customer, a supplier, a trading partner, a prospect, a federal regulatory agency, or another part of the business that is internal to the organization. It all depends on where you draw the boundaries of your business modeling effort.

4 For a more complete introduction, see "Introduction to Business Modeling Using the Unified Modeling Language (UML)" by Jim Heumann in the March, 2001 issue of *The Rational Edge*.

5 It is also worth noting here that business modeling using the UML and use cases helps support an iterative development process. Whereas functional decomposition techniques are not, because they require you to functionally decompose the entire business area of interest before you can move on.

6 M. Hammer and J. Champy, *Reengineering the Corporation*. HarperBusiness, 1993.

References

Ivar Jacobson, Maria Ericsson, and Agenta Jacobson, *The Object Advantage: Business*

Process Reengineering With Object Technology. Addison-Wesley Object Technology Series, 1995.

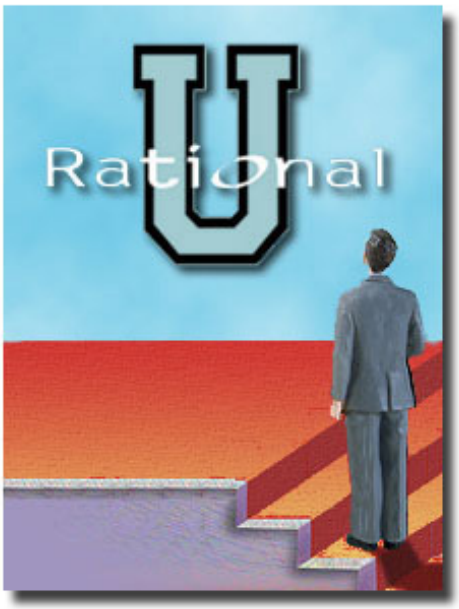
M. Hammer and J. Champy, *Reengineering the Corporation*. HarperBusiness, 1993.

Magness Penker and Hans-Erik Eriksson, *Business Modeling with UML: Business Patterns at Work*. John Wiley & Sons, 2000.

Rational Unified Process, Version 2002.05.00.

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

Rational University's OO Courses: New and More Targeted



Rational Software's education division, called Rational University, has recently revamped its entire object-oriented curriculum. The goal: To provide customers with courses that better fit their specific needs and that deliver more robust working artifacts for them to take home.

The new courses will be offered beginning in January 2002.

Shawn Siemers, Senior Product Manager at Rational, has oversight of the OO courseware for Rational University and works with various stakeholders -- instructors, customers, and his peers within many different groups at Rational -- to determine the content and architecture of the OO curriculum. The Rational Edge *asked reporter Johanna Ambrosio to interview Siemers so he could explain more about the recent changes.*

Q: Before we get into what's new, can you give us some background on the OO courses you currently offer?

A: The object-oriented (OO) courses are the second most popular offerings from Rational University, right after Rose Fundamentals. In any given year, approximately 25,000 students take at least one OO course. The technology is gaining popularity with our customers as a way of developing software. It's a very logical way of putting together a solution or solving a problem. You start with something large and break it down into pieces that fit together. If you're working on an order-entry system, you have a customer object and an order object, and all of these objects collaborate to form something real in the business environment. They each have attributes that mirror those of "real-world" counterparts. And when you're done, you have a model for the solution that's very consistent and

that goes from a grand scale to the micro level. OO starts with behavior, instead of starting with the data.

Q: What are some typical job titles for your OO students?

A: They can be anyone, from client/server and Web developers to data modelers. Some of these people are moving into OO development within their companies, and others are trying to upgrade their skills to meet today's demand for OO-savvy developers. Data modelers take the courses to be able to better communicate with object modelers, and to better understand how to do object-based modeling of data for relational and object/relational databases. We also have software managers who need to understand OO technology and terminology to different degrees, to be able to better work with team members and lead OO-related projects. Then there are members of the real-time community seeking to make a switch to OO development. Each of these students may have different educational needs at different stages of their career.

Q: Okay; let's talk about what's different about the new courses. Have you changed your fundamental approach to teaching OO?

A: Not really. The changes improve on the framework we've had in place for three years, which has been an outstanding success; we've gotten high marks from instructors and students. Based on their feedback, however, we also learned that there were ways we could improve the courses. The new OO curriculum really speaks to three areas where we felt we needed to make some changes.

- First, we have a range of student profiles. We've always welcomed the pure beginners and continue to do so. Another student profile includes people who understand the basic technology underpinnings but struggle with how to apply OO. Finally, there are students who are quite advanced in their understanding but need some help in how to apply OO analysis and design to a specific implementation environment. By seeing what happened in the classroom and based on feedback we were getting, we realized that we should be offering beginning, intermediate, and advanced classes to meet different needs.
- Second, students told us there was a platform gap. They wanted a course platform that was compatible with their platform back home, so they could apply what they learned about OO design in class and then continue to use and learn from those lessons at home.
- And third, we knew we needed to reduce the overlap among the courses we had been offering.

Q: Tell us what you did to address the different levels of understanding among your students.

A: For beginners, we've redesigned our basic course, "Principles of Object Technology," to become "Fundamentals of Visual Modeling with UML." It continues to be a very basic, elemental course. We start students off at the ground floor, and no prior OO knowledge or experience is required or assumed.

For everyone else, we had been covering analysis and design in one four-day course called "Object-Oriented Analysis and Design." We were trying to do too much, and what wound up happening was that the less experienced people would be lost in the second half of the course, and the more advanced people would be bored in the first half. There was always a disjoint, because half the class was either falling off a cliff or totally under-challenged.

So we custom-built two new courses instead of that one: "Object-Oriented Analysis with UML" for those above the novice level, and "Object-Oriented Design with UML" for the expert. Now, you will be able to literally sit through the courses for all three levels -- basic, intermediate, and advanced -- with no overlap. They can take you all the way through the design of your application.

Q: What did you do about the platform issue? How can students apply what they learn to specific environments back home?

A: We've bridged the platform gap by applying our OO courseware to Sun's J2EE environment. Let's say you've gone through Analysis, and you have these OO classes that you've developed in the course. How do you get your simple model to generate real code that you can keep? With J2EE, we can tell you when a certain class will turn into an Enterprise Java Bean, what type of EJB it will be, and how to apply it to both Web and traditional desktop interfaces.

Q: What if you're working with Microsoft's .NET environment? Or some other?

A: The "OO Design with UML" course for Microsoft's .NET environment will be offered soon -- probably early in 2002. Right now, J2EE and .NET are the two big players that Rational will be involved with strategically as development platforms, but that's always open to change.

That said, many of the principles we discuss in the courses are the same, regardless of what environment you will ultimately be developing on and for. The platform won't change the structure of our courses; it will just change some of the specific advice we offer for identifying and dealing with issues raised by the use of these platforms. A lot of design is agnostic to the implementation environment, but there is a point where you get

close to code and have to recognize there's a development platform out there.

Q: How do the new course materials dovetail with the new release of Rational Suite?

A: The new version of Rational Suite, released in November, includes a lot of J2EE support. So the decision to make J2EE the platform of choice for our OO courseware was not incidental; there's a strategic thrust within the company as a whole to provide solutions for the J2EE developer community.

Also, we build all the screen shots in our courseware with the latest suite version of Rose. Although our OO curricula are not tools courses, they can help Rose users get a much better sense of how to use the tool. It gives them a context for the problems they're trying to solve with Rose.

The other thing we've done on the product front is to make the OO courseware compliant with the newest release of the Unified Modeling Language: UML 1.4.

Q: How did you go about developing this new curriculum? Who wrote the course materials?

A: The whole process started with the many mountains of student evaluation forms on my desk. I read through these forms and then worked with a steering committee within Rational that's made up of subject matter experts and others. After we agreed on the overall vision for the curriculum, we designed particular modules and their objectives. I'd send off each module to the steering committee to get their feedback.

I wrote the courses, using tons of source materials. There are fifteen books on my shelf about UML alone. I also asked for many, many reviews of the technical content as well as the instructional approach and pedagogy. Developing these courses took about a year.

Q: Who will teach the courses?

A: Rational field personnel. Or a customer can take an OO class through a Rational partner, and in that case a certified instructor will teach the class.

Q: Have you beta tested the new curriculum on actual customers?

A: We've tested the courseware on internal Rational students -- new hires

and other employees, from novice to advanced beginner. We also asked our most experienced instructors to come in, and we conducted the courses just like we would with external customers. That led to many discussions about timing, overall delivery, the types of information that would be helpful to include in the instructor's notes, and so on.

We performed those tests in mid-October and passed with flying colors. All the participants told us that the courses were a vast improvement over what we had before.

Q: What do you believe customers will find most valuable about the new courses and your new approach?

A: Mainly, customers are going to find that we're really meeting their specific needs now. They'll be in a class of peers -- if they're into design and want nothing to do with analysis, this will be better for them because the course will answer questions that were never addressed previously and get into much more depth in their interest area.

We're betting that students will really appreciate what we've done here. They won't be overwhelmed with a lot of information they don't need, although their brains might still hurt at the end of the week.

Q: Aside from eliminating the platform gap, how will these changes help students be fully productive as soon as possible?

A: We design our materials for the 80/20 rule. Most students will retain only 20 percent of what they hear in the classroom. So we recognize that fact and design our courseware to be more valuable after the course is over. Our student books are very descriptive; they include copies of every slide we use in the course's presentation, along with extensive notes.

Each module in the course contains a UML model built using Rational Rose, so when you do get back to your site, if you're using UML, it will be very easy to say: "This is what I'm doing on my real project; how can I apply what I've learned to that?" We've built models for each module, so students see how the design model is built gradually. We have them design their own model in the course -- and that experience helps them fill in the gaps and be more productive later on.

We've also built in extra lab time to help people get more hands-on experience doing what they're interested in. And now that the course is more targeted, we spend a lot less time on helping people catch up.

Rational University meets you where you need to be. If you're fairly advanced, then you can go straight to the design course. Our goal is to target specific learners and define curricula with a clear beginning and a clear end, allowing you to fit in where you feel you need the most help.

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

The Rational Unified Process for Systems Engineering
PART II: Distinctive Features

by **Murray Cantor**
Principal Consultant
Rational Software Corporation

In Part I of this article, published in last month's issue of The Rational Edge, *I introduced RUP SE, my technique for applying the Rational Unified Process® (RUP®) to address the specific needs of systems engineering projects. RUP SE is not a product but a deployment service provided by qualified Rational consultants. In Part I, I talked about the similarities and differences between RUP SE and the common RUP implementations, and began explaining in more detail some of RUP SE's innovative constructs. In this installment I'll continue that explanation, focusing first on requirements analysis in RUP SE. I'll also explain in general terms how RUP SE impacts the management of systems development projects.*

Requirements Analysis in RUP SE

RUP SE follows the Unified Modeling Language (UML) and RUP in distinguishing two types of system requirements:

- **Use cases**, which describe services provided by the system to its actors.[1] Use cases capture the system functional requirements, and may also include associated performance requirements.
- **Supplementary requirements**, which cover non-functional elements like reliability and capacity.

A critical goal for any successful systems engineering project is to specify a set of system use cases and supplementary requirements that, if met, would result in a system that accomplishes its business purpose.

The purpose of requirements analysis is to determine requirements for all the Analysis level architectural elements. The basic process for deriving requirements for these elements is well known, and common to several systems analysis methodologies:

- Determine the requirements for a given model, such as the business model.
- Decompose that model into elements, and assign roles and responsibilities to the elements.
- Study how the elements collaborate to carry out the model requirements.
- Synthesize the analysis of these collaborative interactions to determine requirements for the elements themselves.

In the case of the business model, for instance, the RUP SE method for deriving system requirements is to partition the business into the system and its actors. The next step is to study the means by which the system and its actors collaborate to meet the business requirements. This, in turn, enables you to extrapolate system requirements. The discussion below explains more about how this basic methodology is applied to systems.

More About Model Levels

For modeling purposes, any system architecture exercise is predicated on levels of specification. As the architecture is developed, it evolves from a generalized, low-detail specification to a more completely described, detailed specification. Like the RUP, RUP SE defines four architectural models:

- The Business Model expresses the business processes that the system supports.
- The Analysis Model reflects the initial partitioning of the system into its primary elements, based on what it needs to accomplish and how that effort should be distributed.
- The Design Model expresses the realization of the Analysis Model in terms of hardware, software, and people.
- The Implementation takes the Design Model to the level of specific configurations.

Use-Case Flowdown

In the RUP SE Analysis Model level, the system architecture elements we're concerned with are subsystems, localities, and processes. The RUP SE activity for deriving functional requirements (i.e., use cases) for these analysis elements is what I call *use-case flowdown* -- a key area of departure from the RUP. The outcomes of this activity are:

- A use-case survey for subsystems.
- A survey of hosted subsystem use cases for localities.
- A survey of realized subsystem use cases for processes.

I begin the use-case flowdown activity with the familiar RUP activity of choosing an architecturally significant set of use cases, and then describing the interactions between the system actors and the system for each chosen use case. In this context, the system's responses to the actions of the actors are "black box"; that is, the descriptions of what happens make no reference to the architectural elements.

Table 1 shows a sample flow of events for making a sale in a retail store. Note that each step has an associated performance requirement.

Table 1: Sample "Black Box" Flow of Events

Step	Actor Actions	Black Box Description	Black Box Budgeted Requirements
1	This use case begins when the Clerk pushes the New Sale button.	The system brings up the New Sale, Clerk, and Customer screens, and enables the scanner.	Total response time is 0.5 second.
2	The Clerk scans the items and enters the quantity on the keyboard.	For each scanned item, the system displays the name and price.	Total response time is 0.5 second.
3	The Clerk pushes the Total button.	The system computes and displays on the screen the total of the item prices and the sales taxes.	Total response time is 0.5 second.
4	The Clerk swipes the credit card.	This use case ends when the system validates the credit card, and, if the card is valid, • Prints out a receipt; • Updates the inventory; • Sends the transaction to accounting; • Clears the terminal.	Total response time is 0.5 second.
		If the credit card is not valid, then the system returns a rejected message.	Total response time is 0.5 second.

In the next steps, you create the subsystem and process diagrams via standard Object Oriented Analysis and Design (OOAD) techniques. The Locality diagrams are found by similar techniques, except that the team specifies the major processing elements as discussed last month in Part I. You'll refine and re-factor the initial Analysis model throughout the flowdown process, as discussed below. For this reason, you can think of the initial model as a starting point, and you should not be overly concerned with its correctness.

The White Box View

Once the initial subsystem, locality, and process diagrams are in place, the subsequent steps depart from RUP activity. At this point, RUP SE revisits the interactions between the system and its actors by specifying how the analysis elements participate in carrying out the use case. As this version of the flow of events relates to specific architectural elements, I call it a "white box" view.

Table 2 shows a sample white box flow of events that parallels the steps in Table 1 above, adding a great deal of information:

Table 2: Sample "White Box" Flow of Events

Step	Actor Actions	Black Box Description	Black Box Budgeted Requirements	Subsystem White Box Description	White Box Budgeted Requirements	Locality	Process
1	This use case begins when the Clerk pushes the New Sale button.	The system brings up the New Sale Clerk and Customer screens, and enables the scanner.	Total response time is 0.5 second.	The Point-of-Sale Interface clears the transaction, brings up new sales screens, and requests that Order Processing start a sales list.	1/6 second	Point-of-Sale Terminal	Terminal
				Order Processing starts a sales list.	1/6 second	Store Processor	Sales Processing
				Point-of-Sale Interface enables the scanner.	1/6 second	Point-of-Sale Terminal	Terminal
2	The Clerk scans the items and enters the quantity on the keyboard.	For each scanned item, the system displays the name and price.	Total response time is 0.5 second.	The Point-of-Sale Interface captures the bar from the scanner. The Point-of-Sale Interface requests that Order Processing retrieve the name, price, and taxable status for the scanned data.	1/8 second	Point-of-Sale Terminal	Terminal
				Order Processing retrieves the name, price, and taxable status for the scanned data.	1/8 second	Store Processor	Sales Processing

				Order Processing adds the item to the sales list.	1/8 second	Store Processor	Sales Processing
				The Point-of-Sale Interface displays the item name, price, quantity, and item total on the clerk and customer screens.	1/8 second	Point-of-Sale Terminal	Terminal
3	The Clerk pushes the Total button.	The system computes the total price of the items and sales taxes and displays the total on the screen.	Total response time is 0.5 second.	The Point-of-Sale Interface requests that Order Processing sum the price and compute the taxes.	1/6 second	Point-of-Sale Terminal	Terminal
				Order Processing sums the price and computes the taxes.	1/6 second	Store Processor	Sales Processing
				The Point-of-Sale Interface displays the totals.	1/6 second	Point-of-Sale Terminal	Terminal
		The system validates the card, prints two copies of the credit card receipt, and closes out the sale.	30 seconds	The Point-of-Sale Interface reads the credit card data and requests that Credit Card Services validate the sale.	.5 second	Point-of-Sale Terminal	Sales Processing
				Credit Card Services requests validation through Credit Card Gateway for the given card number and amount.	28 seconds	Store Processor	Sales Processing

4	The Clerk swipes the customer credit card.			If the sale is approved, then the Point-of-Sale Interface prints a receipt for signature.	1 second	Point-of-Sale Terminal	Terminal
				The Point-of-Sale Interface requests that Order Processing complete the sale.	1/6 sec	Point-of-Sale Terminal	Terminal
				Order Processing requests that Inventory Control remove the items from inventory.	1/6 second	Store Processor	Sales Processing
				Inventory Control removes the items from inventory.	1/6 second	Store Processor	Store Accounting
				Order Processing requests that Accounting Services post the transaction.	1/6 second	Store Processor	Sales Processing
				Accounting Services updates the account.	1/6 second	Central Office Processor	Central Accounting

The purpose of the subsystem white box steps shown in Table 2 is to illustrate how the subsystems collaborate to carry out each black box step. The white box budgeted requirements map the budgeting of the black box performance requirements (see Table 1) to the white box steps. The Locality is the locality that hosts each white box step; the Process specifies which process executes the white box step.

