Inside:
- The art of modeling
- A new look at RUP and systems engineering
- IBM Rational Rapid Developer — more control for architects
- Project planning best practices

Cover Story:
A guide to RUP implementation
Editor's Notes:

"Ah, there's the RUP"

Over the past several years, The Rational Edge has published a number of articles about implementing the Rational Unified Process, or RUP, as a means to improving software development predictability, performance, and repeatability. Just to name a few, these articles include Leslie Probasco's "The Ten Essentials of RUP" (December 2000), Gary Evans' "A Simplified Approach to RUP" (January 2001), D.J. deVilliers' "Introducing the RUP into an Organization" (January 2002), and Maria Ericsson's and Per Lofnertz's "Repeatable RUP Deployments via Solution Components" (May 2003). In all of these articles, the focus is on simplifying the often unwieldy process of changing organizational behaviors and attitudes -- especially in organizations previously committed to traditional, or "waterfall," development processes.

This month, we add to the growing body of knowledge regarding successful implementation of RUP. Many sound approaches recommend introducing RUP via a small project, during which you can try out the four phases, "Inception, Elaboration, Construction, and Transition," before moving on to mission critical projects, but the new "RUP implementation guide" actually offers a way to treat the introduction of RUP as a project in and of itself. Conceived, written, and vetted by a committee including many customers and a handful of IBM Rational RUP experts, this guide was created "...to advise organizations on best practices to follow as well as practices to avoid." The authors characterize the guide's content as "what we wish we had known" when they began RUP implementation projects in their own organizations. So, whether you see yourself
becoming your own organization's RUP champion, or simply want a comprehensive method for getting teams to standardize on RUP practice, this guide is for you.

Taking a different perspective on RUP, Murray Cantor returns this month with his latest view of systems engineering as a software-related discipline and how RUP can support it. This is the first installment in a three-part series that will run through October. And there's even more RUP-related material for you: Dennis Moya's "Delivering packaged application solutions with Rational Unified Process" presents challenges and solutions for incorporating off-the-shelf software for CRM and ERP systems into business operations, and Eric Cardozo and D.J.deVilliers take us on a RUP-guided tour of "Project planning best practices."

As many IBM Rational customers know, visual modeling with the UML is one of our long-held Rational best practices for developing software. But before we ever thought about designing and building software, our brains were at work modeling the world around us in hundreds of ways, according to Ben Lieberman, whose series "The art of modeling" begins this month. He examines modes of system analysis, along with practical applications and techniques for modeling that can help software developers think about the essential elements of design. At the other end of the theory-practice spectrum, Joe Noonan explains "the major attributes of IBM Rational Rapid Developer that enable architectural and construction control, thereby ensuring high-quality business applications."

Book reviews this month include Beth Benoit's on "Waltzing with Bears: Managing Risk on Software Projects" by Tom DeMarco and Timothy Lister; and Peter Hanschke's on "Writing Better Requirements," by Ian F. Alexander and Richard Stevens. And our August Kite is by Grady Booch, who reflects on why so many of us take personally our efforts in software development. As ever, he'll give you something to think about.

Happy iterations,

Mike Perrow
Editor-in-Chief
RUP implementation guide

Part I: Recommended strategy and typical issues and risks

by the RUP Implementation Workshop Group

IBM Rational Unified Process,® or RUP,® is a process platform for software development that supports a wide range of project types, ranging from custom business applications, to commercial-off-the-shelf (COTS) product implementations, to civil and military government applications, and many others.

Organizations can implement RUP for either one of two purposes:

- To execute a single software development project or
- To establish RUP as the standard software engineering process within the organization

The first purpose applies to a software development project using RUP, whereas the second applies to a project specifically established to implement RUP within an organization.

The goal of this guide is to help with the second of these purposes -- to implement RUP within an organization as a project in its own right, independent of a specific software development project.

During the 2002 Rational User Conference, the RUP product team sponsored a workshop to capture experiences -- both good and bad -- of organizations that had implemented RUP for the second of these purposes. Workshop attendees (listed in the Appendix) had "real world" experience in implementing RUP organization-wide and represented a broad spectrum of users.

Working together with the RUP product team, these participants created an outline for a RUP implementation guide to advise organizations on best practices to follow as well as practices to avoid. They characterized the guide's content as "what we
wish we had known" when they began RUP implementation projects in their own organizations.

After the conference, a subset of this group, along with some new participants, conducted a series of online workshops to produce the "RUP implemention guide" that we present in this two-part Rational Edge series. As the title suggests, the content targets RUP implementers, whose job is to champion the RUP implementation effort within the organization and then serve as a mentor during the project lifecycle.

Part I provides an overview of a strategy we recommend for implementing RUP in an organization, which is based on RUP itself. It also discusses typical types of issues and risks that organizations face when first adopting RUP.

Part II will summarize key lessons the workshop members learned in implementing RUP, and describe practices to follow and those to avoid.

This guide is intended for groups that have already decided on RUP and are looking for a place to begin. It is not intended to serve as a tutorial for readers to learn about RUP. IBM Rational provides a RUP "Getting Started" guide on Rational Developer Network (RDN) to licensed IBM Rational customers. The Resources section at the end of this article also suggests more ways to learn about RUP.

The RUP implementer

In most cases an organization designates a person or group to "champion" a RUP implementation within that organization. In this series, we refer to this person or group as the "RUP implementer" and make the following assumptions about that person and his or her organization.

- The organization has already selected RUP as the process framework for a project or as a standard.
- The RUP implementer is not experienced in implementing RUP, but has acquired an understanding of RUP fundamentals through one or more of the following:
  - Experience using RUP on projects.
  - IBM Rational training courses (classroom or Web-based) on RUP.
  - Books on RUP.
- The RUP implementer knows her organization's needs and business goals with respect to process adoption, and understands the organization's formal and informal politics.
- The RUP implementer has a general knowledge of the software development process.

Motivation for implementing RUP

Organizations may choose RUP for a variety of reasons. We will discuss three primary benefits below.

Achieve a common "language"
Today, companies often do business in multiple countries, as a result of either natural growth or business mergers. When they attempt to institute common practices in their global offices, the lack of a mutual language often gets in the way.

A shared process can be a solution to this problem. It provides an "unambiguous language" within the company that everyone can understand, despite geographical and organizational differences. It forges links and creates common understanding among all players within the project -- between customers and upper management, project managers and developers, developers and testers. And it covers all aspects of the software development lifecycle, from business modeling to deployment.

Because it includes Unified Modeling Language (UML)-based modeling concepts, RUP, especially, provides a universal means of communication. UML is an easy-to-understand visual language for modeling, specifying, constructing, and documenting software-intensive systems.

**Achieve a repeatable and predictable process**

Organizations that have experienced repeated budget and schedule overruns often turn to RUP as a solution. Many organizations have no institutionalized process, so they must "reinvent the wheel" each time they work on an application development project. Others use a traditional, non-iterative process that results in crises at the end of the project lifecycle and products of poor quality. With the iterative approach embodied in RUP, they can reduce risks early in the project lifecycle and more easily meet schedule and budget constraints.

**Institute a scalable framework**

A great benefit of implementing RUP is that it gives organizations a framework that can (and should) be adjusted to their particular needs. Typically, they begin by implementing a standard configuration of RUP. However, this is not always enough. Today's IT organizations are involved in such diverse development projects that they often need more than one RUP configuration.

Starting from the standard "out of the box" RUP configuration, organizations can leverage the support tools provided by IBM Rational, including RUP Builder and the IBM Rational Process Workbench (RPW) products, to create RUP configurations that meet the specific needs of the types of software projects that are typical for the organization. RUP Builder and RPW can be used to create "upscale" or "downscale" RUP configurations, depending upon the project characteristics. In this context, "upscaling" refers to adding more process elements, such as organization-specific artifacts, guidelines, tool mentors, and so forth, to augment the standard RUP configuration. "Downscaling" refers to selecting a specific subset of RUP process elements for use on a project.

Here are just a few examples of project types that organizations may define:

- Small projects (scale down)
- Large-scale projects (scale up)
- Mission critical projects (scale up)
- Web projects (scale down)

It should be noted that, starting with Version 2003 of RUP and RUP Builder, there is
more "built in" support for scaling RUP through a componentized process framework, and you can select only the subset of RUP that makes sense for the problem at hand. The informal and formal Process Plug-Ins can help you select the appropriate level of formality through different sets of artifact templates and examples. The predefined small, medium, and large project configuration templates in RUP Builder can help you select the right starting point for creating organizational RUP configurations.

In addition, RUP can be adapted to foster communications within decentralized enterprises having multiple locations around the globe.

**An implementation strategy**

This section describes an implementation strategy for adopting RUP that is based on RUP itself. Although we can recommend this approach, based on our collective years of experience with RUP implementations, keep in mind that this is just one of several possible approaches -- and it should be tailored to fit the specific characteristics and needs of your organization. Also note that, while the RUP implementation strategy we recommend should be considered a project in its own right, a thorough implementation will involve two or more pilot software development projects as a means for testing the success of your organization-specific RUP configuration.

**Use RUP phases**

To adopt RUP successfully, organizations should use a phased, incremental approach. We recommend the four phases specified in RUP: Inception, Elaboration, Construction, and Transition. Table 1 shows the purpose and objectives of each phase.

**Table 1: Recommended phases for a RUP implementation project**

<table>
<thead>
<tr>
<th>Phase</th>
<th>Purpose</th>
<th>Important results after the phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inception</td>
<td>Define the vision and scope for the RUP implementation; make the business case for the project to obtain support and funding from key executive management sponsors.</td>
<td>Stakeholders approve the RUP Implementation project vision and scope and allocate sufficient funding. An assessment of the organizations' current process and development environment has been done; this provides the basis for determining the process and tool changes needed for RUP implementation.</td>
</tr>
</tbody>
</table>
### Elaboration

- **Define the detailed requirements and architecture for the RUP configuration the organization will use.**
- **Implement an initial version of the organization-specific process content changes. The project team has specified, designed, and implemented one or more organization-specific RUP configurations. The team has also specified and implemented a tools configuration that can support pilot projects in the development environment.**

### Construction

- **Use the defined RUP configurations on selected pilot projects and collect feedback.**
- **The team has conducted pilot projects using the initial RUP configuration(s) and recommended supporting development tools. They have also collected and analyzed metrics from the pilot projects to determine if these projects realized the defined business goals for using RUP.**

### Transition

- **Roll out RUP to the entire organization.**
- **Deploy the process and tools to the entire organization.**

---

**Inception phase**

The focus of activities in the initial phase of a RUP implementation project is on clearly defining the vision and scope for implementing RUP process in the organization, and securing the necessary support and funding for the project through the organization’s executive management.

**Key activities**

The primary work in defining the scope of the process implementation is:
• **Analyze the business problem.** Determine the high-level business goals and objectives of adopting RUP in the organization, and record them in a Business Vision document. Develop a Business Case document to request funding to assess the organization's current systems development process.