If a white box step requires more than one hosting locality or executing process, then you can simply break the step into smaller steps, each with a unique locality and process.

When you assign white box steps to subsystems, localities, and processes, you make a series of design decisions, each of which helps flesh out the role that each analysis element plays in the overall system design. As your team makes these decisions, you may decide to re-factor the design, shifting responsibilities from one element to another within a given diagram.

The Subsystem Use Case Survey

Once you create a white box flow of events, the next step is to specify the subsystem use cases. You initiate this process by organizing the white box steps according to the subsystems they relate to. Then you sort the white box steps associated with each subsystem according to how they relate to one another. The result is a survey of use cases for each subsystem. (Recall that subsystem use cases specify what processing occurs at a given locality.)

Table 3 illustrates a subsystem use-case survey. Note that the survey includes both the locality that hosts each process, as well as the process that executes each white box step. This enables you to sort your subsystem use cases by locality or process, once you've completed the survey.

Table 3: Sample Use Case Survey

Subsystem Use Case	Description	Locality	Process	System Use-Case Name	White Box Text
Initiate Sales List	The subsystem initiates a list of items to be included in the sales transaction.	Store Processor e-commerce server	Sales processing	Enter a sale	Order Processing starts a sales list.
				Enter online sale	The e-commerce interface requests Order Processing to instantiate an ordering list and add the item to the list.
Add Product Data	The subsystem adds an item to a sales list when requested by the actor.	Store Processor e-commerce server	Sales processing	Enter a sale	The scanner data is sent to Order Processing. Order Processing retrieves the name, price, and taxable status from Inventory and updates the list.
				Enter online sale	The E-Commerce Interface requests Order Processing to instantiate an ordering list and add the item to the list.
Compute Total	...	Store Processor e-commerce server	Sales processing	Enter a sale	Order Processing sums the price and computes the taxes.

Check Availability		e-commerce server	Sales processing	Enter online sale	Order Processing requests availability status of all items from Inventory Control.
Complete Sale		Store Processor	Sales processing	Enter a sale	When Order Processing receives a valid sale, it returns a Valid status to the Point-of-Sale Interface. Order Processing sends a request to Inventory Control to remove the items from inventory. Order Processing sends the transaction to Accounting Services for posting.

A survey that delineates the use cases hosted at each locality is valuable because it expresses what computing occurs at each locality, along with associated performance requirements. This information helps you specify the hardware components to be deployed at each locality. Likewise, a survey of use cases that each process executes helps you specify the software components you'll need at each locality.

Creating Collaboration Diagrams

The text descriptions of the white box flow of events (see Table 2) also have an important purpose: They form the basis for a series of collaboration diagrams. These diagrams visually convey the traffic between the analysis elements. The objects in each collaboration diagram are proxy diagram elements. The messages that connect the objects represent the subsystem use cases.

Figures 1 and 2 show the subsystem and locality collaboration diagrams for the flow of events represented in Table 2. Figure 2 provides insight into how the subsystems interact. This gives you a very useful way to evaluate your subsystem design. If there is considerable traffic between a pair of subsystems, for instance, then it might make sense to combine them.

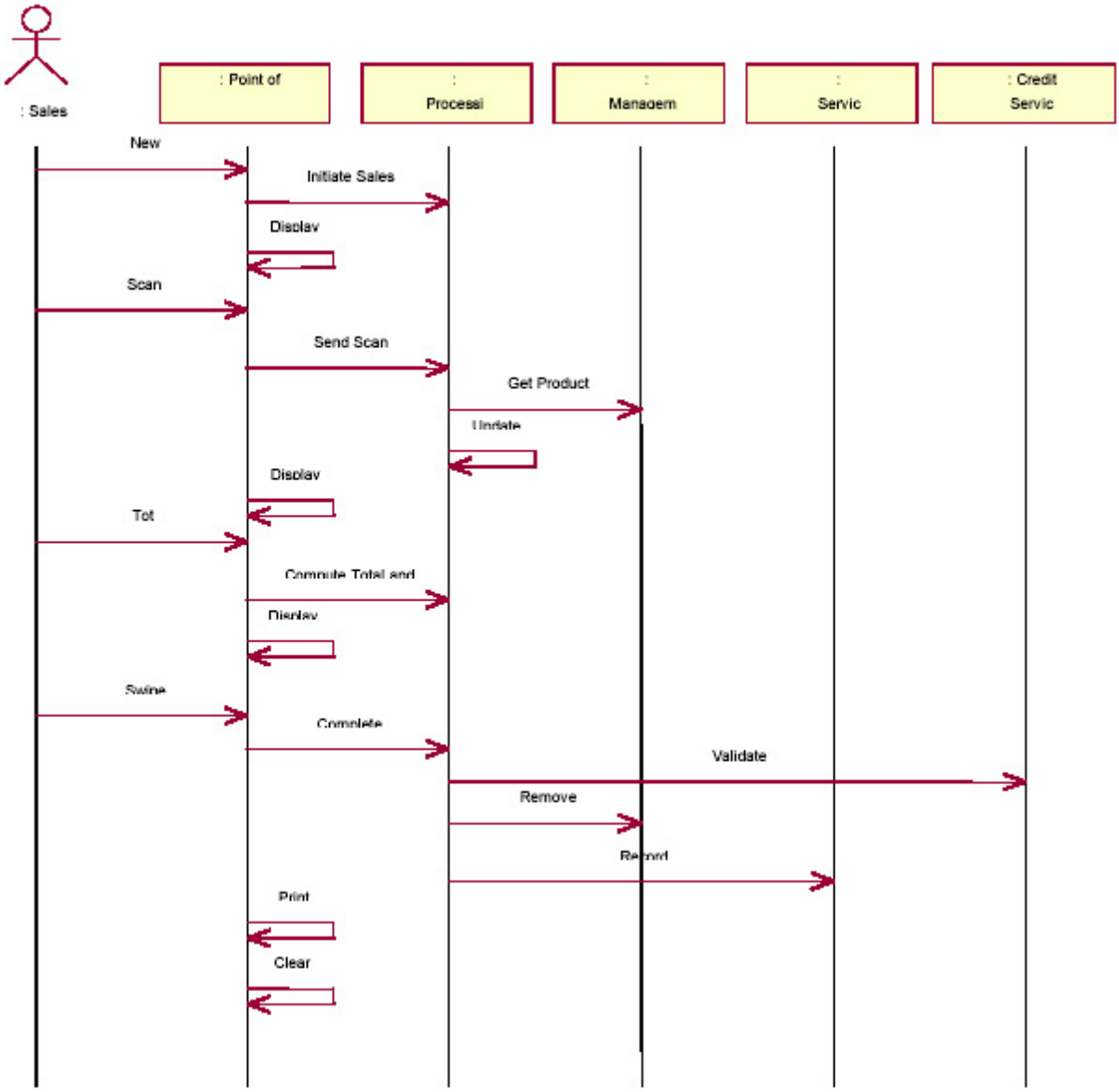


Figure 1: A Sample Subsystem Collaboration Diagram

Figure 2, on the other hand, tells you what data must flow between the localities. From this viewpoint you can better determine how the localities should communicate in terms of protocols, throughput rates, etc., with one another.

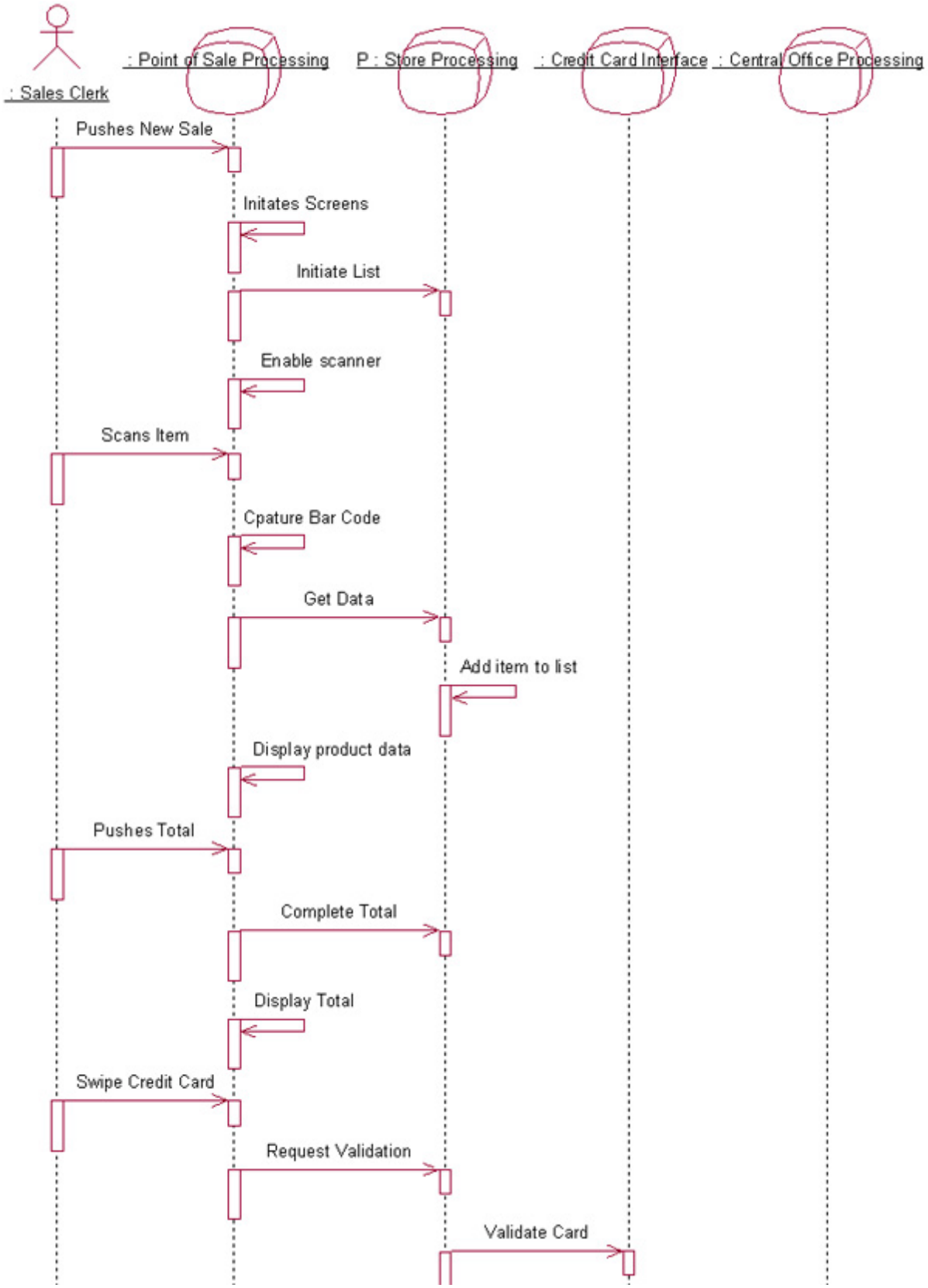


Figure 2: A Sample Locality Collaboration Diagram

Determining Supplementary Requirements

Fear not! I haven't forgotten about supplementary requirements. These are handled as part of the analysis process, in the context of an initial locality diagram that the system architects develop. This locality view provides a context in which you can begin looking at the system's non-functional considerations,

such as reliability, capacity, and permitted failure rates, as part of standard engineering practice.

This analytical effort results in a set of derived supplementary requirements for each locality element. These requirements form the basis for determining your locality characteristics.

When you complete the steps we outlined above, you will have done enough Analysis to begin designing the system. Remember that in an iterative development process, however, you may need to retrace these steps at a later time to adjust your locality requirements.

Component Specification: Moving from Analysis to Design

When you begin to specify the design of hardware and software components, you move from the Analysis level of the architecture to the Design level. You can now determine hardware components by analyzing the localities, their derived requirements, and their hosted subsystem use cases. Your goal is to produce a series of descriptor node diagrams (see Figure 1) that specify the components, servers, workstations, workers, etc., for each locality.

Note that this diagram does not specify the technologies that will be used to *implement* the components. You make those decisions by looking at cost/performance/capacity tradeoffs, which the descriptor diagrams help make more evident. Many systems will ultimately have more than one hardware configuration, each designed to balance these tradeoffs differently.

Figure 3 shows the descriptor view derived from a locality diagram shown in Part I of this series.

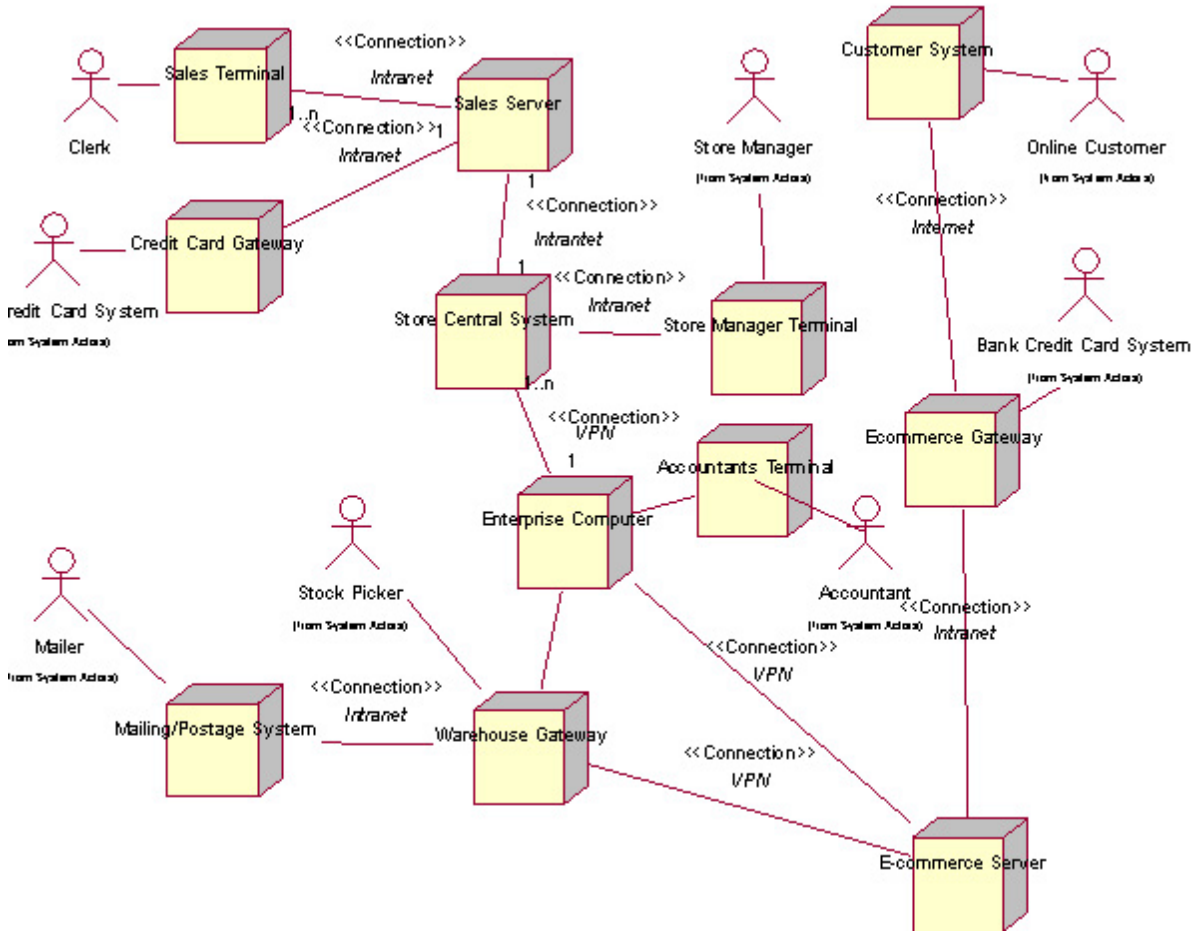


Figure 3: A Sample Descriptor Node Diagram

In RUP SE, you determine software components by specifying a set of object classes, and then compiling and assembling the code associated with those classes into executable files. A complete software component design must reflect a wide range of concerns, such as the locality where each component will run, and the hardware that will host each component. (Hence, you need to specify hardware components before you can fully specify software components.) The information required to specify software components is derived from several sources, including the survey of hosted subsystem use cases for localities and the surveys of executed use cases for processes.

The culmination of your Design efforts is a clear understanding of the hardware components, software components, and worker roles you'll need to implement the various system configurations. Now you're ready to move to the Implementation model level, where you begin to choose specific technologies to implement your design: What server platform? What database application? And so on. At the Implementation level, a single deployment diagram describes the hardware and software components of each system configuration.

RUP SE and Project Management

Project management is where the rubber of RUP SE meets the road of your organization. RUP SE impacts project organization and system development, integration, and testing in many of the same ways that RUP does, but with changes and additions that reflect the complexity of systems engineering projects.

On a typical RUP SE project, the organization is made up of several development teams, each with a project manager and technical lead:

- The **enterprise modeling** team analyzes the business case for the project and generates business models.
- The **system architecture team** works with the enterprise modeling team to create the system context and derive system requirements.
- The **project management team** looks after typical project issues such as reviews, resource planning, budget tracking, etc.
- The **integration and test team** receives each iteration's code and hardware components from the development team; builds the software components; installs the hardware and software components in a controlled setting; and conducts system tests.
- The **subsystem development teams** each design and implement one or more subsystems.
- The **hardware development and acquisition team** is responsible for the design, specification, and delivery of the physical systems, based on the localities.
- The **deployment operations and maintenance team** handles operational issues and liaises with users.

Concurrent Design and Implementation

Because it allows you to break systems down into subsystems and localities and their derived requirements -- any of which can be the focus of concurrent design and development -- RUP SE can scale to handle even the largest projects. Subsystems can be assigned to separate development teams, for instance; localities can, in parallel, be assigned to hardware development teams. Each team works from the appropriate use- case survey to develop their part of the design model and implementation models. In this way, the design and implementation of various design elements can proceed in parallel.