• **Conduct an organization assessment.** Assess the current organization’s process and development environment to determine what changes are needed to meet the business goals and objectives for the RUP implementation. From this assessment, identify the development organization's key business process needs as they relate to RUP. Prioritize the business needs, and from these needs identify the high-level features and architecture you need to configure and/or customize RUP. The feature list and architecture description should identify any new (organization-specific) or modified RUP process elements, and any process elements that will not be needed. Revise the business case to include the effort to develop configurations of RUP that are required to meet the needs of the organization, and then present the business case to executive management to secure funding for this purpose.

• **Planning.** Define the tasks needed to implement the defined process configuration(s). You should develop the process configuration(s) in an iterative manner and prioritize the process implementation according to the organization's most urgent needs.

**Lifecycle Objectives (LCO) Milestone**

The Inception phase for a RUP implementation project ends with a review of the "final" business case for the proposed RUP configuration development by the executive management stakeholders. In some cases this represents the second "go/no-go" milestone in the Inception phase, as there may have been an earlier "go/no-go" review for the initial business case, in order to obtain funding for the development organization assessment.

**Elaboration phase**

The focus of the Elaboration phase for a RUP implementation project is to develop the initial RUP configuration(s) for the organization that you can try out (test) in one or more pilot projects.

**Key activities**

The primary work items in this phase are:

• **Analyze process requirements.** Starting from the high-level features and process architecture defined in the Inception phase, refine the list of new process content materials (guidelines, concepts, activities, artifacts, tool mentors, etc.) you need in the organization-specific RUP configuration. Canvas existing process assets for items that you can reuse in this configuration. For each item you need to develop or modify (in each iteration), develop detailed outlines of the process content.

• **Refine the process architecture.** Starting from the high-level process model for the organization-specific RUP configuration, refine the "design" changes to the RUP process framework as needed, based on the detailed process content changes you identified during requirements analysis. This work applies mainly when the project team is using the Rational Process...
Workbench (RPW) tool.

- **Develop process content.** Develop the detailed, organization-specific process content. Content can be in the form of guidelines, concepts, activities, artifacts, templates, checkpoints, tool mentors, whitepapers, and so forth.

- **Generate process configuration(s).** Create the initial RUP configurations either manually, using the RUP organization Web example available for download on Rational Developer Network ([www.rational.net](http://www.rational.net); authorization required), or by publishing a custom RUP Web site using RPW and/or RUP Builder.

- **Planning.** Develop a plan for the next phase (Construction), in which the process configuration(s) will be rolled out to one or more pilot projects.

In addition to developing the process configuration, the RUP implementation team also needs to consider what development environment tools they need to support the process. They should perform the following work items in combination with those listed above.

- **Analyze development environment needs.** Based on the process automation and support needs you identified during the organization assessment, define tool requirements/features, and research available tools. Conduct evaluations of various tools as needed.

- **Define the development environment.** Select and acquire tools to support the process. Purchase sufficient licenses to support one or more pilot projects, as well as a training session for rolling out your organization-specific RUP configuration.

- **Configure the development environment.** Install and configure tools to support processes. Configuration includes setting up organization-wide tool repositories and/or repository templates, as well as defining tool usage guideline.

In some situations, a RUP Implementation project is part of a larger program for organizational change (i.e., business re-engineering). In these cases, you may need to do additional work in the RUP implementation project to address changes within the IT organization (and interfacing organizations) that are adopting the RUP. Although a discussion of organization change management is beyond the scope of this series, we include the following work items in the workflow as potential "touch points" for a larger organizational change program.

- **Analyze needs for organizational change.** Analyze current roles and responsibilities of IT organization members (and interfacing organizations if necessary). Determine whether any changes are needed to the organization structure and/or technical/managerial position definitions to implement the process changes defined for the RUP configuration.

- **Define organizational changes.** If you do need changes to the organization's structure, define the new structure and map the current structure to it.

- **Plan organizational change management.** Define an implementation plan for transitioning the current organization to the new organizational structure.
Lifecycle Architecture (LCA) Milestone

The Elaboration phase for the initial release of a RUP implementation project ends with a milestone review of the organization-specific RUP Configuration and the associated development tools infrastructure. The level of ceremony and formality for this review will vary significantly, depending on the size and complexity of the organization adopting RUP. The purpose of the review is to demonstrate to executive management stakeholders that the RUP configuration and supporting tools are ready for roll out in support of pilot projects.

Construction phase

The focus of the Construction phase is on the initial release(s) of the organization-specific RUP configuration and supporting tools for one or more pilot projects within the organization. Testing the RUP configuration via pilot projects serves two purposes: 1) to introduce the new process and tools incrementally within the organization and 2) to get early feedback on whether the organization-specific RUP configuration and supporting tools are providing the desired business value/results defined in the project's business goals and objectives.

At the end of the Construction phase, the implementation team should revise the RUP process configuration and supporting tools configuration, based on feedback from pilot project team members (user perspective), and an analysis of the performance (business perspective) of the pilot project(s), using objective metrics defined at the beginning of Construction. These metrics may be either organization-specific or based on an industry standard assessment framework such as the Software Engineering Institute's (SEI) Capability Maturity Model Integrated (CMMI) or SPICE (ISO/IEC 15504 standard).

Key activities

The primary work items in the Construction phase are:

- **Select pilot project(s)**. Decide on pilot project(s) to try out your initial organization-specific RUP configuration(s).
- **Develop training**. Design a training program on the process and tools for pilot project team members.
- **Establish a measurement program**. Define what metrics and/or industry assessment framework you will use to evaluate the performance of each pilot project, relative to the implementation project's business performance goals.
- **Train pilot project(s) staff**. Deliver training on the organization-specific RUP configuration and supporting tools to members of the pilot project team.
- **Conduct pilot project(s)**. Execute the projects, using the organization-specific RUP configuration and supporting tools.
- **Monitor and support pilot project(s)**. Support activities to provide continual mentoring, just-in-time training, problem resolution, and so forth, to the pilot project(s). Also includes activities to collect status and assessment metrics data on the progress of the pilot project(s).
- **Refine measurement program**. Review the effectiveness of the type, frequency, and procedures for the metrics you collect about the pilot projects and make any necessary adjustments.
Refine training program. Survey pilot project team members on the effectiveness of the initial process and tool training, ongoing mentoring, and any just-in-time training conducted during the pilot project.

Planning. Develop a plan for managing the rollout of the tested configuration to the entire organization in the Transition phase.

The number of iterations in the Construction phase should be based on the number and duration of designated pilot projects that you will use to "test" your RUP configuration.

Initial operational capability (IOC) milestone

Upon completion of each pilot project, the RUP implementation team needs to analyze the metrics data collected from the project. The analysis should include a determination of how using the RUP process affected the pilot project team's performance relative to the business improvement goals set forth by the stakeholders. The RUP implementation team should also meet with the pilot project team to conduct a "lessons learned" discussion regarding the effectiveness of the RUP configuration.

If the RUP implementation team is using an industry process assessment framework to measure the organizational impact of using RUP, the team should assess the performance of the pilot project team to measure changes in their process maturity level.

When their analysis is complete, the RUP implementation team can present the results to project stakeholders, along with any recommended final changes/improvements to the organization-specific RUP configuration. These changes should be made prior to either starting the next pilot project or releasing the RUP configuration to the entire organization.

The IOC milestone is complete, and the Construction phase ends, when project stakeholders approve rollout of the RUP configuration and supporting development environment to the full IT organization -- and any other business organizations involved in the systems development process.

Transition phase

The focus of the Transition phase of a RUP implementation project is on rolling out the organization-specific RUP configuration and supporting tools to the entire organization. The implementation team's activities are similar to those in the Construction phase, but the scope of the training, mentoring, and work expands from supporting one pilot project to supporting multiple "real" projects.

Key activities

The primary work activities during this phase are:

- **Implement process configuration (organization wide).** Make the RUP configuration available to the full IT organization and any other business organizations that participate in the systems development process. Deploy the supporting tools and tool customizations.

- **Train project staff.** Using the training model and processes defined for the
pilot project(s), conduct training for each "real" project team adopting the RUP and supporting tools.

- **Conduct projects.** Use the organization-specific RUP configuration on application development projects.

- **Support and monitor projects.** Provide process and tools mentoring support, just-in-time training support, and artifact review support to project teams.

- **Manage change requests.** Maintain the organization-specific RUP configuration, based on change requests and defects reported by the project teams. Also maintain the development environment configuration and any organization-specific customizations to the tools.

- **Planning.** Develop plans for ongoing maintenance and enhancement releases of the deployed RUP configuration(s).

**Product release and maintenance release milestone(s)**

The Transition phase ends with the initial release of the RUP configuration across the organization. Once it is deployed, the RUP configuration will need to be maintained, refined, and enhanced, depending on the needs of the organization.

Maintenance releases may be critical fixes to the existing configuration requested by a project team, or regularly scheduled configuration upgrades. They may also be entirely new configurations designed to support different project types.

**Ongoing education and training**

Continuous education, training, and mentoring are an important part of the RUP implementation. Implementers, process users, and stakeholders that do not directly use the process but need to understand it, all require training. A mentor -- an experienced RUP implementer who can guide trainees through hands-on work during at least the initial part of the implementation -- can also be of tremendous help in institutionalizing the process.

**RUP implementer(s)**

RUP implementers will need to become intimately familiar with RUP. They should begin by focusing on which of the RUP six best practices (we will discuss these in detail in Part II) have been chosen to be implemented first. RUP training is available through IBM Rational University and the IBM Rational Developer Network (RDN), which offers Web-based training. Additional web-based training is available through the many Webinars offered by IBM Rational on the main corporate site (www.rational.com).

In addition, implementers can get intensive instruction on RUP by attending the annual IBM Rational User Conferences, reading IBM Rational whitepapers, joining IBM Rational user groups, and consulting the other resources listed under References below.

Conduct a skill assessment for each implementer to determine the best course of action for training and education. Implementers should have the skills to determine workflow variations for each of the disciplines involved in the best practice scheduled for implementation.
Process users

Users of the RUP process will need role-specific training for each workflow in each discipline they will use. They will need to learn not only the activities involved in the workflow but also how to use the associated tools. Training should educate them about the benefits of each activity so they understand the reasons why they are doing the work.

Either outside vendors or in-house trainers can educate these users. By using an outside vendor you can typically train people more quickly, but the RUP implementer should decide which type of training is best for the organization.

Implementers who choose in-house training need to understand all planned and implemented changes to the workflow and tools before developing the training. They should also make sure they have a good understanding of how RUP will be used in the organization to support each best practice.

Stakeholders not directly using the process

Even stakeholders who will not directly perform workflow activities need to be educated on how RUP will be used in the organization, how it will affect their projects and role, and the organizational benefits that the process will bring. They need to understand the "big picture" business advantages to recognize that a RUP implementation is worth the resource investments it will require. Educate this group in a half-day session that provides a short overview of RUP, explains how the organization will use it, and details the benefits it will yield.

Issues and risks

This section discusses common issues and risks organizations encounter in implementing RUP. Many risks depend on the type of business and technology environment in which the RUP implementer works. We recommend that you consider these along with others that you might face.

An issue is a subject/question/problem that you are already facing in your RUP implementation endeavor; a risk is a danger/threat you could potentially face. Not all issues pose risks, but all risks are caused by issues and result in more issues.

Issues and risks take different forms at different levels in the RUP implementation. Those that arise at the project level also affect the organizational level. The converse is also true: Issues and risks at the organizational level cause project-level risks. We can categorize the risks in a way that lets us understand them better and determine whether those risks might arise in a particular environment.

Tables 2 and 3 show issues and risks for RUP implementation by common categories. Note that these issues and risks apply to implementing RUP in an organization and/or project, which is very different from executing a project using a prior implementation of RUP. The RUP product itself discusses project execution risks (e.g., RUP Activity: Identify and Assess Risks), but these are beyond the scope of this series. The tables use the same basic categories for risks and issues, because many issues have associated risks.

Table 2: Issues by category
<table>
<thead>
<tr>
<th>Issue Category</th>
<th>Description</th>
<th>Issues</th>
</tr>
</thead>
</table>
| Sponsorship    | Does the RUP implementation initiative have adequate sponsorship at different levels in the organization? | - Complete process change may require six months to two years for a large organization.  
- Upper management support is a hard sell, but middle management is even harder to get on board.  
- The RUP implementer needs a long attention span to assemble sufficient support.  
- If the RUP implementer has preconceived notions that do not align with those of the sponsors, it may be difficult to obtain those sponsors' support. |
| Budget         | Does the initiative have an adequate budget to cover the duration of the project? | - RUP implementation is a project that needs dedicated resources and adequate funding that extends throughout the project lifecycle.  
- The RUP implementer needs time and a |
<table>
<thead>
<tr>
<th>Culture</th>
<th>The culture at each level in the organization influences training, process institutionalization, and other factors that can cause issues.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• In most organizations, the key issue is not adopting a process (RUP), but effecting a deep cultural and organizational change. Trying to institute process before making cultural changes may be putting the cart before the horse.</td>
</tr>
<tr>
<td></td>
<td>• The existing culture may be reactive rather than proactive.</td>
</tr>
<tr>
<td></td>
<td>• Training may be difficult if people are resistant to adopting the process; also, training may not be consistent across levels.</td>
</tr>
<tr>
<td></td>
<td>• Among software developers, the traditional (waterfall) process approach may be hard to eradicate, and/or the organization might not be staffed properly for an effective process implementation.</td>
</tr>
<tr>
<td></td>
<td>• Experienced vendor consultants can</td>
</tr>
</tbody>
</table>
be a valuable resource for a RUP implementation; however, if internal staff refuses to work with them collaboratively, these consultants may be saddled with all responsibility for the implementation but have no authority to direct or manage activities.

- Project management practices may be weak.
- Consultants who are not up to speed with RUP may do more harm than good.

<table>
<thead>
<tr>
<th>Priorities</th>
<th></th>
</tr>
</thead>
</table>
| Different priorities, real and/or assumed, at the different levels, can cause issues. Departmental or individual agendas, and conflicting notions about the purpose behind the RUP implementation initiative can cause issues. | • The IT organization may be under intense pressure to get the job done as quickly as possible.  
• Process may always have been a low priority in the organization, or past process initiatives may have failed.  
• The organization may have other needs that management considers more pressing. |
<table>
<thead>
<tr>
<th>RUP misuse</th>
<th>This includes all issues arising from improper use of RUP.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- The organization may have used a version of RUP in the past that is incompatible in places with their current version.</td>
</tr>
<tr>
<td></td>
<td>- The organization may have created confusion by deviating from RUP terminology.</td>
</tr>
<tr>
<td></td>
<td>- RUP may not cover all areas important to the organization in great detail.</td>
</tr>
<tr>
<td></td>
<td>- The organization may be changing technology</td>
</tr>
</tbody>
</table>

- Management may say they support a process implementation but be unwilling to commit resources to accomplish it.
- Customers and consultants may have different (conflicting) agendas.
- Management may not understand the fundamentals of the CMM process maturity model and the link between achieving higher maturity levels and instituting RUP.
platforms or trying to institute a new technology that affects business processes -- an ERP system, for example.

- The organization may have multiple sites and inadequate means for coordinating them.

- The organization may be successful in starting to do requirements management via use cases, but then fail to carry out other RUP activities. For example, it may not do frequent builds, automated testing, or effective configuration management.

- The organization may not recognize that RUP is a "work-in-progress" and may attempt to apply RUP in a rigid way that leads to "analysis paralysis."

- RUP does not provide detailed guidance on planning, and managers may not be experienced
### Table 3: Risks by category

<table>
<thead>
<tr>
<th>Issue Category</th>
<th>Description</th>
<th>Risks</th>
<th>Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sponsorship</td>
<td>If the RUP Implementation initiative does not have sponsorship at all levels, it risks failure for a number of reasons.</td>
<td>● Inadequate management support puts project funding at risk and may result in a &quot;one strike and you're out&quot; attitude.</td>
<td>● Do not shroud implementation project in mystery. Educate participants at all levels early and often to demonstrate value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● Inadequate support at lower levels may create widespread resistance to implementing process steps, resentment about time demands, and alienation with respect to project assessment.</td>
<td>● Once the implementation begins, do not conduct &quot;Spanish Inquisitions.&quot; Assess process in a non-threatening fashion.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● Inadequate commitment on the RUP implementer's part may cause him or her to give up on the project at the first sign of difficulty.</td>
<td>● The RUP implementer should seek support from external sources such as IBM Rational to help overcome internal challenges.</td>
</tr>
<tr>
<td>Budget</td>
<td>If the initiative does not have an adequate budget that extends across the entire project lifecycle, it cannot succeed.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------------------------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Conditional budgets (i.e., further funding depends on meeting certain goals) may kill the project if the team encounters problems in the early stages.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The RUP implementer needs funding to &quot;sell&quot; the concept throughout the organization and spearhead necessary cultural changes.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The project team must secure funding for a realistic timeframe. They should then plan carefully, so that they can demonstrate success early on.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Culture</th>
<th>If the culture at each project level is not compatible with RUP principles, then training may not be successful, and people may not ultimately accept the process.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Some people or parts of the organization may be receptive to training, whereas others resist or refuse it. This lack of consistency creates communication and process problems that can sabotage the project.</td>
</tr>
<tr>
<td></td>
<td>• If project management practices are weak, there may be no way to enforce compliance with training and process implementation, which can create chaos.</td>
</tr>
<tr>
<td></td>
<td>• If the organization's software developers have an entrenched</td>
</tr>
<tr>
<td></td>
<td>• The RUP implementer should devote time and resources to education and cultural change at all levels of the organization before the implementation project actually begins.</td>
</tr>
<tr>
<td></td>
<td>• During the implementation, the project manager should constantly monitor and assess how the implementation is proceeding, paying careful attention to feedback and user behavior as well as metrics.</td>
</tr>
<tr>
<td></td>
<td>• The RUP implementer should become a RUP mentor, working hands-</td>
</tr>
</tbody>
</table>
**Priorities**

Conflicting priorities among different departments or individuals may cause delays and different interpretations of project assessments that put the project at risk.

- Stakeholders may insist on a short timeframe so that the IT organization can get on with "real" software development projects. If this forces the team to skip important steps in the implementation, the project may fail.
  - If process has always been a low priority in the organization, or if past process initiatives have failed, people might not take the implementation effort seriously and will not be willing to work at it.

- Remedies are similar to those for **Culture** risks. Organization-wide education about the benefits of RUP and what will be required for a successful implementation is critical.

  Also, it is very important for the RUP implementer to keep in mind that project success is the top priority for the project delivery team, and the success of the process implementation effort comes second.
In this article we have discussed some elements of our recommended approach to RUP implementation, which are based on the approach embodied in RUP itself. In Part II we will discuss best practices associated with this approach and provide more practical advice on how to make your RUP implementation successful.

**Resources**

There is a wealth of information about RUP in the following books, published by Addison Wesley:


These books provide a complete description of the RUP structure and also discuss key concepts that shape the RUP process framework (i.e., that RUP is a use-case driven, architecture centric, and iterative approach to software development).
The Rational Edge -- August 2003 -- RUP implementation guide Part I: Recommended strategy and typical issues and risks

There is more information about specific topics related to RUP in whitepapers, archives of The Rational Edge (www.therationaledge.com), and archived Webinars on IBM Rational's corporate Web site (www.rational.com, Events > Webinars). Topics of these media range from how RUP best practices for software engineering support the Software Engineering Institute's (SEI) Capability Maturity Model Integrated (CMMI), to how RUP relates to the eXtreme Programming (XP) process.

Readers who have attended past Rational User Conferences or who will attend the IBM Rational User Conference 2003, can also access conference presentations on RUP topics through Rational's corporate Web site.

The RUP product itself also provides information on how to configure and implement the process into an organization and/or on a specific project. The RUP Environment Discipline describes the basic activities needed to configure the process for use on a project and provides general guidelines and concepts about the RUP process.

For readers of this Guide who have already purchased IBM Rational products, Rational Developer Network (RDN) has articles and documentation in the RUP product section of the Rational Products Knowledge Center, and a RUP Discussion Forum. RDN also has a separate Knowledge Center devoted exclusively to the development of special add-ons (plug-ins) to the RUP process framework called the RUP Plug-in Exchange. The Knowledge Center provides a set of downloadable RUP "Plug-ins" that contain additional RUP-based process guidance on specific "horizontal" technologies such as .NET, J2EE, and so on, for specific vertical domains.

Both instructor-led and Web-based training courses on RUP are available through IBM Rational University. See descriptions on www.rational.com under Services -> Education: US and Canada and Education: Worldwide.

Appendix: Participants in the RUP Implementation Workshop at the Rational User Conference 2002

Workshop hosts

Gary Pollice, RUP Curmudgeon, Lexington
Erin Curtis, RUP Marketing, Cupertino
Mike Barnard, RUP Product Manager, Vancouver
Philippe Kruchten, RUP Pundit, Vancouver

Invitees
Brenda Hoover, Aetna, Pennsylvania
Kristin Leard and John Garis, The Hartford, Pennsylvania
Judy Richardson-Mahre, Venturi Technology, Minnesota
Scott Grengs, Wells Fargo, Minnesota
Kirti Vaidya, Covansys Corp., Illinois
Warren Steger, Applied Information Sciences, Virginia
Göran Grahn and Boris Karlsson, Volvo, Sweden
John Pritchard, Freddie Mac, Virginia
Julian Holmes, Cap Gemini Ernst & Young, UK
Martin Piller, Cap Gemini Ernst & Young, Netherlands
James Behling, Accenture, Illinois
The art of modeling

Part I: Constructing an analytical framework

by Ben Lieberman, Ph.D.
Principal Architect
BioLogic Software Consulting, LLC

Models are now, and always have been, an integral part of the human experience. We create models of the world about us based on our five known sensory inputs: visual, auditory, textile, olfactory, and gustatory. These models inform us of changes in our immediate perception and also permit understanding of these changes in comparison to past events. As infants we develop sophisticated internal mental models of motion, shape, distance, time, and cause/effect in an effort to relate to the new and confusing world around us. Starting from a nearly clean slate, we must build an internal understanding for every new experience.