Iterative Development, Integration, and Testing

The iterative project lifecycle driven by RUP SE and RUP differs significantly from the serialized (a.k.a. *waterfall*) process typical of many organizations. With the iterative approach, the system is integrated and tested at each iteration; and each iteration adds functionality. The final iteration yields a fully tested system ready for transition to the operational setting.

When teams use a waterfall, or serial activities approach, workers are typically assigned to a project until their artifacts are complete. Engineering staff, for example, might complete the specifications, hand them off to the developers, and then move on to another project.

By contrast, no such handoff occurs in RUP-based projects. Instead, the specifications (and other artifacts) continue to evolve throughout the project lifecycle. Therefore, staff responsible for artifacts such as the requirements database and UML architecture will be assigned to the Development phase for its duration.

The content of an iteration, as captured in the RUP SE system iteration plan, is determined by the use cases and supplementary requirements for the components slated for development in the iteration. Each iteration is tested by an appropriate subset of system test cases. Subsystems and localities have derived use cases that form the basis for derived iteration plans.

In an iterative lifecycle, the role of the testing organization differs from its role in a traditional, serialized lifecycle. Rather than spending most of their available time planning for an overall system integration effort at the end of the lifecycle, the testing team spends time on integrating, testing, and reporting defects for each iteration.

More Specific Best Practices

RUP SE embodies all the most important fundamental RUP parameters: the four-phase lifecycle; the process disciplines; the iterative development approach; and the use of UML for visual modeling. This enables RUP SE to deliver all the advantages of RUP best practices, while providing a sound methodology for addressing system engineering issues. The most compelling benefits of RUP SE for system development teams include:

- A common modeling language and a unifying process support infrastructure for ongoing, evolving collaboration among business analysts, system architects, engineers, software developers, hardware developers, and testers.
- Comprehensive, visual perspectives, in the form of models, views, and diagrams, which make it possible to continuously verify and address system quality issues in the context of a component-centric process.
- The ability to visually model systems, thanks to the inclusion in RUP SE of UML artifacts for systems architecture and specification.
- Scalability from small systems to the largest systems engineering projects.
- Support for concurrent design and iterative development of hardware and software components.

I believe that RUP SE, like RUP, will help teams get a better handle on complex, evolving requirements; discover and mitigate development risks earlier; reuse more components; improve and ensure system quality; reduce project costs; and compress project timeframes (if only through better communication!). In short, it enables systems engineering projects -- and the organizations that depend on them -- to succeed.

Notes

1 An *actor* is any external entity that interacts with the system, such as a user or another system.

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

Book Review

Jack: Straight from the Gut
by Jack Welch with John A. Bryne

Warner Books, 2001
ISBN: 0-44652-838-2
Cover Price: US$29.95
(496 Pages)

Whether you love him or hate him, there's no disputing that Jack Welch delivered some pretty impressive results during his twenty-year tenure as CEO of General Electric (GE). Yes, that tenure occurred while our country was undergoing one of its biggest economic expansions ever, but there were plenty of companies that didn't perform as well (or at all) during this twenty-year span. It's also true that Welch either laid off or fired a large portion of the GE workforce (when you read reviews of this book posted on various Internet sites, it is pretty easy to guess which ones came from people who were fired!) during his reign in order to get the company's cost structure under control. But that helped him create enormous wealth and value: GE was the first company in the world to achieve a $500 billion market capitalization.

Welch is not perfect, and he would be the first person to tell you so. He agonized over the Kidder-Peabody acquisition and trading scandal, and to this day calls it his biggest failure (although the Honeywell debacle comes in a close second). He's brash and to the point, and has been slapped with a lot of unflattering labels. But this autobiography provides plenty of evidence that, in addition to being a top-notch executive, Welch is also a competent teacher who invested his time wisely in developing good people. The proof that he was an outstanding mentor and coach is that many of his direct reports left GE to become CEOs of some of the nation's largest companies: Allied Signal (Lawrence Bossidy); Home Depot (Robert Nardelli); and 3M (James McNerney, Jr.).

Although there are many books about Jack Welch on the market offering many different perspectives on the man, the one thing they agree on are Welch's basic messages, which have remained consistent over the years. No matter what you read about him and where you read it, these always come through loud and clear. This book is no exception, although it's redundant and slow in some places and probably could have been about one-third shorter. I found the messages so compelling that I took notes, so here we go:

- *Leaders must face reality*. Look at things as they are, not as you wish they were. Although he was the self-proclaimed master of the graphic chart, Welch emphasizes that the way to learn about what your business is really doing is by talking to your managers, not by poring over mounds of data. Numbers and metrics can be deceptive; they should *guide* your business, not *drive* it.
- *Leaders must have a feel for the business*. While coming up through the ranks of GE, Welch worked in many business units. When he was named General Manager of a $1.3B business unit at age 33, he became the youngest GM in GE history. Having an insider's understanding of his company's strengths and weaknesses -- in addition to a keen understanding of the markets -- gave Welch incredible confidence in his own decision making and ability to succeed.
- *Leaders must make the tough decisions*. Too often, Welch believes, management takes the easy way out instead of the right way. Throughout much of his tenure as CEO, Welch grappled with the need to ensure the company's long-term survivability as well as short-term viability. He had to make tough decisions that ultimately involved laying off thousands and closing down business units -- all for the sake of the company's future. In hindsight, he wishes he had made those decisions sooner.
- *Keep your strategy short and simple*. Welch decreed that all GE business units had to be number one or number two in their respective markets. Otherwise, they had to fix, close, or sell their businesses. This strategy was clear and crisp, and Welch communicated it constantly, at all levels of the company.
- *Continually communicate your strategy inside and outside the company*. For your vision and strategy to be effective, people must relate to them on both an emotional and intellectual level, and you can only achieve this through constant communication.

One of the best parts of the book deals with employee rankings. Welch's descriptions and observations about A, B, and C players clearly demonstrate that he spent a large portion of his time developing his people -- and that he held his managers accountable for team development, too. Although some view his style of removing the bottom 10 percent, the C players, as ruthless, he claims that it gave the remaining 90 percent a great feeling of achievement and performance. To prove that he applied this principle to employees across the board, regardless of rank and title, Welch recalls how he chastised Jeff Immelt, who later succeeded him as Chairman and CEO. If Immelt's business unit's numbers didn't get better, Welch had warned, he would have to make adjustments; Immelt retorted that if his unit's performance didn't improve, then he would leave of his own accord. Obviously, this didn't happen; Immelt's unit met its targets, and the rest is history.

In these economically troubled times, many managers have already been faced with tough decisions, and we're not out of the woods yet. Reading this book helps you realize that motivation can be fostered in many ways

through firm leadership that's grounded in the realities of a free and competitive marketplace.

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

Read a review of *Applying Use-Case Driven Object Modeling with UML: An Annotated E-Commerce Example* by Doug Rosenberg and Kendall Scott.

Book Review

Applying Use-Case Driven Object Modeling with UML: An Annotated e-Commerce Example
by Doug Rosenberg and Kendall Scott

Addison Wesley Professional, 2001
ISBN: 0-20173-039-1
Cover Price: US$34.99
(176 Pages)

If you need to create a system model for analysis and design, or if you need to review such a model, then this book might be for you. Doug Rosenberg and Kendall Scott have written a clear description of one effective way to get from requirements to code using the Unified Modeling Language (UML). Both experienced UML users and novice or occasional UML users -- like me -- and will find something useful here.

The book is designed to be a companion to Rosenberg and Scott's *Use-Case Driven Object Modeling with UML* (Addison-Wesley, 1999). It is readable on its own as long as you have a basic understanding of UML.

The authors describe how to do modeling in the context of the ICONIX process (see Figure 1), which is based heavily on Ivar Jacobson's Objectory process and will be familiar to Rational Unified Process®(RUP®) users. They describe the process in a logical order, beginning with developing a domain model, which is a "kind of glossary of the main abstractions." Once you have a domain model, you can use it to help generate the sequence of diagrams -- use-case, robustness, sequence, and class -- that make up the design model. Because the process uses only these four types of diagrams, it works well for novice UML modelers. It keeps to a minimum the amount of new knowledge that's required but provides a powerful tool for attacking system complexity. Whether you adopt the ICONIX process in whole or in part, you can benefit from the practical information in the book.

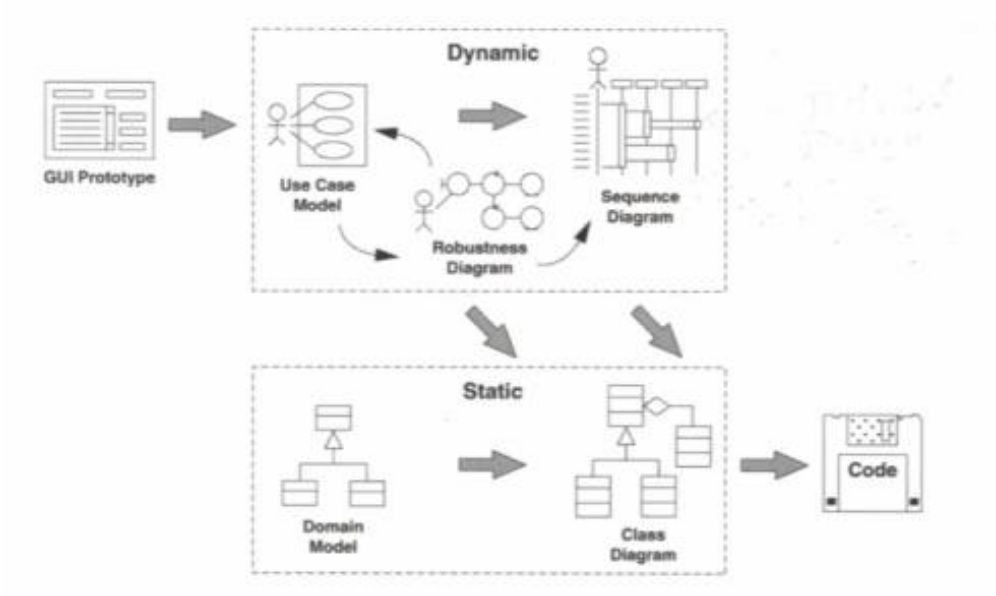


Figure 1: Flow of the ICONIX Process

The authors claim the process is flexible. They state early in the book that:

> Although the full approach presents the steps in a specific order, it's not crucial that you follow the steps in that order. Many a project has died a horrible death because of a heavy, restrictive, overly prescriptive "cement collar" process, and we are by no means proponents of this approach. What we are saying is that *missing answers to any of these questions will add a significant amount of risk to a development effort.*

I'm not sure, however, how you might do the steps in a different order. Also, as I read the book, I had a distinct feeling that the authors would never omit any of the steps, as they do not discuss alternative paths. Although I can see ways to use some pieces of the process and not others, in truth, I think the authors are quite dogmatic about the importance of doing everything in their process in precisely the order they lay out.

With three chapters devoted, respectively, to three different types of reviews, this book is an excellent resource for development teams that want to conduct effective reviews of the design as it progresses. The three reviews these chapters address correspond to the three milestones in the ICONIX process:

1. **Requirements Review (RR).** This review is based upon the use-case model, user interface prototypes, domain model, and initial packaging of use cases. The authors emphasize traceability from requirements to the use cases. They recommend avoiding the inclusion of items such as pre-conditions and post-conditions in use cases in order to keep them simple. I've found that there are many way of describing use cases effectively and suspect that most of them will be appropriate for the requirements review.
2. **Preliminary Design Review (PDR).** The preliminary design review occurs when you have done the analysis of your system and

produced robustness diagrams, which use entity, boundary, and control classes to describe the system. Robustness diagrams are not explicitly part of UML; they are derived from the Objectory process and easy to create using a collaboration diagram. Users of the Rational Unified Process will recognize these diagrams, which are part of the analysis model.

3. **Critical Design Review (CDR).** This is the last topic covered in the book. When you have finished the design by creating detailed sequence diagrams and class diagrams, and assigned all behavior required for all of your use cases' flows of events, you are ready to perform the CDR and translate it to code. If you get this far in the process, then you will have a very detailed UML model -- from which you will probably be able to generate a significant amount of code. You have to decide if you want to get to this level of detail.

Unfortunately, the authors seem to have a contract mentality when it comes to including customers in these reviews. In describing the preliminary design review they say: "You can think of PDR as representing a line beyond which customers are no longer welcome to actively participate in the process." I find this very troubling. I always want my customers involved, right up until the day I ship the software. Admittedly, I do frequently meet customers who are required to give a fixed price up front, and these customers might be able to adopt the posture of excluding customers after the PDR. This approach, however, sets up a needless barrier between the customer and the development team that can lead to misunderstanding as things change. Change inevitably happens throughout the process, and it is by working directly with the customer that you can best manage it.

As the title indicates, the book includes a running e-commerce example; the authors say they will describe the application in the Introduction (first chapter) -- which they do -- and then use it in each chapter's exercises. The chapter exercises present a faulty artifact (e.g., a use-case description, part of a diagram, or the like) along with one or more hints about the top ten errors that have occurred; it is then up to you to identify and correct the problems. Although I found these exercises to be quite simple and useful, they do not convey the feel of a consistent example. The Appendix is where the example is actually worked out. (In the Preface, the authors also provide a URL for obtaining the complete, worked-out example.)

One feature I like is that each chapter includes a list of the top ten common modeling errors that relate to the chapter's subject. I find these are great checklist items to use for my models, although some of the error descriptions are worded in a way that makes it difficult to remember they are describing things you should *not* do. In the chapter on robustness diagrams, for example, Error #6 states, "Allocate behavior to classes on your robustness diagrams." If you read this without thinking about it carefully, it's easy to mistake it for a guideline you should follow. I recommend that you reword these items if you plan to use them in a review checklist.

Overall, the book does deliver on its basic promises. The back cover copy

claims that "With the information, examples, and exercises found here, you will develop the knowledge and skills you need to apply use-case modeling more effectively to your next application." I think the authors do succeed in enabling readers to do this -- and that is a significant accomplishment.

- **Gary Pollice**
Evangelist
The Rational Unified Process

Read a review of *Jack: Straight From the Gut* by Jack Welch with John A. Bryne.

Keep Documentation Up to Date Throughout Development with Rational Rose

by **Robert Pierce**
Staff Technical Writer
Rational Software

Let's imagine we're inside the offices of a big, successful software company. The Program Manager is out on a road show, telling other product groups, partners, and customers about a new extensibility package that will help them provide better system solutions. Meanwhile, back at the ranch, the Lead Developer, Tester, and Technical Writer are having a casual conversation in the kitchen.

"Have you checked out the new interfaces we've added?" says the Developer. "That new API will really help developers do customizations that tie in to our stuff."*

"Yeah, I'm busting my chops right now to make sure everything works," says the experienced Tester, "but no one is really going to use this stuff if they can't understand it." He turns toward the humble Technical Writer. "That's your job -- to make it understandable."

"Right," says the Writer, "but you guys keep making changes. How am I supposed to keep the documentation up to date?"

The Developer and Tester smile and shrug their shoulders. "Well, that's your job," they say, chuckling as they toss their crumpled sandwich wrappers into the trash and walk away. The humble Technical Writer sighs, goes back to his cubicle, and puts his head in his hands. "How can I provide accurate documentation for something that keeps changing?" he asks himself. "I can't put it off until the project is completed -- the development team needs accurate information now, and users will need help the day they get that API. I wish there were a tool or an automated solution to help me!"

Rational Rose to the Rescue

Application Programming Interfaces, or APIs, are often the vehicle for introducing new software technologies. And as our little scenario reveals, trying to document a set of APIs can be maddening, because the details in an API typically change throughout the development lifecycle.

Often, technical writers use a manual process that is repetitive and painstaking. They have to search through code, then search for the correct classes and their properties and methods. Keeping up with changes to the interfaces becomes daunting (and costly), as numerous changes to class structures, methods, arguments, data types, and return types take place from iteration to iteration. The challenge is compounded by the fact that programmers rarely make documentation a top priority, and writers usually do not have access to the source code. Too often, the end result is that API documentation is done only when the API is complete. This leaves programmers without a single, reliable written reference source during the development lifecycle, and it can leave customers in the lurch for days or even weeks after they receive the product.

But our humble writer need not despair. There *is* an automated solution that's simple, efficient, and cost-effective. In this article, I will explain how -- by using the reverse engineering feature in Rational Rose® -- writers can work on documentation *as a project develops* and maintain accurate, up-to-date information.

Rose provides a single source for storing information in a variety of programming languages and deliverable formats, but it does not require a writer to have access to the source code. By storing content in a Rose model, you can readily generate API documentation using a SoDA template. Using this approach not only enables you to support API development efforts with consistent and accurate documentation; it also provides you with the means to create clear, consistent diagrams, using Rose.

Using the Reverse Engineering Feature in Rational Rose for Documentation

The process I am about to describe currently works for component object model-based (COM) data link libraries (DLLs), C++, and Java files. To produce API documentation, you start with a source code file (for example, a .jar or a DLL file) to deliver documentation in one of the following formats:

- Word document or pdf file
- A set of API reference topics as individual htm files, which can be linked in an XLS file to generate an html Help system

The basic process is shown in Figure 1, and the steps are detailed below.