Our internal mental models of the world are also key to the communication of complex concepts to other people. Since no two people share the exact same set of experiences, no two mental models will be exactly the same. For example, it would be impossible for people living in a primitive hunter-gatherer culture, without contact with the modern world, to understand the principles of refrigeration simply by hearing a description of the mechanics; the concepts are completely foreign to their experience.
To succeed, first you would need to have a shared communication medium (language, pictures, gestures, etc.), and translate your internal mental model into ones that can be related to their understanding of the world: examples of food storage, effect of cold temperatures on spoilage, and so forth. In other words, you would need to align your explanation of a phenomenon to a shared set of knowledge with your intended audience. Without such a common information base, it is very difficult for one person to explain his or her understanding of a phenomenon or system to another. Moreover, the probability of misunderstandings increases dramatically.

Similarly, when developing software systems, every person on the development team must have the same understanding of the needs and purpose for the system. This information is often gathered by a few specialists (e.g., requirements engineers) who then present this information to the remainder of the team, using a series of visual and textual models. Because these models are built from the viewpoint of the specialist, it is important for the "modeler" to understand his own biases and assumptions, since they will not likely be shared by all members of the development team. This need for clear communication of complex concepts is the purpose of creating models.

This series of articles will focus on the principles and practices of model conception, creation, and presentation. This first article will be primarily concerned with the principles of system analysis, observation, experimentation, and the techniques that can be employed to study a selected system. The second article will focus more on the practical applications of modeling, with an emphasis on software development models. The final article will focus entirely on techniques and considerations for presentation of information, using UML for the visual examples.

**Perception and representation**

A model can be defined as the simplification of a complex system for the purpose of communicating specific details. In this way a model operates much like a lens; it focuses attention on items of direct concern while obscuring or omitting everything else (Figure 1).
For example, recall how book reports are constructed (for those who can remember back to elementary school). The intent is to capture key elements of the book; highlighting primary plot lines, characters, situations, and so forth, rather than copying the book in its entirety. In this way someone reading the report can gain a reasonable understanding of the important aspects of the work without having to read the full text. This is also a good example of the concept of abstraction and how it is used to conceptualize a model (see the Conceptualization and capture section below).

In my opinion, there are no "bad" models; some are simply less suited for their intended purpose. For example, the least useful model has a scale of 1:1, which is to say that a 100 percent accurate description of the system is identical (or, in "system analysis speak," isomorphic) to the system itself, and thus is not considered a model. So, for any system of significant size, such as one that is larger than can be held completely in mind, it is necessary to sacrifice some accuracy in favor of general understanding. Further, since the human mind has a finite capacity to capture and retain information, in order to reason about a complex topic, it is often necessary to break it down into smaller sections that can be solved and re-synthesized back into the whole.

There are many ways to create models (as will be explored in Part II of this series). Selecting the correct modeling form is very like choosing the form for telling a story. Should the subject be presented as a lyrical poem or a multi-volume novel? Each form has its advantages and drawbacks, and selecting the correct model depends on the experience and education of both the modeler and the audience. Each modeler must carefully consider the intended audience's level of sophistication when choosing a model form and presentation mechanism.

Since there are many ways to represent a system, and since a model is used for the purpose of communication, the model should not be built until the modeler understands the intended audience. Building a model without this knowledge would likely lead to an ineffective model and
miscommunication. To communicate ideas via a model, it is useful to consider how information flows between people. Figure 2 shows two models, taken from the works by Lasswell and Satir listed in the References below. Several theories have been developed to explain the process by which information flows from one person to another; each posits the need for a "communication channel" through which the information flows. Often this occurs through spoken language, as is the case in a lecture hall or a meeting room. Visual communication is also used to convey information, as is the case with paintings and advertisements. In all cases, there are two key aspects to understanding: the generation of a message, and the interpretation of that message.

Figure 2: Two models for communication flows between people (Lasswell and Satir)

As was mentioned above, individuals perceive the world through the use of mental models. However, no two people share the exact same experiences, so no two mental models are alike. This can lead to errors of interpretation based on differences in perceptions. For example, consider Figure 3, from the work by Meyhew listed under References below. What is in the box? Depending on which context you choose (vertical or horizontal), the answer may be the letter B or the number 13. In point of fact, the contents of the box are simply a straight line next to a line with two curves. It is the mental model of letters and numbers that we have constructed that influences our interpretation of the symbology. To someone who cannot read or write (or uses different symbology for those operations), these figures would be meaningless, and this example would represent a very poor model indeed.
So the goal of modeling is to match the description of a complex phenomenon or system to the expectations of the audience that will receive that description. Moreover, we wish to create a representation that faithfully exhibits specific aspects of the system in way that furthers the audience's understanding.

**Systematic approach: The case of software development**

Most models are created to describe complex systems. This is particularly true in the software development field. By its very nature, a computer system is composed of a myriad of interacting parts that all combine to produce visible behavior.

I have found three effective approaches to the analysis of complex systems, including both software systems and other types of systems I've engaged through scientific study: top-down, bottom-up, and middle-out. Top-down analysis starts with a high-level description of the entire domain before moving down to the details. Bottom-up analysis moves in the opposite direction by starting with the concrete details of the system and moving to a more abstract or functional description. In contrast to each of these approaches, middle-out is often useful for dealing with existing systems that are poorly documented. Here, the trick is to pick some point to start (e.g., a section of code), and drill down to the details while simultaneously discovering the functional system description.

Imagine that you are presented with a poorly documented legacy software system. You have access to the user interface, so you could begin investigation of the user functionality by inspection of the interface behavior, perhaps with a subject matter expert as a guide. This would represent a top-down approach. Perhaps you wish to better understand the data model before moving into the business logic that manipulates that data. This would be a bottom-up approach. Or, you could start with one section of the code base and determine the screens that it supports as well as the data that is manipulated. This would be a middle-out approach.
Another avenue to investigation of a complex system is to use classical systematic descriptions (systems analysis), based on boundary, interface, component, and dependency, as shown in Figure 4, from the work by Athey listed under References below. Boundaries are represented by the differentiation between what is inside and what is outside the system. The Interface represents the communication across the boundary between the system inside and the outside. Components are the elements of the system that interact to perform behavior. Dependencies are the relationships between components in complex systems.

![Figure 4: Classical system analysis approach (Athey)](image)

The technique of describing a system via systems analysis works well with the three other directed approaches, since the modeler can begin by looking at the system boundary or by working outward from the system components. As illustrated in Figure 4, icons can be used to represent the various system elements such as the interface (which is exposed externally to the system), the controller (which provides for the system logic), and the entities (which are the manipulated system elements) as the system details are captured and elaborated. These systems elements are grouped by components to show cohesive behavior and connected by dependency relationships. (For additional information, see the works of Athey and Jacobson in the references list below.) Systems analysis is often used to investigate and model existing real-world systems into software programs, but it is useful when considering any system (e.g., experimental investigations).

A simple example of this approach could be the analysis of a door lock. The boundary is the wall, the interface is the door, the controller is represented by the keyhole and tumblers, and the entity is the deadbolt. The component is the full locking mechanism, which has a structural dependency on the door and door frame.

**Observation vs. experimentation**

There are two principal ways to discover information about a system: by
direct observation, or by experimentation -- both of which have their place in the art of software development. Observation is used when you want to gather information without disruption to the system under study. For example, cultural anthropologists often attempt to have minimal impact on the culture they are studying, since direct involvement between observers and their subjects tends to distort the data they collect.

Biological studies, on the other hand, often require the use of direct manipulation of the organism under study. For example, to determine the genetics of yeast, the geneticist will interbreed different genetic strains and study the outward appearance and behavior of resulting cells. In this way the geneticist can infer mutation points, genetic mechanisms, and other details of the mechanics of yeast biology.4

In software development, observation is frequently used as a mechanism for determining requirements, whereas experimentation is the basis for many design decisions.

**Conceptualization and capture**

Once a model form and investigation mechanism has been selected, the next step is to capture the information from that investigation into a usable form. Here are four general techniques for creating a conceptual model that can be formally captured into a physical model for software development.5

**Technique 1: Abstract/specialize**

The first and most useful technique in the modeler's toolkit is the ability to find commonalities and differences among the different data elements that comprise the system under development. The term *abstraction* is used to indicate the modeler's attempt to denote common characteristics between two or more system elements, with the goal of reducing redundancy and providing for a common description. The direct opposite of this is to *specialize*, or find the distinctions between two system elements to highlight the unique characteristics of each.

Figure 5 illustrates the idea of grouping for a defined abstraction. Here a group of shapes can be sorted into a variety of categories based on their characteristics: color, shape, and size. For example, if this set of shapes is intended for use in a visual recognition study of colors, then the first model is more appropriate than the other two. Alternatively, this set of shapes may be intended to illustrate the principles of congruent geometry, in which case the second grouping is more useful. Finally, the intent may be to illustrate effects of volume changes, in which case the third grouping may be most appropriate.
There is no "right" or "wrong" way to group this set of shapes; the correct "model" is dictated by the intended use. A real world example can be found in human anatomy books, which show each of the human body systems (e.g., circulatory, skeletal, nervous, lymphatic, endocrine) in isolation from the others. Thus, the skeletal grouping is based on bones and the concept of structural support, and the circulatory grouping is based on artery/veins/capillaries and the concept of blood flow.

Figure 6 illustrates a dependency map for groups of related elements. In this example, some of the components of a house are dependent on the construction of previous components. The walls and floors are dependent on a foundation, and all the rest of the house components are dependent on the creation of walls. Finally, painting can occur only after the creation of a roof to protect rooms from the weather. This example also shows the use of icon-based symbology to represent the various house components and the concept of obscuring detail, as there is no indication of what comprises each of these components (e.g., is the roof slate? Tile? Wood?).
Technique 2: Fragment/synthesize

The second technique uses the well-known strategy of "divide and conquer." By fragmenting the problem under study into component parts, each piece of the problem can be investigated independently prior to synthesis back into the full model. Someone creating a software program would not attempt to solve the entire problem at once, but would instead create smaller program elements (e.g., objects) that would perform relatively simple but well-defined tasks. These elements can then be joined together to create robust functionality. As shown in Figure 6, the elements of our simplified house have been broken up into a series of components that can then be considered independently of the others.

Technique 3: Obscure/emphasize
With this technique, the goal is to create a model that will selectively draw attention either toward or away from selected system elements. By using the principle of *obscuring* or hiding elements, the modeler insulates the viewer from non-essential elements. By *emphasizing* other elements, the modeler can draw the viewer's attention to exactly the system elements that the modeler is interested in discussing. This technique can often be seen in artistic presentations or heard in musical compositions. In music, for example, one instrument may have emphasis because it carries the melody, while the other instruments carry the rhythm or underlying musical structure (e.g., the Baroque idea of *continuoso*). In software development models, this principle is at work when a class diagram omits the attributes and operation of every object in the system and shows only the attributes and operations relevant to the current activity. As in Figure 6, the details of each component are obscured so that the dependencies between components can be emphasized.