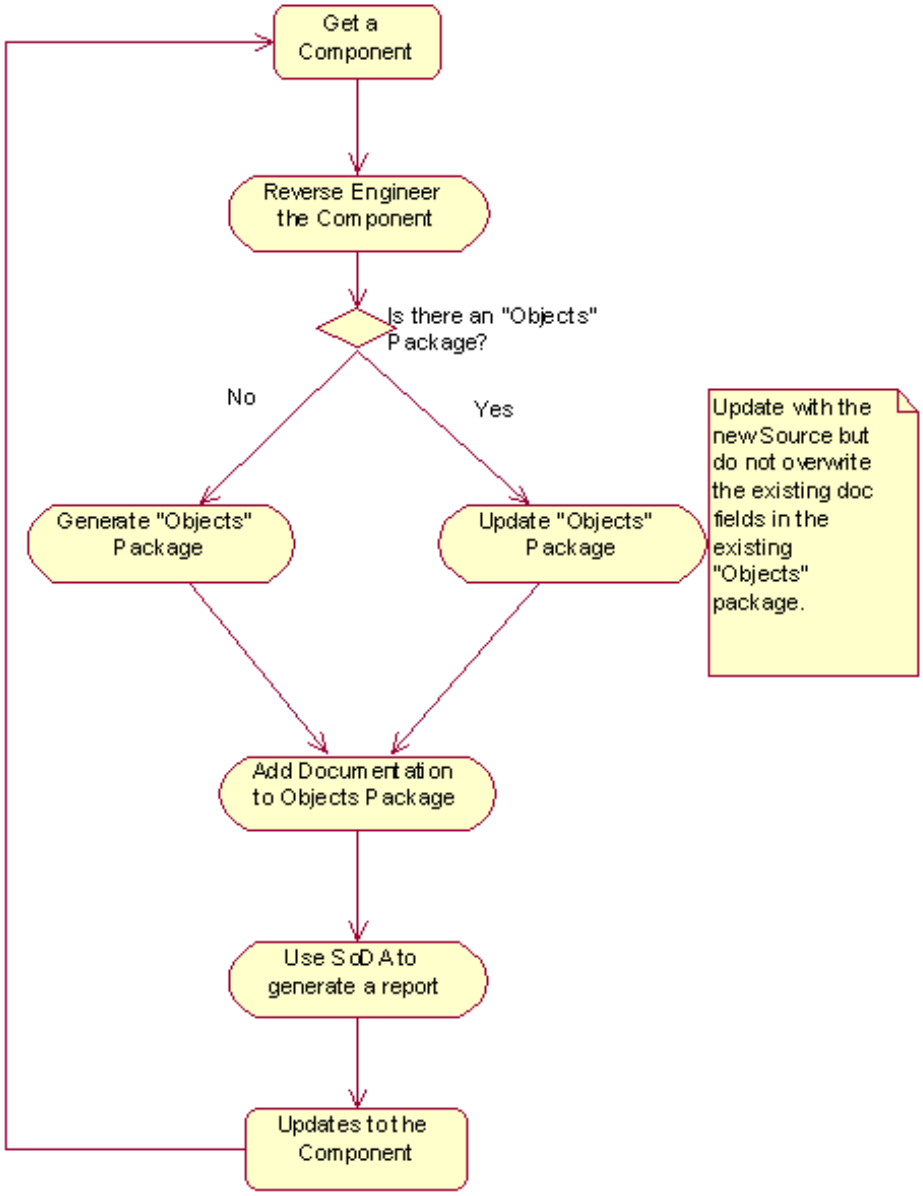


Figure 1: Basic Process for Producing API Documentation

1. Start with a Component file. The Component represents the code that the developers are working on.
2. Reverse engineer the Component (e.g., a DLL (COM) or .jar (Java) file).
3. Clean up the model by running a Rose script or Visual Basic application. This will:
    - Create a new Objects Package for a newly created reverse engineer source file. It copies code comments into the doc fields in Rose.
    - Update an existing model's Object Package when you reverse engineer an updated source. It does not overwrite any existing documentation fields that have content (that is, it won't overwrite doc content you may have added in documentation fields in the Objects Package, which is your

doc workspace).

- Generate changes made to an updated source file that you reverse engineer in log.txt file.

4. Add documentation content to the cleaned up Objects Package. Insert the content in documentation fields and add class diagrams. Name each class diagram the same as the class name (e.g., create a class diagram for the Locator class and name it Locator).
5. Generate a SoDA report, using a supplied template.[1]
6. Iterate with an updated source file. When you iterate, the Rose script (or Visual Basic application) automates the process of documenting the updated source by:
   - Updating an existing model's Object Package when you reverse engineer an updated source. It does not overwrite any existing documentation fields that have content (that is, it won't overwrite content you may have added in documentation fields in the Object Package, which is your workspace).
   - Generating changes made to an updated source file that you reverse engineer in the log.txt file.

The following sections provide more details about these steps.

Reverse Engineering an Existing Source File

The first step in automatically generating API reference documentation is to reverse engineer the DLL (or .jar) file containing the interfaces, objects, properties, and methods that you want to document. To reverse engineer a DLL file, do the following:

1. Start Rose.
2. Create a new file (click **File** > **New**).
3. Drag the DLL from Windows Explorer and drop it onto the blank main class diagram (or onto the Logical View Package). A dialog will appear that asks for Quick or Full Import.
4. Select **Full Import**. A COM-based model will be created from the DLL file.

Converting the Model

After reverse engineering a DLL file into a COM model, you can convert it into a Visual Basic model (i.e., VB data types and interface structure). You can also convert it into a C++ model (i.e., C++ data types and interface structure).

After reverse engineering a .jar file into a model, you need to convert it into a Java model (i.e., Java data types and interface structure).

Run the Rose Script (or Visual Basic application) to clean up your model. This creates a new Objects Package, where you can add content.

Note: This step can be customized with a Visual Basic application that allows you to select whether you are cleaning up a DLL or a .jar file and what type of API you want to generate. It can also provide other features such as generating a log file that notes all of the changes in an updated source.

Adding Documentation Content in Rose

Use Rational Rose to add descriptions for all objects in the API. For example, in the screen shown in Figure 2, you can add descriptions to each of the classes in the Objects Package.

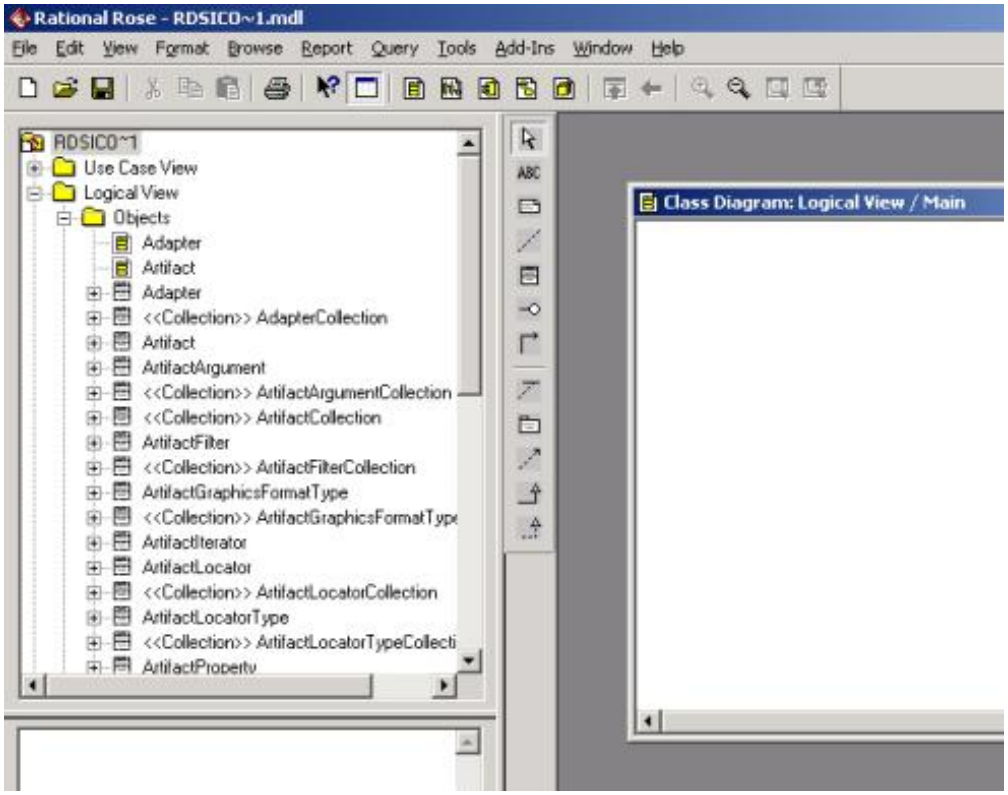


Figure 2: Classes in the Objects Package

Using the documentation fields for each object specification, you can create single-source descriptions for the following features of an API:

- Classes
- Attributes
- Operations
- Parameters of operations

Figure 3 shows a documentation field for the GetFilterString method of the ArtifactFilter class.

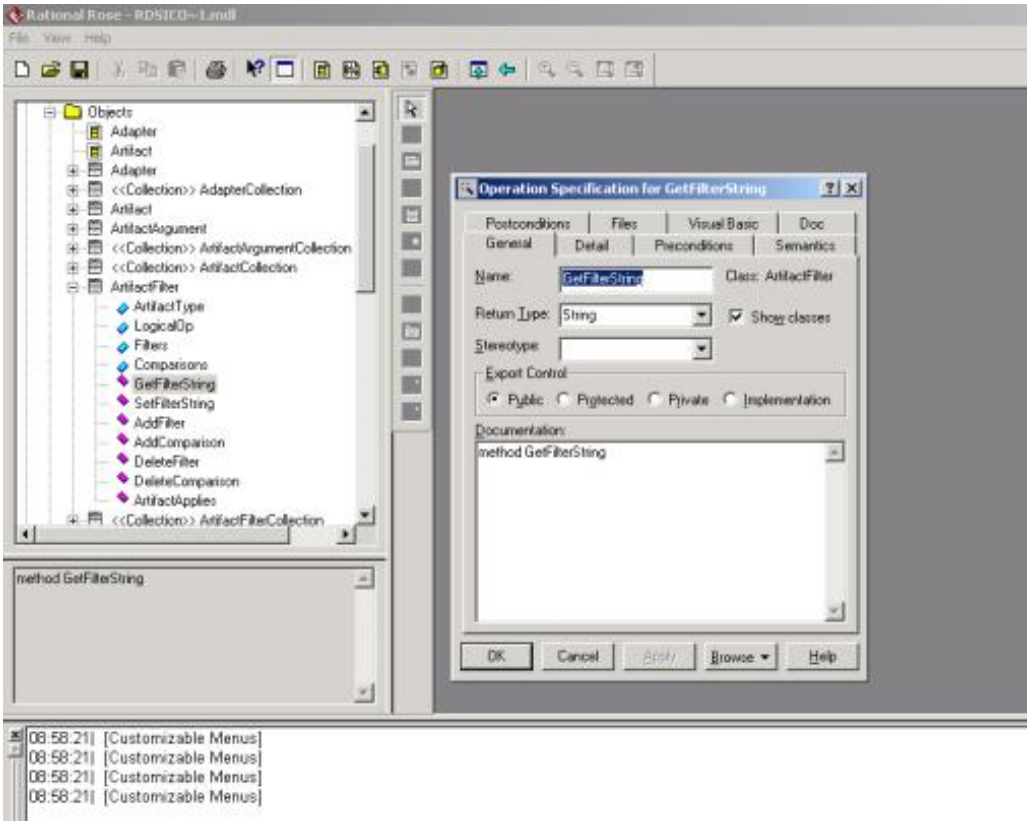


Figure 3: Documenting a Method

If you are maintaining both a Java API and a COM API, you may be able to map the common methods with one description source.

You can also create standardized class diagrams for an API by using simple drag and drop techniques provided by Rose. You can use a SoDA template to retrieve an Overview diagram for the API and class diagrams for classes in the API.

Note: The class diagram must have the same name as the class so that the template can successfully add this content to a generated report.

Updating a Rose Model

Perhaps the most important feature of the process we have been describing is that it enables a writer to work in tandem with ongoing development efforts. It lets you efficiently maintain documentation on an API by updating the documentation source (Rose) with updated code source files -- all without losing any existing documentation. You can also generate a log file that lists all changes to your model (and thus to the API). Currently, most API writers can discover these changes only through manual comparisons.

You can update your existing model by:

- Dragging the new component into your existing model.
- Rerunning the supplied Visual Basic application. (The Objects Package is modified to reflect any changes to the actual API --

including additions or deletions of, or changes to, Classes, Methods, Attributes, Arguments, and Return Types -- without overwriting any documentation fields, or class diagrams, that are still in place.)

- Using the generated log file to help you document the changes to the API in the Objects Package.

Generating a Report with SoDA

To generate a report for your deliverable, you can use a standard SoDA template that Rose provides for:

- Generating a COM API (TEST_API_Template.doc)
- Generating a Java API
- Generating a C++ API
- Generating separate htm files for each topic for a COM API
- Generating separate htm files for each topic for a Java API
- Generating separate htm files for each topic for a C++ API

You can also create a customized template.

First, run a SoDA template on the Objects Package to generate a Word document containing all the skeleton information for all the API reference topics. Then, to generate the SoDA report, do the following:

1. From Rose, click **Report** > **Soda Report**.
2. Select the appropriate SoDA template.
3. Click **File** > **Save As** and navigate to the location where you want to store the automatically generated folders and topic files.

Note that you should run this process from Rational Rose, not from the SoDA menu in Microsoft Word, which is slower.

- Copy the SoDA template into the Rational\SoDAWord\template\rose directory.
- Right-click on the file and select > **Properties**.
- Make sure that the file name appears as the Title on the Summary tab.

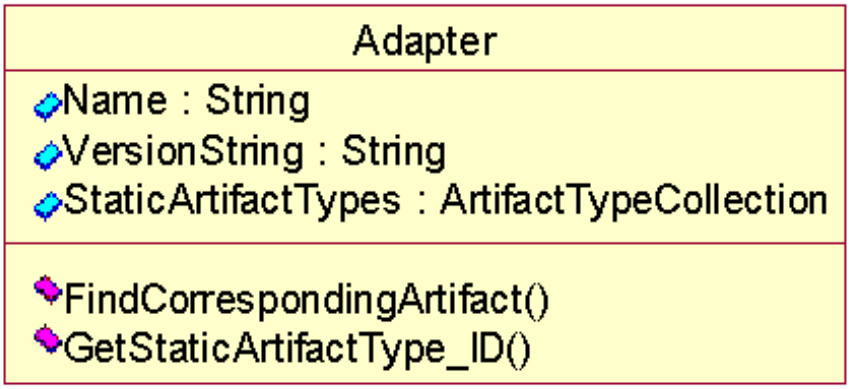
The following is a sample of what you can retrieve from Rose, using a SoDA template. In this example, all the descriptions were added in the documentation fields within the Objects Package, as was the diagram (the diagram name is "Adapter").

Adapter Class

Adapter Class

IAdapter Interface
SubClasses of Adapter

Adapter has no subclasses.



Properties Specific to Adapter

Property Name	Inherited From	Description
Name		property Name
VersionString		property VersionString
StaticArtifactTypes		property StaticArtifactTypes

Methods Specific to Adapter

Method Name (Return Type)	Arguments	Type
FindCorrespondingArtifact (Artifact)	pInternalObject	IUnknown
GetStaticArtifactType_ID (ArtifactType)	classID	Long

A Happy Ending

Now, let's return for a moment to that Technical Writer in our opening scenario. Today we find him happily working away at his computer and see no evidence of despair. Why? Because he has been making use of the process we just described, doing updates every few weeks as development of the API progresses. First, he retrieves the updated component from where it resides in Source Control, within the company's Change Management system (in this case Rational ClearCase®, of course). Then, he reverse engineers this component and prints the generated log file to see what changes have taken place. Using this list, he then browses through the Objects Package in his Rose model and makes the necessary documentation updates.

"Now I'm happy," he muses. "The developers are happier, too, because they have a reliable, up-to-date information source throughout the project -- and of course, customers will be thrilled to get complete documentation delivered right along with the product. I'll bet almost any API writer could make use of this process."

Appendices

Appendix A: More Details on Using Rational Rose and SoDA to Automate API Documentation

Appendix B: Sample Visual Basic Application and Rose Model (47K .zip file)
Don't have Rational Rose? View the Web version of the Rose Model

Appendix C: Sample SoDA Template and SoDA Report (52K .zip file)

Notes

* Application Programming Interface

1 For a full set of the samples used in this article, see the Appendices.

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

Appendix A: More Details on Using Rational Rose and SoDA to Automate API Documentation

This Appendix provides more detailed instructions for some of the steps described in the main article.

Before you begin the process, you will need the following:

- Installed and licensed copy of Rational Rose.
- Installed and licensed copy of Rational SoDA, corresponding to installed Rational Rose.
- The supplied Visual Basic docgen.exe.

Installing the SoDA Template

To generate the API documentation directly from Rose by running the SoDA template (TEST_API_Template.doc), install a custom SoDA template as follows:

1. Copy the template file to the SoDA Rose templates directory (for example, C:\Program Files\Rational\SoDAWord\template\rose).
2. Open the Template file.
3. If the SoDA menu does not appear, do the following:

   a) Reattach the soda.dot template. (Note: If you do not know how to attach a template, see the Microsoft Word online Help.)
   b) Click the **File** > **Properties** > **Summary** tab and set the title to the name of the Template file (for example, "TEST_API_Template").

Reverse Engineering a Component

The first step in automatically generating API reference documentation is to reverse engineer the DLL (or .jar) file containing the interfaces, objects, properties, and methods that you want to document. To reverse engineer a

DLL file, do the following:

1. Start Rose.
2. Create a new file (click **File** > **New**).
3. Drag the DLL from Windows Explorer and drop it onto the blank main class diagram (or onto the Logical View package). A dialog will appear, asking for Quick or Full Import.
4. Select **Full Import**. A COM-based model will be created from the DLL file.

Converting the Model with the Doc Gen Tool

After you've reverse engineered the source file into a Rose model, you can use the Doc Gen tool to create an Objects package.

The Documentation Generator (Doc Gen) tool automates the process for documenting the reverse engineered API. It creates the Objects package in your Rose model, and populates it with all of the relevant specifications for an API reference manual. This package is the place where you add documentation content.

Note: This step can be further refined by adding functionality that allows you to select the type of component you are cleaning up and what type of API you want to generate.

To convert a model from a COM basis to a Visual Basic basis, do the following:

1. Run docgen.exe. Then fill in the two required fields for your reverse engineered model.
2. In the **Rose model name** field, provide the full path to your model. (You must save your model file to a location on your system in order to have a file path to specify.) In the following example, the file name is RSE_DLL_Test.mdl.
3. In the **Root package name** field, enter the name of the package that contains the API information you want to document. Figure A-1 shows an example.

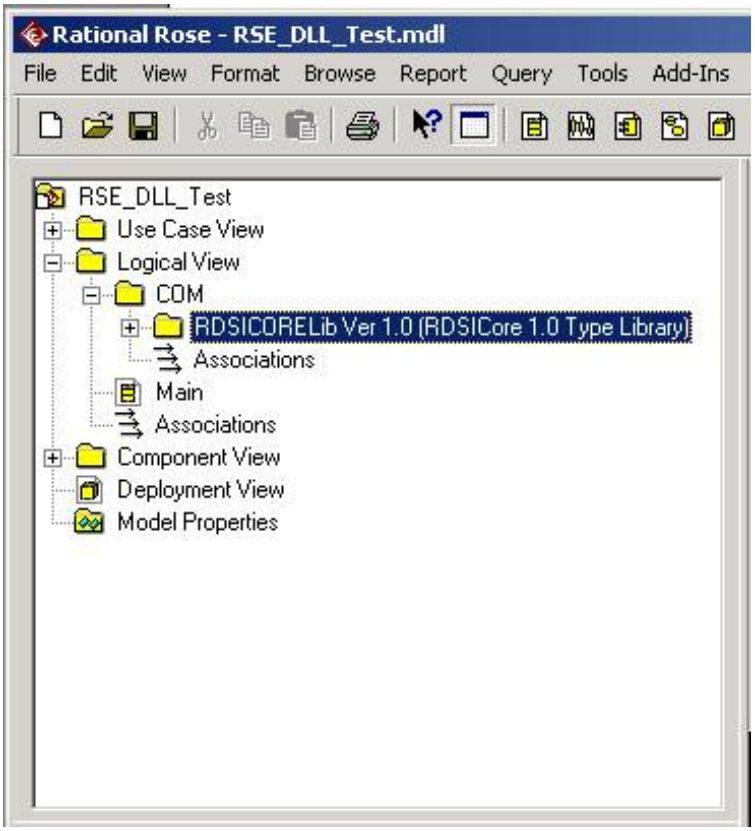


Figure A-1: Package Name with API Information

You are ready to run the Doc Gen tool once you have the entered the correct information into the two required fields, as shown in Figure A-2.

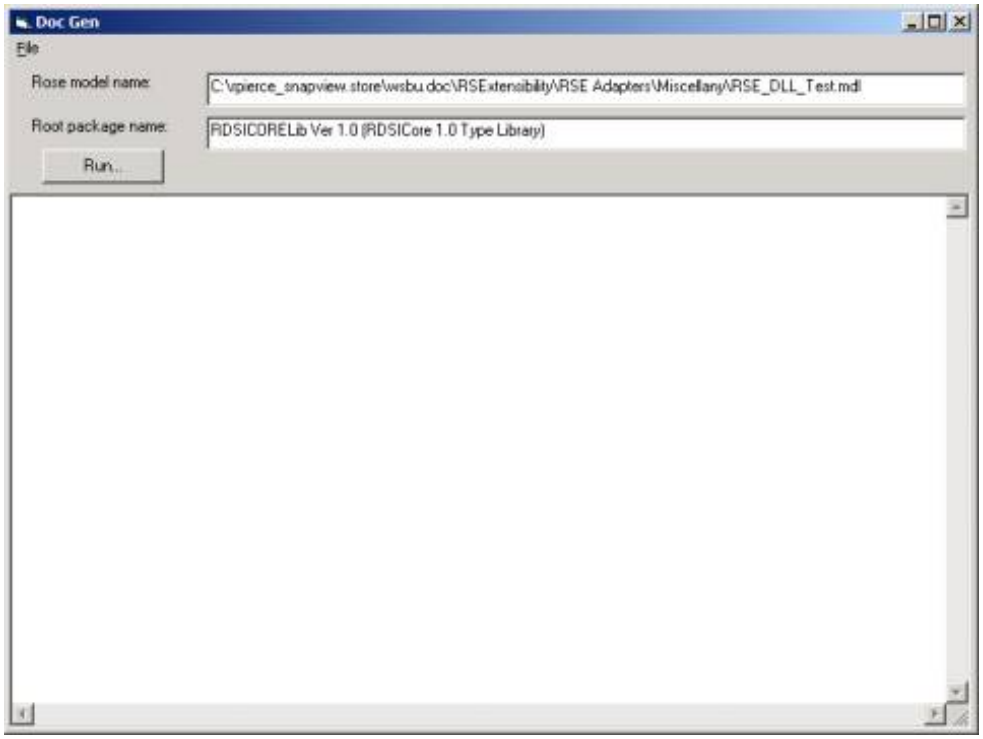


Figure A-2: Providing Model Name and Package Name

4. Click the **Run** button. The Objects package will be created. The Doc Gen tool then prints a list of all items that it is adding to the Objects package, as shown in Figure A-3.

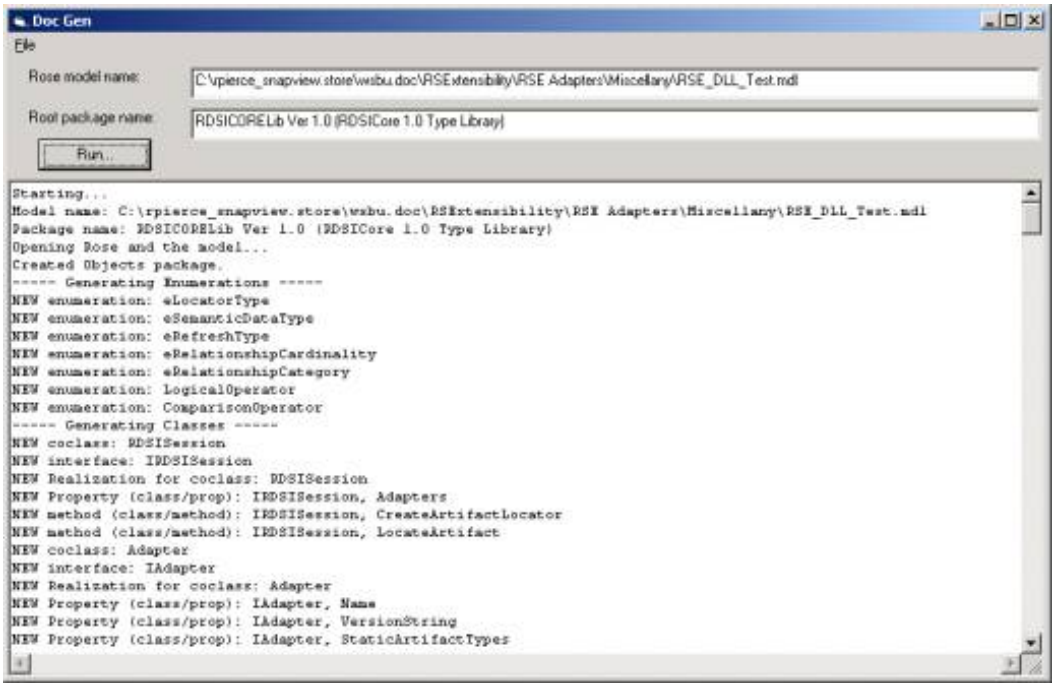


Figure A-3: Running the Doc Gen Tool

The Objects package is added to the Logical View that contains the converted model elements, as shown in Figure A-4.

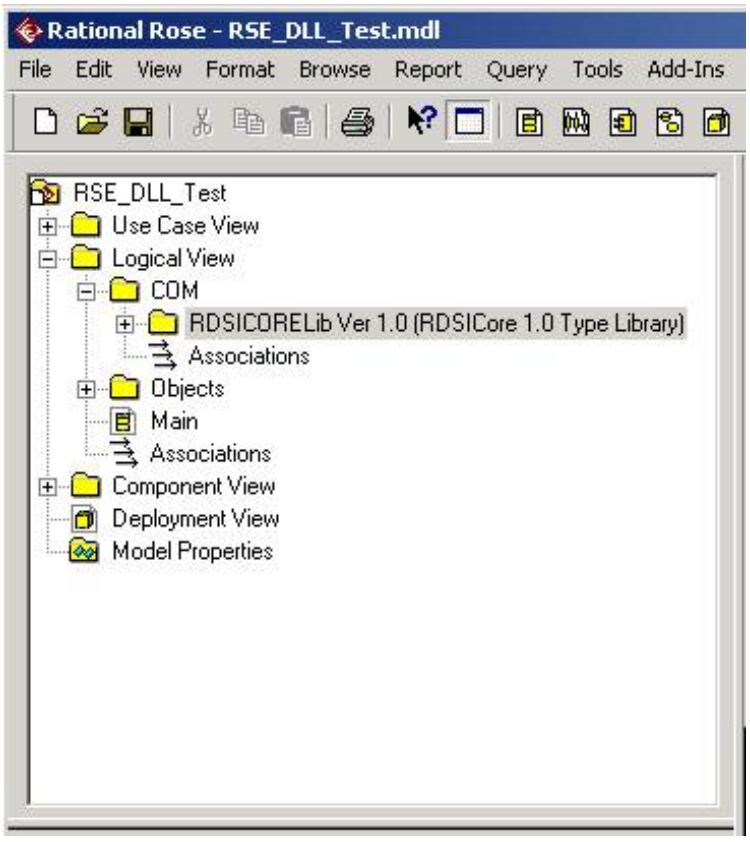


Figure A-4: Objects Package in Logical View

5. Save only the Objects package. Delete the other packages under the Logical View. When only the Objects package is left in the Logical View, you will work with it as your documentation source.

Note: Since different programming languages use different terminology and have different data types, the Doc Gen tool can be designed to handle these differences by generating the appropriate structure and data types for each language. For example, a DLL will have properties and methods, whereas a java source will have only methods. The data types for a given method will conform to the given source file type.

Using Rose to Add Documentation Content

Use Rational Rose to add descriptions for all objects in the API. For example, in Figure A-5, you can add descriptions to each of the coclasses in the Objects package.

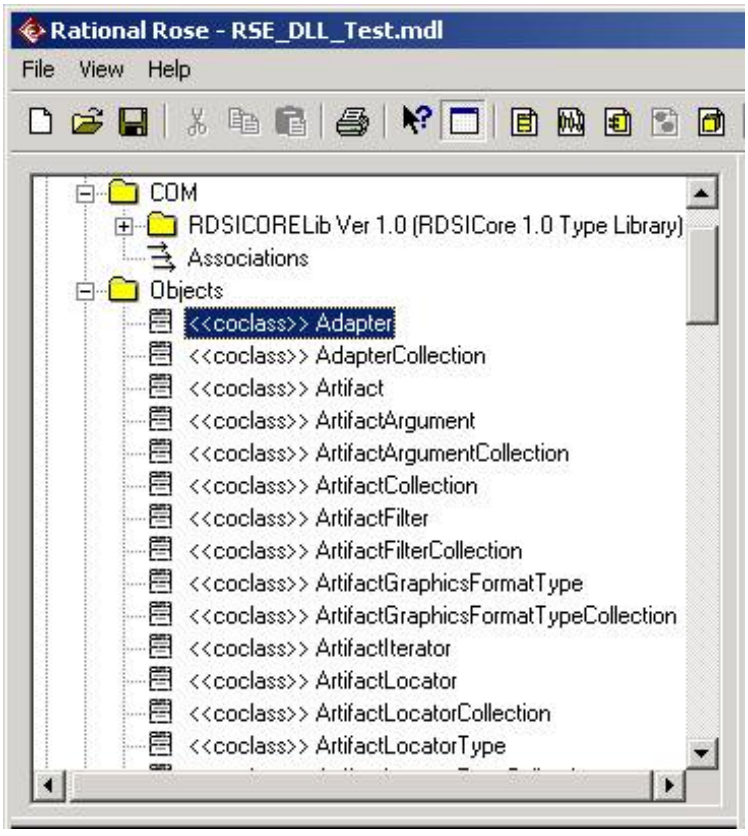


Figure A-5: Coclasses in the Objects Package

Using the documentation fields for each object specification, you can create single-source descriptions for the following features of an API:

- Classes or interfaces
- Attributes
- Operations

- Parameters of operations

Figure A-6 shows a documentation field for the GetFilterString method of the ArtifactFilter interface.

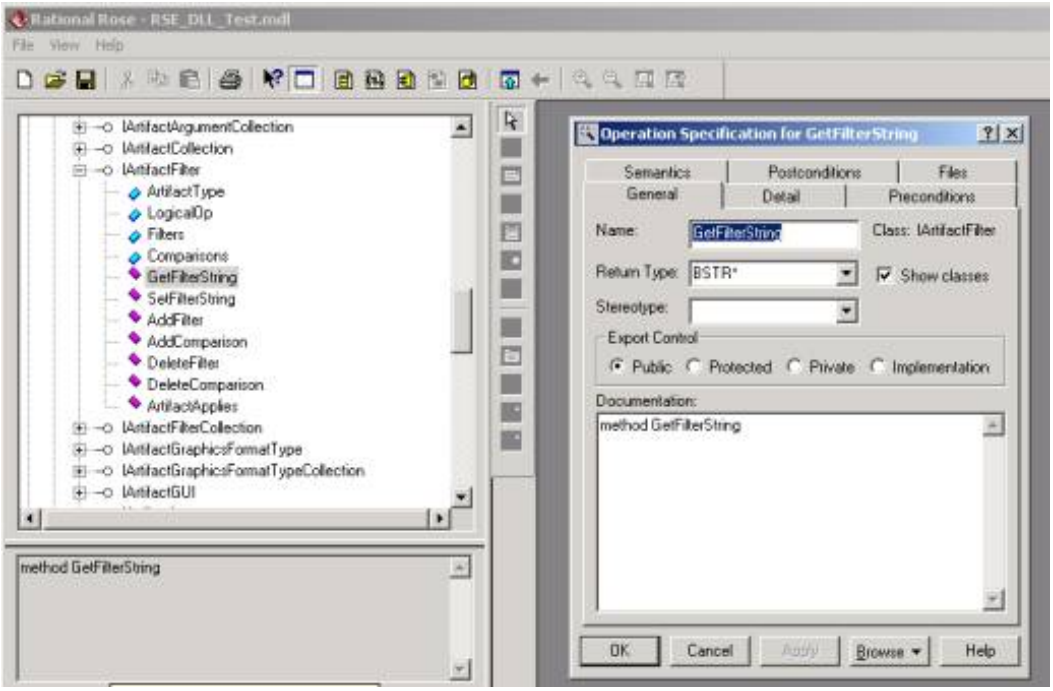


Figure A-6: Documentation Field for GetFilterString Method

If you are maintaining both a Java API and a COM API, then you may be able to map the common methods with one description source.

Creating Class Diagrams

You can create standardized class diagrams for an API, using simple drag and drop techniques provided by Rose. Use a SoDA template to retrieve an Overview diagram for the API and class diagrams for classes in the API.

Note: The class diagram must have the same name as the class so that the template can successfully add this content to a generated report.

Create an Overview Diagram

1. Right click on Objects package.
2. Select **New > Class Diagram**.
3. Name the diagram "Overview."
4. Double-click on the diagram to open it.
5. Click on classes and drag onto the diagram, as needed.

Create Class Diagrams

1. Create a new class diagram and name the diagram the same as a

Class.

2. Double-click on the diagram to open it.
3. Click on the Class with this same name.
4. Drag the Class onto the diagram.

You can also graphically display all attributes and operations for a class by selecting: **Format** > **Show All Attributes and Format** > **Show All Operations** as shown in Figure A-7.

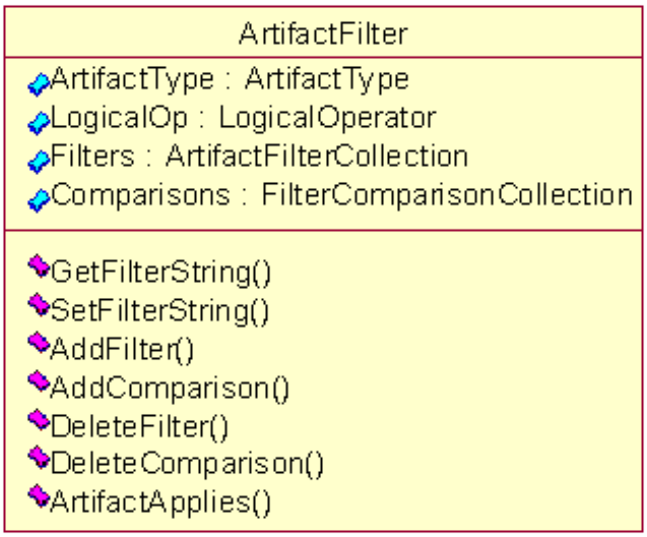


Figure A-7: Displaying Class Attributes and Operations

Running a New Iteration with an Updated Source

Follow the instructions in the main article under Updating a Rose Model.

Note that if some items have been added to your API:

- Reverse engineer the updated DLL.
- Run the Doc Gen tool. The Doc Gen tool adds the items that you deleted back into the Objects package. It also prints these changes providing you with a list of all changes to the API.
- Use the list to add documentation to the new or changed items in the API.

In Figure A-8, the Doc Gen tool lists items that were added to an API.