**Technique 4: Chaining**

In the creation of models it is often necessary to segment the model into a collection of specialized views. This is a direct result of the application of the other three techniques; the system under study is, for example, fragmented, abstracted, and emphasized. In order to connect the various views or to lead the model reviewer from one section of the model to other logically connected sections, it is necessary to *chain* elements of the model. This kind of logical connection is frequently seen in software development models, since software is a highly complex system with many interacting parts. One form of this kind of linkage is shown in Figure 7, which depicts the creation of a trade ticket (shown in activity diagram form) that requires the selection of an account and a security. For both of these activities, the model shows an activity that is stereotyped to «connector»; this represents the linkage to another diagram that describes the steps for the selection of an account and security, respectively. In fact, the diagram suggests that there are at least three types of security that can be selected: an equity stock, bond, or mutual fund.
By creating chaining elements in the model, it is possible to lead the viewer to exactly the area of the model that you desire -- a form of emphasis. Chaining also allows for the indication of parts of a system that are dependent on other parts, a key feature of any system analysis.

**Conclusion**

Arguably, anyone can create a model; but creating a truly effective model is hard work. As mentioned previously, we continuously create models of the world we experience. Our difficult goal is to communicate those models to others in a way that enhances their understanding.

The techniques and frameworks described in this article are one path toward achieving this goal. In the next article we will delve deeper into the techniques of model capture, including examples of the different forms models take, and the types of reasoning that can be employed when creating a physical model from a mental one. Finally, in the third and final article for this series, we will look at visual model presentation and discuss concepts of color, balance, composition, and form.

**References**


G. Miller, "The magical number seven plus or minus two: some limits on our capacity for processing information." *Psychological Review* 63 (1956): 81-97.


---

**Notes**


3 G. Miller, "The magical number seven plus or minus two: some limits on our capacity for processing information." *Psychological Review* 63 (1956): 81-97.

4 It is interesting to note that this is a form of *indirect reasoning*, because the conclusion is reached by inferring the mechanism from the experimental result. We will investigate various reasoning methods in the second article of this series.

5 Please do not confuse conceptual and physical models used in this context with the Rational Unified Process definition, based on the 4+1 architectural views proposed by P. Krutchten. Here the term is meant to express differences between a mental model and the constructed physical model.

---

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

Part 1: Introducing RUP SE Version 2.0

by Murray Cantor
Principal Engineer
Rational Brand Services
IBM Software Group

Editor's Note:
The IBM Rational Unified Process,® or RUP,® offers software development organizations a framework for streamlining all activities related to the development lifecycle. Since its formal debut in 1996, RUP has evolved to support a variety of development requirements, including "systems engineering," or SE. In 2001, the first RUP Plug-In to support systems engineering was proposed by Rational Software's strategic services organization. The RUP SE v1 Plug-In was made generally available in 2002, and it continues to be supported by the newly created IBM Rational brand services team.

Outlining a vision for the next generation of RUP SE, Murray Cantor begins a series of three articles this month; we will continue this series through the September and October issues of The Rational Edge. While these articles are consistent with the current RUP SE Plug-In, they introduce a few extensions to the process framework. Please note that the currently available RUP SE Plug-In is RUP SE v1, downloadable from Rational Developer Network (authorization required).

Introduction

This article provides an overview of the latest evolution of Rational Unified
Process for Systems Engineering,® or RUP SE.® RUP users should note that the currently available RUP Plug-In for SE is the RUP SE v1 Plug-In, which was made available in 2002.

RUP SE is an application of the Rational Unified Process, or RUP, software engineering process framework. RUP SE supports the development of large-scale systems composed of software, hardware, workers, and information components. RUP SE includes an architecture model framework that enables the consideration of a set of different perspectives (logical, physical, information, etc.) in order to deliver a solution that addresses the concerns of various development stakeholders. A distinguishing characteristic of RUP SE is that the requirements for these different sorts of components are jointly derived in increasing specificity from the overall system requirements.

RUP SE addresses projects that:

- Are large enough to require multiple teams with concurrent development.
- Have concurrent hardware and software development.
- Have architecturally significant deployment issues.
  OR
- Include a redesign of the underlying information technology infrastructure to support evolving business processes.

RUP SE is delivered as a RUP Plug-In. This article contains a few concepts that, at this writing, have not yet been included in the Plug-In. Nevertheless, these concepts are presented here in order to provide our latest and best understanding of how to meet the needs of systems development.

We will begin with a discussion of systems and the challenges facing the modern systems developer. This discussion is followed by the design points of RUP SE -- that is, how it addresses the challenges of systems development. The next two sections introduce the RUP SE UML-based modeling and requirement specification techniques as well as the use of Unified Modeling Language (UML) semantics. The first section, System Specification, provides a complete blackbox description of the system. Part II, to be published next month, will focus on system architecture and introduce the RUP SE architecture framework, which describes the internals of the system from multiple viewpoints. Part III, in October, will cover requirements analysis and flowdown, an introduction to the method for deriving requirements and specification for the elements of the RUP SE framework. This will include a description of the Joint Realization Method, a novel technique for jointly deriving specification of architectural elements across multiple viewpoints. Part III will also include a discussion of RUP SE programmatics.

**Terminology and concepts in systems development**
By a system, we mean a set of resources that provide services that are used by an enterprise to carry out a business purpose or mission. System components typically consist of hardware, software, data, and workers. Systems are specified by the services they provide, along with other non-behavioral requirements such as reliability or cost of ownership. Designing a system consists of specifying components, their attributes, and their relationships.

System is one of those words that have a set of different, if related, meanings that can cause confusion when used to discuss technical development. Generally, a system is a set or assemblage of elements that exhibit behavior collectively. All of the definitions found in the systems engineering literature build on this idea. In addition to the definition of system we mentioned above, here are some additional definitions of systems and systems engineering:

From the International Council on Systems Engineering

Systems Engineering is an interdisciplinary approach and means to enable the realization of successful systems. It focuses on defining customer needs and required functionality early in the development cycle, documenting requirements, then proceeding with design synthesis and system validation while considering the complete problem:

- Operations
- Performance
- Test
- Manufacturing
- Cost & Schedule
- Training & Support
- Disposal

From Mil-STD-499A

Engineering Management -- The management of the engineering and technical effort required to transform a military requirement into an operational system. It includes the system engineering required to define the system performance parameters and preferred system configuration to satisfy the requirement, the planning and control of technical program tasks, integration of the engineering specialties, and the management of a totally integrated effort of design engineering, specialty engineering, test engineering, logistics engineering, and production engineering to meet cost, technical performance and schedule objectives.

From The Art of Systems Architecting by Maier and Richtin
systems are collections of different things which together produce results unachievable by the elements alone.

Examples of systems include

- **Information technology** systems consisting mainly of software, computer hardware and peripherals, and workers.
- **Products** such as airplanes, satellites, automobiles, and telephone switches consisting of hardware and software.
- **Enterprises** that provide services or carry out a mission. These enterprises may consist of workers enabled by hardware and software. Often these enterprises cross organizational boundaries.

All of these definitions suggest that systems can be viewed from two different perspectives:

- **Blackbox** perspective: The system as a whole: the services it provides and the requirements it meets.
- **Whitebox** perspective: The elements or parts that make up the system.

Which of these two perspectives we use in viewing, understanding, and using a system depends on the needs involved. The problem of systems engineering is to design and implement a system that meets the needs of all system stakeholders, including

- **Users** who are concerned with functionality and performance.
- **Owners** who are concerned with cost of deployment and ownership.
- **Investors** who are concerned with competitive advantage in the market or mission space.

RUP SE provides mechanisms and a UML-based model framework to support teams of systems engineers as they determine the blackbox view of the system and specify an optimal whitebox system design that meets all stakeholder needs.

**Modern system challenges**

Systems engineering was identified as a discipline in the mid-twentieth century. In the early days, systems engineers were challenged to design and specify complex, standalone entities that provided unprecedented capabilities. Some early successes of the systems approach include the Athena Rocket, NORAD, and the Apollo program. These days, systems are not only expected to provide the right set of features; they are also expected to meet a new set of challenges.

Of course, early systems engineering projects presented significant
challenges of their own. For example, the launch of the first imaging satellite represented a great technical accomplishment. First, the development team for that project had to analyze what capabilities were required for this unprecedented system to meet user needs. Then, they had to figure out how to use existing technology to image a structure on the ground, store the image long enough to download it to a ground station, and then make it accessible from an analyst's workstation. At that time, it was of little concern that the solution required dedicated resources. Providing the capability itself was enough to meet stakeholder needs.

Today, those designing satellite imaging systems are concerned less with providing the basic capability and more with optimizing the system to meet a broader set of stakeholder needs. For example, in addition to wanting information that the system gathers as soon as possible, analysts want the ability to integrate that information with data from other sources. System owners want to minimize dedicated hardware (and software) use, because they cannot afford to maintain one system per capability, and they want greater reuse of existing resources. Further, they are very concerned with lowering ongoing operation and maintenance costs.

On top of this, it is certain that the mission and enabling technologies for the system will change several times over the program's lifespan. In addition, those who invest in the system want it to evolve in response to these changes at minimal cost and with minimal disruption. They also expect their investment to result in reusable intellectual property.

**Many requirement types**

To meet stakeholder needs, systems engineers need to consider a broad set of requirements. Here is a partial list of considerations:

- **Functionality**: The capability provided to users and other systems for meeting the business need. Functional requirements should include the behavior the system exhibits as it provides the functionality.
- **Usability**: Ease of access to system function.
- **Maintainability**: Ease of discovery, isolation, and removal of defects.
- **Extendibility**: Ease of adding functionality.
- **Scalability**: The ability to support increasing numbers of users, data items, and so forth, as requirements grow over time.
- **Reliability**: The probability of a correct system response, possibly including safety concerns.
- **Performance**: The expected response time of the system to a step in a use case under capacity loads.
- **Capacity**: The expected number of users and data items.
- **Supportability**: The ease of service in the field, including
acceptable down time.

- **Manufacture**
- **Deployment cost**
- **Operational cost**

Depending on circumstances, there might be other system requirements such as logistics support, security, and remote training needs.

Some of these requirements are familiar to software development teams. Some cannot be addressed without hardware, software, and worker considerations, all three of which must be specified concurrently in a systems design discipline.

Another feature of systems development is the possible need to maintain a number of system configurations. For example, one might want to maintain system specifications for products that have common architectures but different hardware and/or software deployments that meet different cost/performance points.

**Rapid change**

Much of the original systems engineering methodology was developed in the early days of the technology revolution (~1955-1980), when today's rate and impact of technological change were not anticipated. The original systems engineering methods focused on ensuring that requirements were carefully specified, then met to the extent possible.

Today, we know that a system can have a useful life of thirty years. Therefore, because no engineer can know exactly what the system will need to do in five years -- let alone thirty -- there is a higher premium placed on systems that can adapt to evolving needs. Today's system designer must also consider that the environment or context in which the system operates will evolve. Over its lifespan, any system fielded today is likely to interact with unanticipated systems. For example, commercial systems will continue to become more integrated, which will place new requirements on legacy systems; new defense systems will come online, placing new requirements on existing systems.

We also know that the enabling technology for any system will change. Over time, existing systems will fail to be competitive or will not be cost effective to maintain. Modern systems development architecture frameworks need to provide a technology-independent means to reason about rehosting or redeploying.