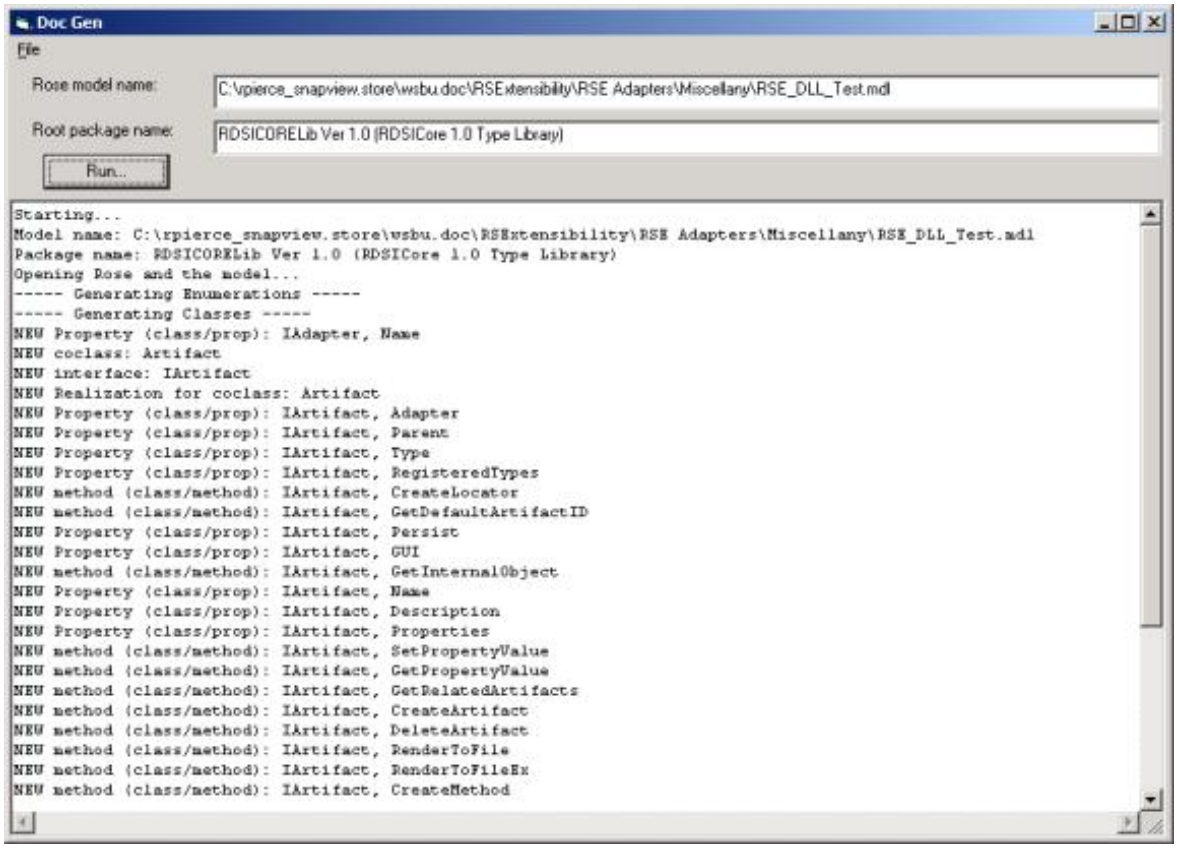


Figure A-8: Updating the Objects Package

You can test this process by deleting classes or methods in your Objects package and then reverse engineering the same DLL you started with (make sure you save your changes). When you run the Doc Gen tool, it adds all of the items back to the Objects package that you deleted.

Generating Reports with Customized SoDA Templates

When you follow the steps in the main article under Generating a Report with SoDA, keep in mind that you can also create customized templates for:

- Generating a Java-based API.
- Generating separate htm files for each topic for a COM-based API.
- Generating separate htm files for each topic for a Java-based API.

Detailed instructions are available in the *Using Rational SoDA for Word* manual; if you don't have a copy, you can order one online at: http://www.rational.com/support/documentation/index.jsp.

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

Thank you!

Automating Risk Management with Rational RequisitePro

by **Cindy Van Epps**
Software Engineer

Risk is like fire: if controlled it will help you; if uncontrolled it will rise up and destroy you.
-- Theodore Roosevelt

Software development is inherently a risky business. Software project assessment guru Capers Jones writes, "Software has long been regarded as one of the most risk-prone of all engineering activities."[1] Yet, based on my experience working with numerous software7 projects over twenty years, most projects do not follow through on effective risk management. Just creating a list of risks at the start of the project is not enough. In fact, if you do not do any more than listing the risks, then you have simply defined candidates to print on a "Top Ten Reasons My Project Failed" team T-shirt at the end of the effort. As elaborated in the April 2001 Software Development article "Keep Your Project on Track,"[2] risk management is a critical activity in creating project success. Failure to acknowledge this is the biggest risk to your project.

So what is the multi-tasking, fire-fighting, priority-juggling manager to do?

- First and foremost, risk management is everyone's business, not just the concern of a manager, project manager, or some tech lead. Therefore, management of any kind on a project, or associated with a project, must get everyone involved -- from the project manager to tech leads and developers.
- Second, anyone involved with risk management should have a repeatable process to follow to ensure clear, well-defined roles and responsibilities, and a quality execution of the risk management activities.
- Third, with all this involvement and all this process, automating risk

management can only help the onerous activity become more widely accepted, and the more user-friendly and helpful an activity is, the more likely it will be used.

The first two steps focus on people and process, and we can use the Rational Unified Process® or RUP®, product roles, disciplines (Project Management), and artifacts to accomplish them. So, let's look at some of the automation opportunities in the context of the RUP.[3]

The Risk Management Process

In our example, based on a large hardware manufacturing company, the RUP is in place as a corporate standard process. The proposed automation is for the *Identify and Assess Risks* activity of the RUP. Readers can add detail as their experience suggests -- adding more risk types or using different values for measuring their criticality, for example. Further detailing of the process can only accentuate the need for better automation.

The RUP defines the steps of the *Identify and Assess Risks* activity as follows:

- Identify potential risks
- Analyze and prioritize risks
- Identify risk avoidance strategies
- Identify risk mitigation strategies
- Identify risk contingency strategies
- Revisit risks during the iteration
- Revisit risks at the end of an iteration

The Activity Diagram in Figure 1 shows these steps in the context of the iterative development aspect of the RUP.

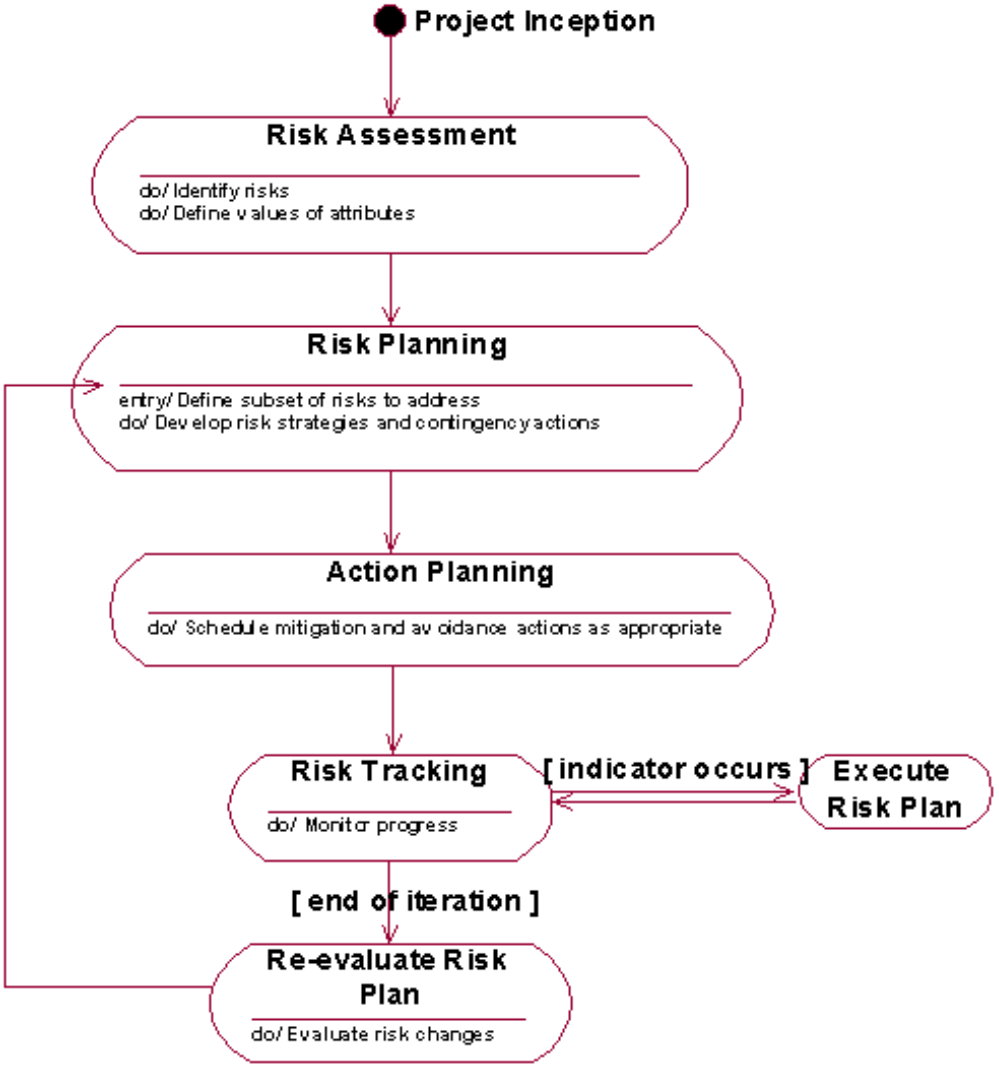


Figure 1: Risk Identification and Assessment Over an Iteration

Automation of the Process

Automated support for the risk management process can simplify the enactment of the process, help ensure accountability for following the process, and leverage the risk management activities beyond the scope of the project itself. Although some of these automation capabilities can be implemented in Microsoft Excel or Word, Rational RequisitePro® is a much more powerful tool that gives you a full range of risk management automation capabilities. In this section, we will look at how RequisitePro supports automated risk management, including the ability to:

- Manage risks and related information
- Trace avoidance, mitigation strategies, and contingency actions to risks
- Link to requirements
- Link to activities in project schedule

- Graph metrics to help manage risks
- Trace risks to multiple projects to leverage effort
- Keep a corporate or organization-wide risks/problems history

Management of Risks and Related Information

An effective way to simplify risk management is to facilitate the collection and viewing of risks and related information. This includes:

- Capturing the list of risks
- Assessing the values of risk attributes (impact, likelihood of occurrence)
- Related information (such as the *Risk Avoidance* or *Risk Mitigation* strategy, or the *Contingency Action* and *Owner)*. Automating sorting and searching via a spreadsheet, for instance, becomes valuable as the amount of risk information being managed increases.

As the state of the risk moves from identified to incorporated in project plans to mitigated, the ability to focus on a particular context is useful. For instance, you may want to see all of the high-impact risks that are not mitigated; you may plan activities in the next iteration of the project based on a ranking of the risks still not addressed.

In RequisitePro, you can establish software requirement types for *Feature*, *Use Case*, *Supplementary Requirements*, and more. For risk management, you establish requirement types that represent *Risk*, *Risk Avoidance*, *Risk Mitigation*, and *Contingency Action*. For each type of Risk information, you establish attributes that will help manage the risks. For *Risk* requirement types, we suggest the following minimum set of attributes (see Figure 2):

- **Approval:** values of Proposed, Approved, Scheduled, or Rejected
- **Impact of Risk:** High, Significant, Moderate, Minor, or Low
- **Likelihood of Occurrence:** High, Significant, Moderate, Minor, or Low
- **Overall Risk:** High, Significant, Moderate, Minor, or Low (derived as Impact of Risk multiplied by Likelihood of Occurrence)
- **Owner**
- **Cost:** May be dollars or other
- **Notes**

For *Risk Avoidance*, *Risk Mitigation*, and *Contingency Action* requirement types, we use the following minimum set of attributes:

- **Approval:** Values of Proposed, Approved, Scheduled, or Rejected
- **Owner**
- **Cost:** May be dollars or other
- **Notes**

The *Contingency Action* has an additional attribute called *Indicator* that describes the specific condition or event that determines that a *Risk* has become a reality.

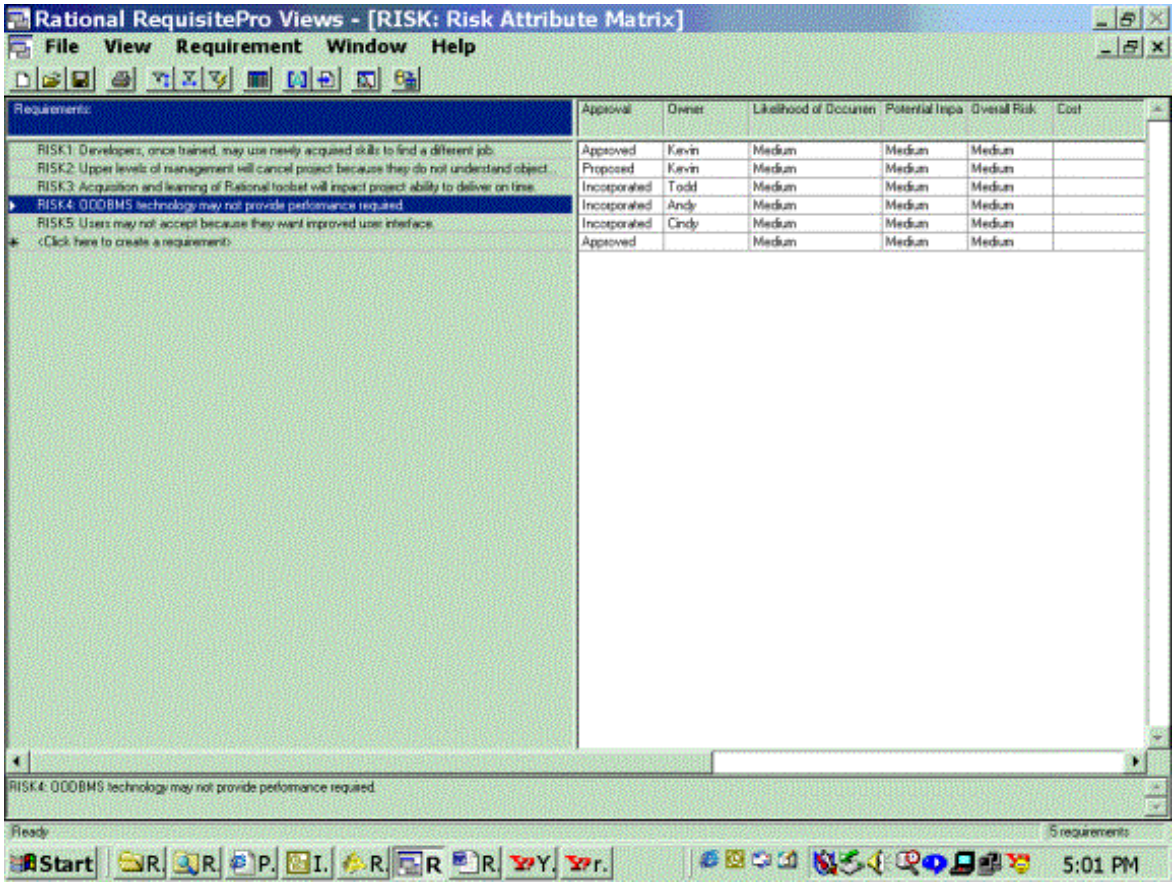


Figure 2: Rational RequisitePro Risk Attribute Matrix

Trace Avoidance, Mitigation Strategies, and Contingency Actions for Risks

The ability to trace from risks to the associated strategies and actions becomes a valuable way to help ensure accountability for following process. For each risk, this traceability makes it easy to determine how many strategies are being considered for reducing or eliminating a risk. Because contingency actions define the action to be taken if a risk becomes a reality, it is important to be prepared to react to high-impact risks. Integral to the planning and preparation are the identification and detailing of the contingency actions.

Even more effective is the ability to trace the same strategy or action to multiple risks. This enables the project manager to leverage effort and resources for greatest impact.

RequisitePro provides an easy way to trace risks to their associated strategies and actions. It is easy to add a risk avoidance, mitigation, or contingency associated with a risk and view all of the strategies and actions related to that risk (see Figure 3). It is just as easy to take the opposite approach and look at all of the risks traced to a particular avoidance strategy.

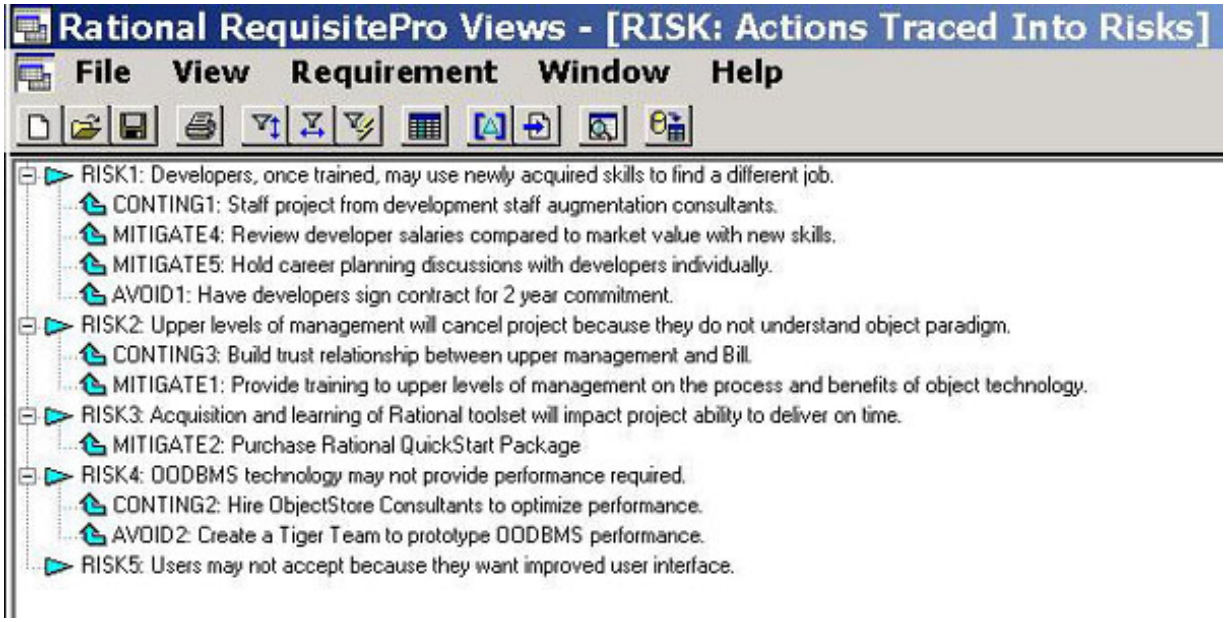


Figure 3: Actions Traced to Risks in Rational RequisitePro

Link to Requirements

So far, we have been discussing the use of RequisitePro, a requirements management tool, for risk management automation. Can we just extend our requirements management project to include these risk elements? That would be one way to include risk management in the automated software development process; we could easily trace risks to requirements in the same way that we trace risks to risk strategies and actions (see Figure 4). But consider that some risks may be so sensitive that we do not want them visible to all team members. An attrition risk and the proposed contingency action, for instance, would be sensitive information.

Therefore, *we recommend keeping the risks in a project that is separate from the project requirements*. The cross-project tracing capability in RequisitePro allows tracing from the risks in the risk project to the requirements in another RequisitePro project.

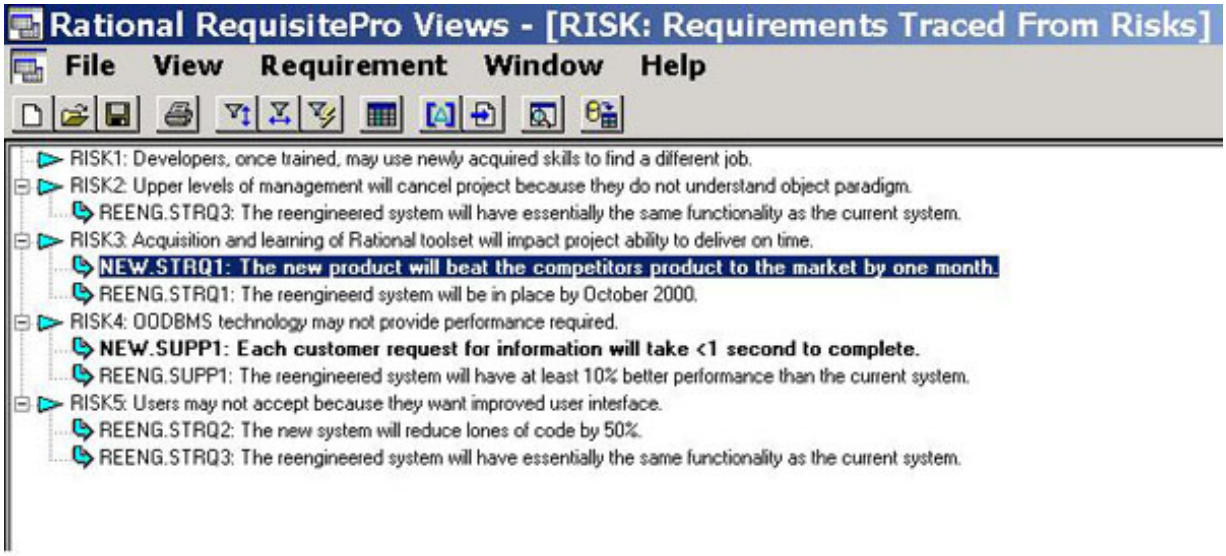


Figure 4. Requirements Traced from Risks in Rational RequisitePro

Link to Activities in Project Schedule

Clearly defining and executing the risk management activities helps ensure accountability in the risk management process. This means *we do more than identify risk lists and strategies; we do something about them*. We schedule the risk management activities in our iteration plans, as indicated in the RUP. This may involve nothing more than revisiting the acceptance strategy at the end of an iteration, or it may require actually setting aside time and effort to work on risk mitigation or avoidance.

A clear path from the risk management activities to the project schedule can be achieved through the RequisitePro integration with Microsoft Project™. With this integration, selected risk avoidance and mitigation strategies as well as contingency actions can be inserted into a project schedule. The synchronization between RequisitePro and Microsoft Project allows the project manager to have a clear understanding of the risk management activities as they progress through each iteration. The integration also gives project team members a clear understanding of their risk management responsibilities.

Graph Metrics to Help Manage Risks

Two kinds of metrics help simplify enactment of the process:

- *Snapshot metrics* give a perspective on the risks facing the project at a particular point in time.
- *Trend metrics* show how the project is proceeding with respect to risks over time.

The project's condition with respect to risk is a critical element in the RUP. One criterion for transition from the Elaboration Phase to the Construction Phase is that the risks are sufficiently mitigated to be able to predictably determine the cost and schedule for completion of development. This type of evaluation requires some quantifiable analysis such as:

- Which risks have been addressed?
- The severity of the remaining risks.
- The amount of change in risks over the life of the project.

RequisitePro contains a metrics generation capability that allows a project manager to graphically depict these kinds of metrics. Figure 5 is an example of a trend metric.

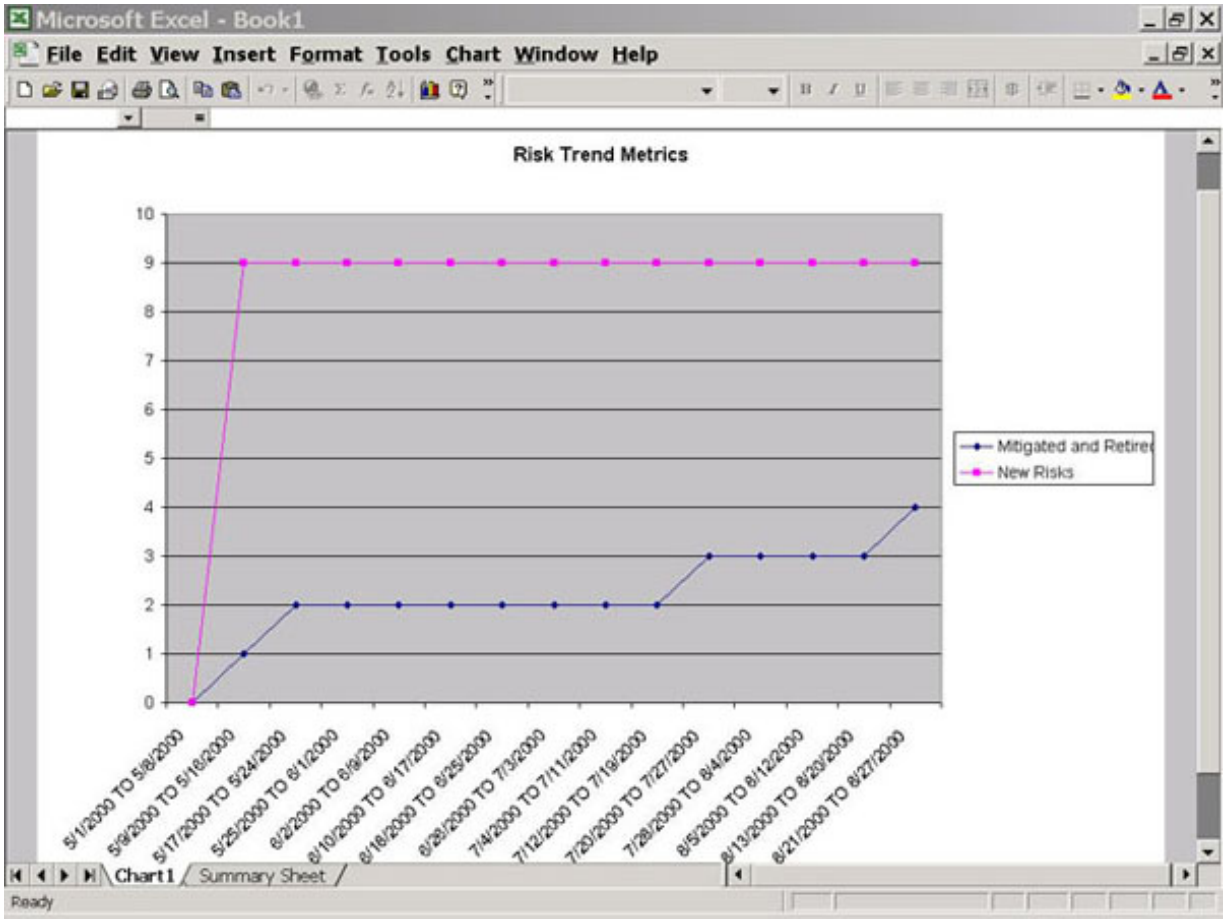


Figure 5: Trend Metric Generated by Rational RequisitePro

Trace Risks to Multiple Projects to Leverage Effort

The same approach we use to trace risk to requirements can be used to leverage the risk management activities beyond the scope of the project itself. This can be done across multiple projects using RequisitePro. Risks, avoidance and mitigation strategies, as well as contingency actions can be traced from one central risk management project to requirements management projects for various software development projects within the same organization.

Because projects in an organization very often use similar technologies when working in a given domain, the risks are often the same or closely related. The activity of addressing risks is quite often elevated to a higher level than the project itself; therefore, the resources to address risks, either authoritatively or financially, come from outside the project. Cross-project risk management allows an organization to prioritize and address risks across the organization, making more efficient use of risk management resources.

Figure 4 (above) shows two projects: A new development project whose requirements are tagged "NEW," and a re-engineering project whose requirements are tagged "REENG." We see that both projects have a performance risk related to the chosen database technology. The risk mitigation, then, is to create a task-specific team consisting of members from both projects who will test and suggest optimization of the database technology to meet the stringent requirement.

Keep a Corporate or Organization-Wide Risks History

Organizations striving to reach a higher level of process maturity seek to learn from the failures and "speed bumps" they encounter. This helps managers of new projects avoid the pitfalls of previous ones. Although books, consultants, and techniques can help us avoid problems that plague software development in general, the real risks to a project stem from the unique combination of management, technology, politics, and skills present in the particular project domain.

The approach to managing risks at the organizational level that we discussed above in the **Trace Risks to Multiple Projects** section, then, becomes a powerful tool for collecting historical information and ideas for risk management on new projects. For example, a post-mortem analysis of the risks on a project that was cancelled could reveal which risks contributed to the failure, and which risk avoidance or contingency actions used for the project were ineffective. Conversely, analyzing a project that was fraught with risk, but that was very successful, could provide valuable information about which risk management strategies DO work. The team could then document this assessment information in the notes attributes of the risks and actions.

Conclusion

We discussed several ways that automation can help bring risk management into the mainstream project management process, including support for managing the risks and associated information, linking to project schedules, and managing risks across several projects.

Rational RequisitePro is a powerful tool for automating this process, by using a configuration that includes risk requirement type and attribute definition, traceability, and metrics.

For a template RequisitePro project for risk management, click here.

Notes

1. Capers Jones, *Assessment and Control of Software Risks*. Yourdon Press, 1994.

2. Neil Potter and MarySakry, "Keep Your Project on Track." *Software Development* Magazine, April 2001.

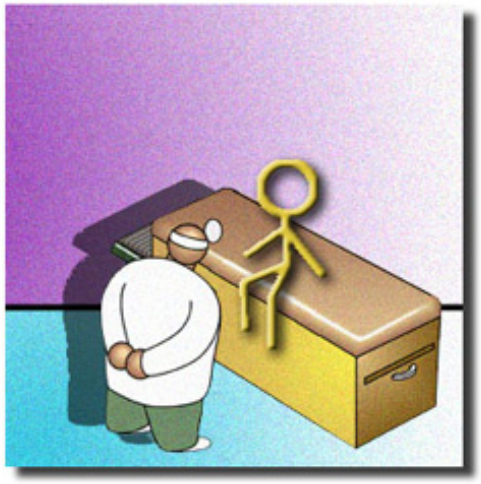
3. Note that the process, risks, and examples used in this article are excerpted from real projects.

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

Dear Dr. Use Case:
What About "Shall" Wording in a Use Case?

by **Leslee Probasco**
Rational Software Canada

Dear Dr. Use Case,



I am a principal software engineer working on specifying requirements for a new business software project. The main product of my current efforts will be a project Software Requirements Specification (SRS). My question concerns the use of "shall" language in SRS use cases.

I recently came across your whitepaper, "Combining Software Requirements Specifications with Use-Case Modeling," co-authored with Dean Leffingwell. In this paper and the accompanying SRS template, you recommend including the use cases directly in the SRS (in section 3.1, Use-Case Reports).

Traditional *SRS requirements* are stated using "*shall*" language (for example, "The ATM *shall* prompt the user for a PIN"); however, *use-case flows* describe user/system scenarios without "*shall*" (for example, "A user inserts her card in an ATM. The ATM prompts the user for a PIN. The user enters her PIN, then selects 'Withdrawal.' The ATM prompts the user for a withdrawal amount.").

My question is this: *In a use-case-based SRS, is it appropriate to use the non-"*shall*" wording of a scenario directly, or is it necessary to reword the scenario flows so that they look and sound like traditional requirements?*

Any reply will be greatly appreciated. Thanks.

Signed,
What "Shall" I Do?

Dear What "Shall,"

Many projects have a convention -- or perhaps even a strict project *requirement* -- of indicating each stated SRS *requirement* with the word "*shall*." Using "*shall*" has the distinct advantage of making very clear which statements are actual "requirements," and which statements are merely suggestions of what the system might do. This provides not only clarity about what the actual requirements are, but also flexibility for the designer in the other areas.

What About "Shall" Statements in the Use-Case Flow-of-Events?

When project convention dictates the use of "*shall*" for requirements, there is often a misconception that the detailed statements in the use-case descriptions must also follow that convention. Generally, though, "*shall*" wording in a use-case flow-of-events sounds stilted and unnatural, so I would *not* recommend it.

If the customer wants (or requires) "*shall*" wording in the details of the use case, there is no hard restriction stating that they *cannot* use it -- the wording of the use-case description is very flexible. However, we recommend using *easily understood* language -- language that the customer would use in everyday speech.

If my customer insisted on using "*shall*" wording in the use-case descriptions, I would *very strongly* recommend that they *not* do so, pointing out at least the following two reasons why:

1. *Use cases become less comprehensible*. One of the use case's main objectives is to present a description of the functionality of the system in *easily understood* language, and the use cases are to be read and used by ***all*** team members. But many people find "*shall*" statements make the use-case narrative harder to read and understand.
2. *There is increased requirements traceability maintenance*. On a few past projects, we tried considering each line of a use case as an individual, traceable requirement. This made them time-consuming to maintain, and the individual requirements were not always stand-alone or testable -- two more criteria for a good SRS requirement.

What to Do When a Project Requires "Shall" Statements

Now as to your specific question: If a project has a requirement to use "*shall*" statements, is it *necessary* to reword the use-case scenario flows to have "*shall*" statements in them?

The bottom line answer is ***No***, it is ***not*** necessary. You have a few options here:

Use the "shall" at the feature level. I have worked on projects that had this requirement, and we satisfied it by using "*shall*" at the *feature* level -- then tracing these *features* to the *use cases*. This allows the *use cases* to do what they do best: to put the requirements (in this case, the *features*) into the context of a *user's goal*, written in an easily understood style, describing a complete flow through the system.

Use the "shall" in the use-case short description. If the "*shall*" statement is a requirement within the SRS, it could also be used in the use-case short description (which would clearly state the requirements covered in this use case and be available for traceability, and so on), leaving the flow-of-events in a freely flowing, natural-language format.

Use one "shall" statement for each use-case flow. Another option to consider is having one "*shall*" statement per flow (one for the basic flow and one for each alternate flow). For example:

- *The system shall allow a user to Register For Courses.*
- *The system shall allow notification of a user that the Registration period is over when Registering For Courses.*

At this point, each "*shall" will* be testable -- and the statement is more informative, too. But it involves quite a bit of work to extract these "*shall*" statements from the use cases and maintain them, so I would only do this if the "*shall*" statements are requirements. In such instances, you would have your "*shall*"s and still be able to read your use case.

I would suggest you try very hard to convince your customer to satisfy their requirement for "*shall*" statements in one of these three ways. However, if (after giving it your best shot to help them see the light) you *cannot* convince them to take this approach -- since there are no hard requirements for writing use-case flows, and not even recommendations in the Unified Modeling Language for how to write them -- you can certainly finally agree to reword the use-case scenario flows with the "*shall*" statements, if describing use cases in this way still makes sense and helps clarify the requirements of the system.

Just make sure your customer has thoughtfully considered whether using the "shall" statements in the use-case flows really makes the requirements *more understandable* for the whole team. Ultimately, clear requirements will increase your ability to build a *high-quality* product *on time* and *within budget*, and to successfully meet your customer's *real needs*.

The Bottom Line: Remember the Reason for Having Use Cases!

The bottom line is this: *Whatever the customer wants in the use-case description is what should be there*. But remember that the *main objective* of the use-case model is to *COMMUNICATE* to all parties involved what the system should do. This is adjusted on a case-by-case basis, based on the needs, requirements, and language of the customers.

Hope this helps. Let me know how it goes.

Usefully yours,

Dr. Use Case

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

How to Learn a New Programming Language

by **Joe Marasco**
Senior Vice President
Rational Software

Every few years a new programming language arrives on the scene, promising to be the answer to a maiden's prayer and superior to all languages that have gone before it. But how can we discover the true value of a new language without sinking too much valuable time into learning it? This article talks about a cost-effective way to quickly learn the new language in order to gauge its usefulness.

The Problem

The first reaction to a new language, for those of us who have been around the barn a few times, is "What, again?" We're jaded enough to believe that all these languages are more alike than they are different, and we are quick to dispense with the hype surrounding each new introduction.

On the other hand, hope springs eternal; maybe this time there is more gold than dross. Maybe someone has invented a better "do" loop. Whatever. The point is that simply dismissing the new candidate out of hand is not an option for those of us seeking competitive strength wherever we can find it. So we grit our teeth and once more leap into the breach.

Reading books on new programming languages is rarely an uplifting experience. Frankly, most of them are awful. Every now and then there is a gem that not only kick-starts a new generation of programmers but also stands the test of time as well; for example, "*K&R*"[1] has been the classic manual on C programming since the language first became popular. It introduces C and supplies everything you need in a reference manual, all in a short, readable, and (once) inexpensive paperback. (Today that slim

paperback costs $40.) But as I said, this kind of brief, well-organized, and highly informative book is a rarity.