**A larger solution space**

Today's technologies can help designers meet these complex challenges because they provide more ways to approach system design. For example, whereas the designers of the first satellite image system had limited choices and had to be clever about overcoming technical constraints, today's system designers actually have excess processing capacity and
can reallocate subsystem responsibilities in order to optimize the system. So sometimes they need to make difficult choices. This phenomenon is true for a number of domains, such as telecommunications, avionics, information technology, and so on.

**Balancing logical and physical considerations**

There are many ways to conceptualize a system. Two of them are

- **As a physical entity**, governed by the laws of physics and classical disciplines such as mechanical, electrical, and civil engineering.
- **As a large state machine**, governed by the insights of computer science and software engineering.

Some systems, such as rocket engines, are perhaps best thought of as a physical entity. Others, such as most information technology systems, are large state machines. In fact, all systems fit some aspects of both concepts. Rocket engines have software-driven embedded controllers. Information systems are governed by the physical laws that constrain their performance, latency, and reliability.

In the past, project leaders would typically adopt one of these points of view, depending on the kind of system, and manage accordingly. If they thought of the system primarily from a hardware perspective -- as a physical thing -- they viewed the software as a "necessary evil" that enabled the hardware to do its job. If they thought of the system from a software perspective -- the hardware as a hosting mechanism -- they usually treated the hardware as an afterthought.

**Increased software size and complexity**

With the advent of object technology, component frameworks, and software development automation, software development productivity has increased threefold since 1970, and the size and complexity of software applications continues to grow. At the same time, there have been large gains in processing power, computer memory, available data storage, and network bandwidth. These changes, in turn, have led to ever more sophisticated operating environments. The software and systems industries have used all of this improved capability to develop increasingly large and more complex programs. In fact, competitive pressures have led to a tenfold increase in the size of an average software application, as measured by function points. And, presumably, as technology continues to improve, this trend will continue.

**Many disciplines**

In modern systems development, a typical development team consists of workers playing roles such as architects, developers, designer, testers, and others. In both RUP and RUP SE, all of these people work concurrently to evolve their particular artifacts throughout the lifecycle. They workers do not hand off work to each other in serial fashion. Instead, they work together throughout the effort, evolving levels of detail to address their
areas of concern. One of the chief goals -- and challenges -- of an engineering process and an architecture framework is to provide a means for the various development stakeholders to communicate and align their design decisions. Given the complexity of modern system design, the system development team must address a wide variety of concerns:

- Builds and integration
- Business modeling
- Data modeling
- Domain issues
- Hardware development
- Human factors
- Information technology
- Logistics and field support
- Software development
- Overall system specification and design

Different development team members take responsibility for different, but overlapping, sets of concerns. For example, in addition to ensuring the adequacy of the software architecture to meet functional requirements, software architects are generally concerned with

- **Usability:** ease of accessing the system functionality.
- **Maintainability:** ease of isolating and removing defects without introducing others.
- **Extendibility:** ease of adding new functionality to an existing software product.

Whereas systems engineers usually address issues of

- **Availability/reliability:** the likelihood that the system will be available and respond correctly to input.
- **Performance:** responsiveness of the system to some input.
- **Capacity:** the number of items such as users or data records that the system can handle.
- **Scalability:** the ease of increasing capacity.
- **Supportability:** the ease of providing support in the field. Supportability can include installing the system and applying patches.

Other domain-specific systems engineering concerns include security, ease of training, and logistics support.
Meeting the challenges of complexity

RUP SE provides the artifacts for addressing the concerns we have described, and the workflows for evolving their detailed specification.

Although this article series does not address all concerns of all stakeholders, it does describe an architecture framework that provides model viewpoints to enable a separation of concerns at all levels of system specification. The flowdown mechanism we will describe provides a way to maintain model consistency across the viewpoints. Use of these mechanisms enables the team to jointly follow the RUP practice of continual evolution of artifacts throughout the development lifecycle.

RUP SE design points

The systems development problem differs from the software-only development problem in that systems development addresses a broader set of requirements than are normally addressed in software efforts. Even so, it is important to note that almost all software development efforts contain some elements of the systems problem. Examples of software development efforts that present system concerns include Web-based applications, business applications, information technology integrations, and embedded software, as well as defense and intelligence systems.

The important point here is that systems development projects must address the many challenges today’s systems engineers face, as we discussed in the previous section. In order to address these problems, RUP SE embraces the following design points:

- Follow industry (de facto) standard definition of systems.
- Apply the RUP framework to systems development.
- Extend the RUP 4+1 architectural model into the RUP SE model framework, and extend or modify the RUP roles, activities, artifacts, and disciplines to account for new views.
- Employ UML as the modeling language.
- Provide tool assets.
- Maintain all model levels as a program asset.

Let’s explore each of these design points in more detail.

Follow industry (de facto) standard definition of systems

As we noted in the Terminology and concepts section above, systems can be viewed from both blackbox and whitebox perspectives. RUP SE follows this principle. The blackbox perspective is described in the System specification section (see below). The whitebox perspective is described in the System architecture section, included in next month’s installment. Note that the elements described by RUP SE include
Apply the RUP framework to systems development

The RUP lifecycle and disciplines are shown in Figure 1. RUP SE follows the RUP in these ways:

- **Lifecycle**: Focusing on removing risks, RUP SE follows RUP's four phases by leveraging the team's evolving understanding of the project details.

- **Iterations**: RUP SE advocates a series of system builds based on risk identification and mitigation; an iteration will generally include at least one system build. In particular, all of the artifacts, including the detailed project plans, evolve through iterations. A key feature that RUP SE inherits from RUP is a rejection of waterfall development and the use of iterative development.

- **Disciplines**: RUP SE follows the focus areas, or "disciplines" shown in Figure 1, which provide a number of views into the underlying process definition and the effort that will be carried out by the team in developing the system. Although the RUP project team contains systems engineers, there is no separate systems engineering discipline. Rather, systems engineers play one or more RUP roles and participate in one or more RUP disciplines. Note that the disciplines' workflows and activities are modified to address broader system problems. These modifications are described in the following sections.

**Figure 1: The RUP Process Framework (adopted by RUP SE)**

As described below, RUP SE supplements RUP with additional artifacts, along with activities and roles to support the creation of those artifacts. These are described in more detail in the RUP SE Plug-In.
In addition, as a RUP framework application plug-in, RUP SE provides the opportunity to employ these underlying RUP management principles to systems development:

- Results-based management
- Architecture-centric development

**Extend the RUP 4+1 architecture model into the RUP SE model framework**

Architecture frameworks allow developers to reason about different specification and design concerns, and then document the results of that reasoning in a standard and consistent manner. The original RUP is a software development process; its 4 + 1 architecture framework is inadequate to address all of the concerns of systems development, because more stakeholders require more views. We will describe the new framework in Part II of this series, on architecture.

**Employ UML as the modeling language**

The RUP SE model framework uses UML to express the various diagrams that comprise the architecture model framework views. The current version adopts the UML 1.4 semantics, including stereotypes. In the next version, we will move to UML 2.0 semantics, along with the in-process systems engineering profile when adopted.

**Provide tool assets**

To support RUP SE, IBM Rational Software provides a RUP Plug-In that describes the RUP extension in detail, along with IBM Rational Rose® and IBM Rational RequisitePro® tool add-ins. The currently available Plug-In was released in 2002.

**Maintain all model levels as a program asset**

As mentioned earlier, systems lifespan often outlasts the initial requirements and enabling technologies, which leads, over time, to either outdated or otherwise insufficient functionality, or unacceptable cost of ownership. It follows, therefore, that an effective architecture framework should maintain model views at increasing levels of specificity: The top levels establish context and specification; the lower levels establish components and bills of materials. Traceability should be maintained throughout. Maintaining these levels provides the setting for reasoning about the impact of the changes. Changes in mission usually results in changes at the top level in the model that flow to the lower levels. Changes in technology permit either different design trades or different realizations of the current design. RUP SE provides the needed model levels and traceability.
System specification

System specification is the process of designating the blackbox features of the system: its externally visible functionality, what services it provides, and what measures of effectiveness it is expected to meet. In RUP SE, system specification consists of studying how the system is expected to perform in context. That is, the system is considered a participant in a broader enterprise. The system specification follows from an analysis of the enterprise and the role the system plays in enabling the broader enterprise to meet its business purpose or mission. This process is described below.

Enterprises

Note that an enterprise, like a system, is also a set of resources (workers, hardware, software, and data) used to meet a business purpose or fulfill a mission. Like a system, an enterprise has actors -- entities that use or collaborate with the enterprise. In fact, it is often very useful to consider an enterprise as a system of systems.\footnote{9}

Even a product such as an automobile or an airplane is a part of a broader enterprise. The airplane must interact with the pilot and all of the air traffic control systems.

Context analysis

The system being developed is always part of a larger enterprise. To specify the system, one needs to understand the activities of the broader enterprise, partition that enterprise into the system and other entities, analyze how the system participates in helping the enterprise provide its services, and then capture the results of that analysis as the system specification. In this section, we explore this workflow in more detail.

Note that systems have characteristics of both physical and logical entities. They are logical entities in that they provide services, pass messages, and interact with other logical entities; systems are physical in that they have finite limits on their capabilities, and these limits must be considered. Examples of these limitations are responsiveness and capacity. Currently, the UML has no semantics for such an entity. As discussed below, in RUP SE we model systems as stereotyped classifiers but add the relevant physical characteristics as class attributes.

A system specification captures a blackbox description of the system. It establishes the scope and boundaries of the system, the services it provides, its other attributes, and the things it exchanges with other entities. This information is captured in a system context diagram\footnote{10} as shown in Figure 2.
As shown in Figure 2, a context diagram consists of

- System classifiers
- System actors (external systems and users)
- System actor relations
- Input/output entities
- Input/output entity relations
- Design constraints

Let's explore each of these elements.

**System classifiers**

Systems are entities. A set of systems, each of which meets the same specification, is an instance of a system class. For example, the Saab '93 automobile forms a class and the specific Saab '93 with VIN 12345678 is an instance or object in that class. Hence, systems may be described by UML object and class semantics. Even though in some cases, such as IT systems, it is expected that there will be only one instance of the system, these semantics are still useful.

Following the UML semantics, system objects may have three types of attributes:

- **Class attributes** that have the same value for every system object -- for example, total fuel capacity in the Saab '93 or total number of simultaneous users in an IT system.

- **Instance attributes** whose value can vary between system objects.
-- for example, current fuel load in the VIN 123456789 or current number of users in an IT system.

- **Measures of merit** that are more general design goals such as mean time to failure or technical risk of development. These measures can be captured through tagged values.

Note that the instance attributes provide parameters for test-case development and simulations. The class attributes provide ranges for the test cases and simulations, whereas the measures of merit are used to determine adequacy of the system design and set the decision criteria for design trades.

System class operations are called **system services**. In UML terms, these are operations. Recall that, in UML, operations are classes, and instances of these classes are messages found in sequence diagrams or interaction diagrams that are part of the use-case realization. In systems development, they are found by studying how the system interacts with its actors to meet the needs or mission of the enterprise.