So, after mucking around in a few of the early books available on a new language, most professionals start to learn a new programming language by writing their first program. This enables more learning than simply reading the somewhat contrived and artificial examples that most books set forth to introduce you to the language.

The Problem with the Problem

Once you decide to write a program, you must then get calibrated. Ever since "*K&R*," it has been almost a cliché to write your first program to do nothing more than print or display the message "Hello, world." While this does teach you how to print out a string and use the compiler and linker/loader, it doesn't do much else. You don't get enough return on your investment to consider this even a first effort.

I believe that the first program shouldn't be so trivial. Nothing ventured, nothing gained. On the other hand, we don't want to make this into a huge effort. We want a project that is just hard enough to cause us to learn how to implement things in the idiom of the new language, yet we don't want to get confused by having to do new domain learning.

What this means is that we would like to have a "standard problem," so to speak, so that each time we have to re-implement its solution in a new programming language under evaluation, we have a "calibration." In other words, we don't want to have to invent new science or devise new algorithms. Theoretically, the exercise should get easier each time we do a new implementation in a new language, because the "problem space" is already familiar to us, and we can spend more of our time judging how well the new language articulates the solution. If the new language takes us four times as long as usual to solve the "standard problem," then we might begin to ask questions about the new language and/or its learning curve characteristics.[2]

What Should the Standard Problem Contain?

Glad you asked. Here are a few of the things I like to discover in any new language:

1. How to print out a string. This is useful in terms of prompting the user for input, for example. As mentioned above, this is as far as you get with the "Hello, world" problem.
2. How to accept input from the user. You can start with simple strings, working your way up to formatted numbers. You can do an awful lot by just reading character strings, so that's a good place to start.
3. Simple algorithmic stuff. You should manipulate some data. Nothing fancy here, maybe just some simple assignment, arithmetic operations, and so on. This will expose whether you need to call in math libraries or not, and so on. It's not necessary to test your

ship's rigging under storm conditions at sea, but you do have to get the boat out of the slip.

4. How to store data persistently. This is a big step up, because it asks that you write out a result and store it somewhere that survives the execution of the program. Ideally, you would like your sample problem to both read and write to the permanent store. Generally speaking, this exposes the language's interface to some sort of file system. Note that the file need be nothing more elaborate than a text file.
5. How to implement a "typical" data structure like a linked list. Since these beasts come up over and over again in programming chores, it is good to have one in your sample problem so you can see how this trick works in the new language.
6. How to do simple error handling. What do we do when user input or a data file is not what we expect -- when it's blank, corrupt, or just plain silly?
7. How to evaluate the abstraction and encapsulation capabilities. Will it be easy or hard to react to a change in requirements or problem definition?

Note that number five is really just an extension of number three.

I offer one big caveat here. What we are exploring are "programming in the small" features of the language. While these are important, they do not test the other very important "programming in the large" features -- things like the ability to have public and private interfaces, interactions with other programmatic infrastructures, and, of course, graphics. Nonetheless, those things can be more easily investigated once the programming in the small issues are better understood.

The Animal Game

Here is the program I have been re-coding as my own standard problem since the 1960s.[3] It is called "The Animal Game."

The program is an interactive dialog between a user and the program. The user is prompted to "Think of an animal." The program then begins by asking, "Is your animal a beagle?"[4] If the user was thinking of a beagle, he answers, "Yes"; the program congratulates itself on its perspicacity, thanks the user for playing, and the game is over.

If, on the other hand, the user is not thinking of a beagle, he answers, "No." Downcast, the program responds, "Sorry, I did not guess your animal. Tell me your animal and give me a question that has the answer 'Yes' for your animal, and 'No' for a beagle."

For example, if the person is thinking of a trout, he would enter "trout," followed by the question, "Is it a fish?" The answer is yes for a trout, and no for a beagle.

Having stumped the program and entered his animal and his question, the user is thanked, and the program again terminates.

However, the next time one plays, something different happens. After being prompted to "Think of an animal," the first question asked is, "Is it a fish?" If the user answers "Yes," then the program asks, "Is it a trout?" But if the user answers "No" to "Is it a fish?" then the program asks, "Is it a beagle?" If the user was thinking either of a beagle or a trout, then the program wins by guessing right. On the other hand, if the person was thinking of some other animal, then once again the program admits defeat, and asks for the new animal and a question that will distinguish the new animal from either a beagle or a trout, as appropriate.

So, in the beginning, the program is exceedingly "dumb." But by storing up the new animals and the new questions, it gets "smarter" as it plays. It won't always guess your animal by the shortest possible route, but after a while it can fake intelligence and "guess" your animal almost every time. That's because as its database gets bigger, it appears to "track down your animal" more and more surely.[5]

Does the Animal Game Fit the Criteria?

You betcha. Here they are again:

1. **Output strings**.The program needs to give the user instructions, and to respond to his responses to your questions.
2. **Input**.The program needs to take strings from the user and do something with them. The dialog nature of the problem requires some (but not much) parsing of the input.
3. **Simple algorithm**.The program is going to branch to different questions depending on the answers. Thus, depending on the "Yes" and "No" responses, it will traverse a data structure.
4. **Interaction with a permanent store**.The program needs to keep the current animal and question database file somewhere, and read it in on program invocation. This data will be used to exhaust the question set. If the program does not guess the animal, it will have to update the file with the new animal and the new question. Then it will have to store these for the next user session.
5. **Prototypical data structure.** Once again, the program will traverse some sort of linked list to achieve the result. When the user adds a new animal and a new question, the program will have to add those links and update some of the old links to point to the new information.
6. **Error handling**.The program needs to be able to deal with blank input from the user when actual values are required, and it has to cope with the data file being corrupted. This area has lots of latitude, depending on how user-friendly you want the program to be. For example, what do you do if the user quits halfway through?
7. **Abstraction and encapsulation**.You can easily generalize the problem to be a vegetable game, a mineral game, or a famous

person game. These variations should depend only on loading a different data file; the language should let you do all of them with the same program.

Note that we have kept this very simple. For example, we don't ask you to allow several people to play the game simultaneously; this would vastly complicate the problem. But even for serial interactions by different users, it makes for a nice programming problem.

Does It Pass the "So What?" Test?

Over the years, I have implemented "The Animal Game" in the following languages:[6]

- FORTRAN
- BASIC
- APL
- Pascal
- FORTH[7]
- C
- Ada
- C++

I would have done Java, but[8]...

In general, it takes me a few hours to recall how to make the linked list work and get something on the air. To allow someone else to play the game unsupervised (which means doing some error handling) usually takes me a few days of programming. I could probably go faster if I could ever find legacy code for any of the previous implementations, but I never can. The exercise gets repeated with a periodicity of every four to five years, which is just long enough to lose the last example.

But, you say, why should you need legacy code? Don't you have a design spec or a pseudo-code document lying around that would enable you to do the implementation without referring to the last piece of code you wrote?

The answer is twofold: Yes, I do develop the pseudo-code, but I get to do it over again each time because I can never find that, either. Second, seeing how you did something in the last language is a useful crutch when trying to do the same thing in the new language. If I had my druthers, I'd be able to locate both my last pseudo-code version and my last language implementation. Together, they would give me a quicker start. If only I kept better records. Oh well. Although this would be a grievous infraction for a software development organization, it is somewhat less of a sin for what is, after all, a small, personal, programming effort.

By far the most bizarre implementations I've done were in APL and

FORTH. Practitioners of those languages will probably understand why, and it is futile to try to explain this to non-practitioners.

Absent from the list of likely suspects are LISP, Smalltalk, PL/I, and COBOL. There is nothing to prevent you from trying these; I just never have.

Incidentally, one of the other things you learn each time you do this exercise is the nature of the development environment and the available tools. I learned, for example, that the early C++ debuggers were not that wonderful; on the other hand, the utility of the Rational Environment was apparent, and my Ada implementation was done in record time. When I did it in FORTH, I learned how to reboot my machine every time there was a runtime error.

It's Your Game

The best way to get one's feet wet with a new programming language is to program a "standard problem" that you are already familiar with. This allows you to explore the language and learn "How do I do this?" on a known set of useful questions. My standard problem is one that I have used for many years, one that I believe forces you to confront a minimal but useful set of basic issues.

If you have developed your own standard problem(s), we would love to hear from you.

Notes

1 Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, 2nd edition. Prentice Hall, 1988.
2 There is, of course, the possibility that we are losing prowess with the passage of time. Only the true pessimists among us would admit this possibility.
3 I must admit to being proud of having delivered production-quality software in five different decades, starting with the sixties. In the early days, I wrote much of it myself; later on, my colleagues did everything they could to prevent me from writing code. Over that span of time, I have had to deal with many different languages.
4 An homage to Snoopy and his creator, Charles Schultz.
5 There is a flaw here. A user can insert a bad question or otherwise get things bollixed up -- like reversing the roles of "yes" and "no." Don't laugh; I have seen it done. I have even seen a player forget the animal he was supposed to be thinking of in mid-game. Once this happens, the database is contaminated for future use. Historically, we have had more success with third graders than with adults; apparently, the simplicity of the game is perfect for eight-year-olds and daunting for their parents.
6 The list reveals my programming roots: I come from the scientific side of the house, not the business side. Although I did have to manage a group of COBOL programmers once, I was so busy at the time that learning their lingo was way down on the priority list. And, to tell the absolute truth, I still suffered from "language snobbery" at that point in my career.
7 FORTH is the exception to the rule that programming languages are all caps only if they are acronyms. Charles Moore, the inventor of FORTH, wanted it to be a fourth generation language, but the computer he was working on only allowed five letter identifiers, or so goes the folklore. The individual letters of FORTH do not stand for anything themselves.
8 I really am getting too old for this. And to be perfectly evenhanded about it, I will decline the opportunity in C# as well.

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

Reader Mail

Got questions or comments about something you've read in* The Rational Edge? *Send them to mperrow@rational.com, and we will try to get you an answer ASAP! All questions and answers that could be useful to other readers will be printed in this section.

Dear Rational Edge:

I need to model the design of a character-based, menu-driven user interface. I would like to model the tree structure and see the various links as the tree is traversed. Any ideas? This is a very large menu system which is very deep.

We forwarded this question to Terry Quatrani and Bob Maksimchuk, two of Rational's Rose and UML gurus.

From Terry Quatrani:
I would recommend using an activity diagram, since this diagram shows flow from one activity to another. You can also show decision points and synchronization points. Bob, do you agree?

From Bob Maksimchuk:
Well, it depends. An activity diagram offers a good solution if the user wants to capture the "business" flow and decisions made. If he wants to capture the structure of message traffic among the menus, it may be best to use sequence diagrams with each object representing a menu (use a lot of hyperlinks also). It depends on what key elements of the system must be focused on. If it's really only the tree structure and the links, heck, you could use class diagrams.

Applying the WAVE test to your use cases

Dear Rational Edge:

I have a question concerning Rose (UML). I'm showing the realization of the use case called "main" with a sequence diagram. The "main" includes another use case called "login" (for example); how do I present this in the sequence diagram?

Ethereally grateful (for the answer or the reference to somebody familiar

with it).

Alex Djekic
Ericsson, Canada

Bob Maksimchuk replies:

There are different ways this could be shown. First the fact that Main includes Login (and by the way, I hope those are not your *real* use case names, but that is another topic) should be shown on a use case diagram with an <<includes>> relationship between the two use cases. However, I assume the core of your question regards how best to show the processing of the included use case in the sequence diagram of the including use case. Its really quite simple once you see it.

Build two separate sequence diagrams -- one for Main and one for Login. Now on the Main sequence diagram (assuming you are using Rose), attach a note at the point in the sequence where the "included" processing would begin. And it's probably best to attach to a lifeline. Then drag the sequence diagram for the included use case (Login) from the browser onto the note and drop it. This creates a hyperlink between the two sequence diagrams.

So when you read the Main sequence diagram and come to the link, double click it and you jump to the sequence diagram of Login. When done with Login sequence diagram just click the back arrow and you return to the Main sequence diagram . Keeps everything nice and clean. Also makes changes easier to incorporate.

Hope this helps,
Bob Maksimchuk
Data Modeling Evangelist

Thanks for the reply!

Your suggestion works very well. I suppose we would be able to use the same approach for <<extend>>. As for those use case names, how about "Present Main View" and "Authenticate User"?

You are welcome Alex...

Yes the same technique would work for <<extends>>. But in that case you might want to put text on the connecting line to the hyperlink that states the condition on which the <<extends>> user case will be executed.

Regarding your use case names, they are better. I do have a concern though. There are a few rules of thumb to determine if your potential use cases are properly "leveled" -- that is they are not to big, not to small, not functionally decomposed, etc.

I have dubbed these rules my "WAVE" test. If you fail anyone of these, just wave that use case goodbye. The acronym means:

What - Does the use case state "What" is to be done, not how? Yours seem OK here. Fail this test and you are probably way down into implementation.

Actor Viewpoint - Are the use cases coming from the Actor's point of view? Your name "Authenticate User" might slide by because (I assume) it is the included use case. But "Present Main View" is something *you* want the system to do. Ask yourself, would the Actor say to him/herself, "Today I want to use the system to Present Main View." Not likely. The "Present Main View" may be one thing your system needs to do to fulfill the actor's request, but that is a system viewpoint (i.e., the system would say "I want to present a view to the user"). Approaching from the Actor's viewpoint keeps you focused on the actor's needs, i.e., the system requirements, not the implementation.

Value - The use case must provide Value to the user. "Authenticate User" is OK again, since it is included and since being authenticated enables the user to do whatever he or she came to do. But as stated under "A" above, the actor probably didn't come to "Present Main View." The actor may want to "Sell Stocks" or "Make a Call" or "Buy a Product." Making a call gives the actor what he wants (i.e., provides value). Presenting a view may be needed, but in itself it provides no inherent value. This is an indication that a use case may be too small or that it may involve just a few steps that belong in a larger use case.

Entire Flow - Does the use case (with all its includes and extends) provide an Entire flow? "Authenticate User" is OK, since authentication is a whole process (whereas "Provide Account Number," "Validate Account," "Enter PIN," etc. would not be entire flows). But once again, "Present Main View" is probably not a complete flow. It seems to be just part of a larger "transaction," such as "Make a Call," with the actor.

I hope this helps with your use cases. Good Luck!
Bob

Dear Rational Edge:

A few days ago I read the article "UML Activity Diagrams: Detailing User Interface Navigation" by Ben Lieberman [October 2001].

This article was very inspiring for my work. Our project will use this manner of modelling. But we encountered the need for an extension which I would like to suggest.

If you are modeling an Application Interface, you need to model dynamic menus that contain entries which are disabled or are unvisible depending on the state of the application. Therefore I would suggest modeling the name of the menu-entry in the event-entry of the state-transition and special conditions in the guard condition entry (Rational Rose).

For example if you have to model a menu entry "Print," which is only enabled if a document is open, the Transition would be labeled with "Print|ALT P [enabled if doc is open].

What do you think about this idea?

Many thanks in advance,
Detlef Heinze

Ben Lieberman responds:

Dear Detlef,

I am very pleased to hear that you are interested in adopting my approach to UI modeling using UML Activity diagrams. You may also wish to review some of my other *Rational Edge* articles (particularly the April and May on UML Activity Diagrams).

As for your suggestion, I have a few thoughts:

1) Activity diagrams can support state objects as well as activity objects. You may wish to have the Activity transition guard condition as you describe, or you may wish to indicate the states that the menu can have, such as "disabled" or "enabled" for a menu choice. To save on space in the diagram, you may wish to have one state object for each full menu and use the action section of the transition to indicate the change of state for the menu choice.

2) Sometimes an element is visable, sometimes it is invisable, sometimes it is greyed-out, sometimes the behavior for the element changes depending on the state of the system processing (such as during a transaction). You may want to try embedding the state changes describing this behavior inside of the activity icon (drag and drop the icon and it will embed). This will nicely show the possible states, the conditions that will change state, and at what point in the processing flow these changes become relevant.

Sincerely,
Ben

Dear Rational Edge:

I have two questions for you:

1. In one of the RUP workflows do we get the description of "how to configure it"?
2. Is UML in RUP obligatory (a prescription)?

Thanks a lot!

We asked Bruce MacIsaac and Philippe Kruchten of Rational's RUP Development Group to respond.

from Bruce MacIsaac:

As an answer to your first question, configuration of the process is described in the Environment discipline. The artifact "Development Case" is used to document the configured process.

As for your second question, no, UML is not obligatory. RUP recommends Visual Modeling as a best practice, however, there are notations other than UML which can be used. UML has the widest industry acceptance, and so is generally the best choice. Much of RUP's guidance is expressed in UML terminology, and so fits best when UML is used.

from Philippe Kruchten:

The Environment Discipline is concerned with the process itself. It contains roles activities guidelines and templates to set up, install, configure the process to suit a given organization. It contains tools to regenerate a RUP web site.

In the Analysis and Design Discipline, the RUP makes extensive use of UML, both as a tool to express design, a reference for terminology and concepts, and to describe most examples. UML is becoming the lingua franca of software description.

But on another hand, Analysis and Design represents only a fraction of what software development is about, and only about 20% of the RUP. We have some telecom customers who have used the RUP with SDL (System Description Language from UIT).

You can get more info on the RUP at http://www.rational.com/products/rup/index.jsp and in particular check an evaluation version at http://www.rational.com/tryit/rup/index.jsp

---

On Peter Eeles' "Capturing Architectural Requirements" [November 2001]

A reader from Brazil writes:

I enjoyed very much Peter Eeles' paper on "Capturing Architectural Requirements." It succeeds in summarizing a rich set of significant questions on the development of sound software architectures.

As well as a reader from India:

The Article "Capturing Architectural Requirements " by Peter Eeles was very impressive as was the Appendix info provided to the readers. Excellent articles are coming to us from *The Rational Edge*! Keep it up.

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