In addition, following the usual UML semantics, services may be aggregated into interfaces.

*System actors*

Recall that in UML, actors are entities that interact with the system, typically users or other systems. Many systems developers have found it useful to include environment elements such as time or atmospheric conditions as kinds of actors.

There are generally two kinds of system actors:

- **Enterprise actors** are external to the enterprise that interact directly with the system.
- **Internal actors** are part of the enterprise, not the system. These actors are usually business workers or other enterprise systems.

In addition, system actors should not be confused with system workers. Workers that are part of the system are not actors. As we will see below, they may be actors to some of the system subsystems.

*System actor relations*

Actor and system collaborations are similar to UML1.x system and subsystem collaborations: The actor instances invoke system services to fulfill their role in the collaboration, and vice versa. It follows that the semantics specifying that an actor uses a system is a UML dependency. The directionality of the dependency denotes whether the system invokes the actor services, vice versa, or both.

Sometimes the actor and the system may be more tightly coupled so that the attributes of one are affected by the attributes of the other, and the
relationship of the actor and the system is promoted to an association. For example, consider a driver (the actor), and an automobile (the system). When the automobile is moving, presumably the driver is in a higher state of readiness than when the automobile is parked.

The system services can be aggregated into interfaces. The interfaces can be used to specify which set of operations is used by which actors or to simply create categories of services. Note that a given service may be included in more than one interface.

**Input/output entities**

Input/output (I/O) entities include anything generated and/or received by the system, such as information or physical items. A retail system may provide credit card information to another credit card information system. An air conditioning system may exchange hot and cold air with the atmosphere. Typical input entities include database queries, file updates, sensor inputs, and control inputs. Typical output entities include query results and sensor outputs.

I/O entities are UML classes with attributes but no operations.

**Input/output entity relations**

Note, as shown in Figure 2, that I/O entities have special stereotyped actor relationships. The association captures whether the entity is sent or received by the actor. The semantics maintain the actor-system relationship in the model and do not suggest that a context diagram is also a data flow diagram.

**Design constraints**

Frequently, system specifications are not strictly blackbox considerations. The specifications include constraints on the internals of the system. Examples of such design constraints include components that must be used and algorithms that must be followed. As the design is evolved and the whitebox elements are specified, these constraints will find a natural expression in one of the RUP SE views described in the following section. There are no specific UML semantics for expressing design constraints. In RUP SE, these may be captured in one of two ways: either as notes in the context diagram or, preferably, as a supplementary requirements document associated with the system class.

**System use cases and services**

RUP SE uses both use cases and services ¹¹ to capture system behavior. In both cases, standard UML semantics ¹² are used:

- **Use cases** describe how the system is used by its actors -- in other words, the set of anticipated collaborations/scenarios between the actors and the system. Use cases convey how the system is used to meet the broader enterprise business purpose or mission. In RUP
SE, Use-Case Descriptions are identical to those used in the standard RUP. In particular, they only describe blackbox interactions between the system and its actors.

- **Services** are the (possibly abstract) operations provided by the system so it can fulfill its role in the use-case scenarios. Instances of services are the messages found in the UML sequence diagrams that realize the use-case scenarios. Generally, there is an n-to-m mapping between use-case and services.

Figure 3 is an example of a partial context diagram for a retail system.

![Figure 3: A retail system context diagram](Click to enlarge)

**Next month:**

This series of articles on RUP SE will continue in the September issue of *The Rational Edge*, with a discussion of System Architecture as it pertains to systems development. The third and final installment will appear in the October issue, with a discussion of requirements analysis -- including flowdown of functional and supplementary requirements to architectural elements -- project organization, iterative development issues, and integration.
Notes


2 For example, the *Oxford English Dictionary* lists eleven definitions.

3 [http://www.incose.org/whatis.html](http://www.incose.org/whatis.html)


8 *Ibid*.


10 The RUP SE context diagram is semantically equivalent to the Elaborated Context Diagram first described by Sanford Friedenthal Howard Lykins, and Abe Meilich. See [http://www.omg.org/cgi-bin/doc?syseng/2001-09-05](http://www.omg.org/cgi-bin/doc?syseng/2001-09-05)

12 This distinction between use cases and services was is a feature of OOSEM, *loc cit*.


---

For more information on the products or services discussed in this article, please click [here](http://www.therationaledge.com) and follow the instructions provided. Thank you!
In a way, a project is like a story. Each project is different, depending on the circumstances, or the project's environment and state. Like a story, each project has objectives, and it defines actions to achieve those objectives. The results of these actions determine how the project will end. And just as the ending of a story affects readers, the execution and outcome of a project affect its stakeholders.

Project planning is related to all of these elements. A plan is influenced by the project's circumstances, objectives, and desired effect on its stakeholders. A plan defines a sequence of actions that leads to results. And if these actions are well planned, the project can be like a good story with a happy ending.

What is project planning?

Planning is a difficult, creative activity that involves juggling with many variables until the circumstances, objectives, and actions are in equilibrium. The result of planning is the Project Plan, which should satisfy the constraints imposed on the project. Because there are complex interdependencies among all the variables, you cannot just sit down and write a Project Plan. First, you must define an initial plan that satisfies one or two major constraints. Then you must refine this initial plan iteratively, until it satisfies all constraints. As Dwight Eisenhower put it, "The plan is nothing. The planning is everything." In other words, the final values of
the planning variables are less important than the understanding of the relationships among these variables that develops during planning.

Much of the material and many of the theories about project planning treat the activity as a recipe: Just follow the steps in the correct order, and you'll automatically produce a good plan. Nothing could be farther from the truth. Planning is an inherently iterative activity; often you must repeat steps and perform them in parallel. Typical instructions give project managers very little practical guidance regarding the instantiation of a Project Plan. This article attempts to fill that gap, by providing a set of proven best practices for planning -- with a focus on concurrent engineering projects.

Even within that focus, we will not attempt to cover all dimensions of project planning, such as:

**Management planning.** In order to successfully execute a project, you need to plan for effective management, including project organization, communication, and quality assurance.

**Estimation.** Planning cannot begin without an understanding of the required effort and project schedule. These parameters are the result of estimation, which necessarily precedes planning.

**Negotiation.** Unreasonable effort and schedule estimates cannot be resolved by any amount of planning. To arrive at realistic constraints, you need to negotiate terms that are satisfactory to all stakeholders involved in the project.

These concerns are beyond the scope of this article. We have intentionally chosen to focus only on planning work because, in our experience, this is the most challenging part of project planning.

**A practical application for RUP**

In this article we use an imaginary software development project to explain project planning techniques. This project uses an iterative approach, following the project lifecycle defined by IBM Rational Unified Process, or RUP. However, the techniques described in this article are not restricted to software development projects or RUP projects. The approach is suitable for all kinds of projects, as long as the project manager defines clear objectives, derives effort and schedule for each phase, and produces product and work breakdown structures.

We have read other articles on the subject of project planning with great interest, specifically "Planning an Iterative Project" by Philippe Kruchten and "Planning a Project with the Rational Unified Process" by Dave West. Both articles describe essentially the same approach to iterative project planning, although from different perspectives. It is not our intention to depart from this approach, but rather to illustrate a practical application of the planning framework these articles describe.
Definitions

Throughout this article, we will use the terms and definitions shown in Table 1.

Table 1: Basic planning terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective</td>
<td>A desired condition at some future point in time. Objectives should be specific and measurable; they should <em>not</em> refer to activities or artifacts.</td>
</tr>
<tr>
<td>Effort</td>
<td>The labor required to deliver a product of a given size. Effort is measured in person-hours, person-days, or person-months.</td>
</tr>
<tr>
<td>Schedule</td>
<td>The timeframe required to deliver a software product. Schedule is measured in days, weeks, months, or years.</td>
</tr>
<tr>
<td>Estimation</td>
<td>The process[^4] of determining the amount of effort and schedule required to deliver a product.</td>
</tr>
<tr>
<td>Role</td>
<td>A definition of the expected behavior and responsibilities of an individual or team, within the context of a project organization.</td>
</tr>
<tr>
<td>Activity</td>
<td>A unit of work to be performed by a role.</td>
</tr>
<tr>
<td>Resource</td>
<td>Things required for project execution that have limited availability, such as personnel, hardware, software, training, licenses, and so on.</td>
</tr>
<tr>
<td>Artifact</td>
<td>The tangible result of activities performed by the project team. An artifact can be a document, diagram, or model.</td>
</tr>
<tr>
<td>Deliverable</td>
<td>An output resulting from activities performed by the project team that has a value, material or otherwise, to a customer or other stakeholder.</td>
</tr>
<tr>
<td>Phase</td>
<td>The time between two major project milestones, during which a well-defined set of objectives is met and artifacts are completed.</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Iteration</td>
<td>A time-boxed period during which one or more teams perform a set of related activities to produce a coherent and (partially) complete product.</td>
</tr>
<tr>
<td>Planning</td>
<td>The process of finding a set of variables (objectives, artifacts, resources, activities, effort, schedule, etc.) that satisfy all constraints imposed upon a project.</td>
</tr>
<tr>
<td>Project lifecycle</td>
<td>A sequence of defined stages over the full duration of a project.</td>
</tr>
</tbody>
</table>

**Simplifications**

To describe the complex process of planning in a simple and understandable manner, we will use the following conventions:

- Schedule is always measured in weeks.
- Effort is always measured in person-hours.
- Effort is always rounded up or down to reflect the level of uncertainty.  
- Phases and iterations consist of whole weeks (starting on Monday and ending on Friday).

**A multi-level approach**

This article consists of two main sections:

1. Project planning
2. Iteration planning

One best practice for project planning is multi-level planning, or developing plans at different levels of detail. These levels are necessary because not all stakeholders are interested in the same information. Executives and customers are interested in high-level planning information, such as the overall budget and schedule. The project manager and project team both need very detailed information in order to execute and manage the project. Furthermore, different levels of detail
are also very useful for effective project planning. The combination of high-level (top-down) and detailed (bottom-up) planning is key to iteratively refining the Project Plan.

**Project planning**

The result of project planning is a coarse-grained plan, or roadmap, for the project that defines boundaries regarding objectives, budget, schedule, and activities. A good high-level Project Plan will remain roughly intact no matter how the iterations are planned, as long as the detailed iteration plans fit within the Project Plan boundaries. You can develop a Project Plan by performing a set of basic activities:

1. Define project objectives
2. Define phases
3. Partition project effort and schedule across phases
4. Prioritize risks
5. Define milestones
6. Define iterations
7. Partition project effort and schedule across iterations

Remember that, in general, you must perform these activities repeatedly, and not necessarily in the sequence shown here.

**Define project objectives**

Before we can write a plan, we must exactly know what we want the project to achieve. We can derive project objectives from the project's business case: The part of the business problem that the project will solve becomes the project objectives. Project objectives describe what is to be done, and not why. The motivation for achieving these objectives should be documented in the project's business case.

Project objectives should be specific, measurable, and realistic. They should start with a verb, as follows:

- Develop a Web-based system for handling customer complaints in order to...
- Roll out the system for all companies...
- Enable a 10 percent decrease in turnaround time for...
- Achieve a 10 percent decrease in turnaround time for...

Project objectives alone are not enough to measure the success of a project. You need to define project acceptance criteria that ensure all stakeholders will be satisfied once you achieve the project objectives. These criteria will also prevent you from forgetting something important.
In fact, the project acceptance criteria will also help stakeholders determine whether the project objectives have been achieved.

Always formulate project acceptance criteria as closed (yes/no) questions:

- Has the system been delivered to the customer?
- Is the system operational?
- Have all the agreed upon deliverables been delivered?
- Have all the agreed upon deliverables passed the customer's internal quality reviews?
- Has the project been completed within the agreed upon schedule and budget?
- Can end users use the system effectively?
- Is the support department able to provide support for the system?
- Is the historical data accessible and accurate?
- Can the previous system be switched off without impacting operations?

Define phases

In this activity you describe the project's lifecycle. Phases provide project stakeholders with the opportunity to make significant decisions about committing resources to the project, one step at a time. Each phase must be concluded with a formal milestone decision -- the point at which stakeholders must decide to commit resources to achieve the objectives of the next phase (not the entire project!). The last project milestone is the point at which stakeholders decide to formally accept ownership of the project's results.

You should document the project lifecycle by splitting the project into sequential phases. Name each phase, using a name that represents the phase's focus, which you can refine by defining phase objectives.

In this article we will use the phases described by the RUP project lifecycle. This lifecycle model suits software development projects specifically but is also very useful for other complex creative projects.

Table 2 highlights the focus for each phase defined in RUP.

**Table 2: RUP phases and objectives**
<table>
<thead>
<tr>
<th>Phase</th>
<th>Objectives</th>
</tr>
</thead>
</table>
| **Inception** | ● Gain stakeholder agreement on scope and boundary conditions.  
● Economically motivate the project.  
● Ensure mutual understanding of critical functionality.  
● Produce overall estimates of cost and schedule.  
● Gain insight into risks that may threaten project success. |
| **Elaboration** | ● Demonstrate that the system architecture will satisfy the significant constraints imposed on the product.  
● Achieve stability of the product vision.  
● Produce a realistic plan for executing the remainder of the project.  
● Eliminate any significant risks that may threaten the success of the project. |
| **Construction** | ● Build and test the system in increments.  
● Ensure increasing understanding and acceptance of the system by stakeholders.  
● Minimize costs and schedule. |
Partition project effort and schedule across phases

In this activity, the project manager allocates the estimated effort and schedule to the four phases. A distinct advantage to using the RUP phases is that RUP provides heuristics for effort and schedule distribution, as shown in Table 3. These heuristics apply not only to software development projects, but also to most complex creative engineering projects. It is important to note that the data in Table 3 serves only as a rough guide and should be tuned to the specific project. For example, if the organization will be using RUP for the first time, the project will require more budget and schedule during Inception to do training and develop understanding among team members.

Table 3: Percentage of effort and schedule across RUP phases

<table>
<thead>
<tr>
<th></th>
<th>Inception</th>
<th>Elaboration</th>
<th>Construction</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Effort</strong></td>
<td>~5%</td>
<td>20%</td>
<td>65%</td>
<td>10%</td>
</tr>
<tr>
<td><strong>Schedule</strong></td>
<td>10%</td>
<td>30%</td>
<td>50%</td>
<td>10%</td>
</tr>
</tbody>
</table>

To complete this activity, you will need the following input:

- Estimate of effort
- Estimate of schedule
- Project start or end date

You can allocate a percentage of the total effort to each phase, as shown in Table 4; round up or down to make partitioning at a lower level (iterations) easier. Then, you can partition the total estimated schedule
across the phases, and determine the start and end dates of each phase accordingly. It is useful to determine the required amount of effort per week (burn rate) for each phase to avoid sharp increases or decreases in staffing requirements. In Table 4, the burn rate for Inception is 150 hours per week, for Elaboration it is 230, for Construction it is 400, and for Transition it is 200. These values indicate acceptable staffing gradations.

Table 4: Allocation of schedule and effort across RUP phases

<table>
<thead>
<tr>
<th>Phase</th>
<th>Start date</th>
<th>End date</th>
<th>Schedule</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inception</td>
<td>2003-01-06</td>
<td>2003-01-31</td>
<td>4 weeks</td>
<td>600 hours (15%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(8%)</td>
<td></td>
</tr>
<tr>
<td>Elaboration</td>
<td>2003-02-03</td>
<td>2003-03-14</td>
<td>6 weeks</td>
<td>1400 hours (21%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(17%)</td>
<td></td>
</tr>
<tr>
<td>Construction</td>
<td>2003-03-17</td>
<td>2003-06-06</td>
<td>12 weeks</td>
<td>4800 hours (43%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(60%)</td>
<td></td>
</tr>
<tr>
<td>Transition</td>
<td>2003-06-09</td>
<td>2003-07-18</td>
<td>6 weeks</td>
<td>1200 hours (21%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(15%)</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>28 weeks</td>
<td>8000 hours</td>
</tr>
</tbody>
</table>

Prioritize risks

Proactive risk elimination is an undisputed best practice of modern project management. You can achieve this by planning to eliminate the highest priority risks in specific phases or iterations.

Once you have identified all the significant risks, gauge the impact and probability of each one, and then sort the resulting list in descending order of severity, using the grid in Figure 1.
This grid is useful because managers often underestimate the influence of probability on severity. But it should serve only as a guideline; you must take your risk sorting a step further by seeking the opinions of a few key project team members.

Once you have prioritized the risks, it should become obvious in which phase you should target each risk for elimination. Although the highest priority risks should be tackled first, you can usually eliminate risks relating to the project environment -- such as those affecting project objectives or feasibility -- during the Inception phase. Technical risks, such as new development tools, multiple platforms, or multiple systems, are often eliminated during the Elaboration phase. You can usually eliminate risks that might interfere with the rapid production of quality software, such as scope creep or delayed integration and testing, during the Construction phase. Target risks relating to deployment and commissioning, such as legacy data conversion or user skills, during the Transition phase.

This list of prioritized risks is an important input for the next step: defining project milestones. Before proceeding, however, it may be necessary to refine the phase objectives based upon the risks.

**Define milestones**

After you define the objectives, budget, and schedule for each phase and prioritize risks, then you can identify project milestones and describe exit criteria for each one. A milestone marks an important point in time at which you can assess a particular artifact, synchronize activities, or deliver a product. Milestones that mark the end of a phase (referred to as *major milestones*) also serve as important decision-making moments. At these milestones, stakeholders assess the results of the preceding phase and give formal approval to proceed with the next phase of the project. In
order to set clear expectations and ensure that each phase is assessed objectively, you must specify exit criteria in advance for each milestone.

If you have done a thorough risk analysis, defining milestone exit criteria for the Inception and Elaboration phases will be easy. If you find yourself struggling to refine the default criteria described in RUP, you should revisit the Prioritize Risks activity.

Exit criteria for the Construction and Transition phases will be more difficult to define at this stage. Usually, the default criteria are a good place to start; you will refine these criteria sometime during Inception or Elaboration. Formulate milestone exit criteria as closed yes/no questions because you will use them to determine whether the project may proceed to the next phase. The results of this activity should look similar to Table 5.

Table 5: Sample exit criteria for RUP milestones

<table>
<thead>
<tr>
<th>Milestone</th>
<th>Exit Criteria</th>
<th>Date</th>
</tr>
</thead>
</table>
| Lifecycle Objectives       | • Do stakeholders feel confident that the project team understands the problem to be solved?  
                              • Do stakeholders feel confident that the major risks have been identified?  
                              • Have the initial estimates been refined based on experiences during Inception?  
                              • Are budget/schedule variations acceptable to the stakeholders?  | 2003-01-31 |
<table>
<thead>
<tr>
<th>Lifecycle Architecture</th>
<th>2003-03-14</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Do stakeholders feel confident that the project team is capable of delivering an appropriate solution within the parameters of the Project Plan?</td>
<td></td>
</tr>
<tr>
<td>- Does the system architecture reflect both functional breadth and critical depth?</td>
<td></td>
</tr>
<tr>
<td>- Have all major risks been eliminated or mitigated?</td>
<td></td>
</tr>
<tr>
<td>- Are budget/schedule variations acceptable to the stakeholders?</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Beta Release</th>
<th>2003-06-06</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Is the product sufficiently complete and of sufficient quality to start production acceptance testing?</td>
<td></td>
</tr>
<tr>
<td>- Have the end-user and support organization been prepared for deployment?</td>
<td></td>
</tr>
<tr>
<td>- Are budget/schedule variations acceptable to the stakeholders?</td>
<td></td>
</tr>
</tbody>
</table>
Be sure not to include the state of a particular artifact or activity in these criteria. Simply producing an artifact or performing an activity is not in itself valuable to the project. Rather, it is what you achieve by producing the artifact or performing the activity that is valuable. Table 6 shows two examples of poor milestone exit criteria, as well as suggestions for improvement.

**Table 6: Examples of poor milestone criteria**

<table>
<thead>
<tr>
<th>Poor Exit Criteria</th>
<th>Suggestion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is the Vision complete?</td>
<td>Do both stakeholders and the project team agree to the problem and solution outlined in the Vision?</td>
</tr>
<tr>
<td>Have the test cases been executed?</td>
<td>Do stakeholders find the quality of the product acceptable?</td>
</tr>
</tbody>
</table>

**Define iterations**

Once we have defined the phases, it is time to refine the Project Plan by defining iterations. The first step is to divide each phase into equal iterations. Iterations of the same phase do not have to be of the same length by definition, but dividing them this way provides a good starting point.

To decide on the number of iterations per phase, you can simply apply the default scheme:

- 1 Inception iteration
- 2 Elaboration iterations
● 2 Construction iterations
● 1 Transition iteration

This scheme works for most small- to medium-sized software development projects. More information on determining the number of iterations can be found in the IBM Rational whitepaper "Planning a Project with the Rational Unified Process." When in doubt, divide Elaboration and Construction into four-week iterations and Inception and Transition into six-week iterations.

After you determine the number of iterations, you need to define the objectives for each one. Iteration objectives are derived from the phase objectives and should therefore be very specific to the project. **Iteration objectives should eliminate risks.** You can present them as shown in Table 7.

**Table 7: Iteration objectives**

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Objectives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inception-1</td>
<td>● Achieve stakeholders’ concurrence on <strong>scope</strong>.</td>
</tr>
<tr>
<td></td>
<td>● Refine <strong>estimates</strong> based on actual experiences during Inception.</td>
</tr>
<tr>
<td></td>
<td>● Identify and prioritize <strong>risks</strong>.</td>
</tr>
<tr>
<td></td>
<td>● Construct a detailed <strong>plan</strong> for the remainder of the project.</td>
</tr>
<tr>
<td>Elaboration-1</td>
<td>● Develop a <strong>skeleton solution</strong> which covers the complex, volatile, and risky parts of the system.</td>
</tr>
<tr>
<td></td>
<td>● Eliminate highest priority technical <strong>risks</strong>.</td>
</tr>
<tr>
<td></td>
<td>● Ensure project team proficiency with <strong>process and tools</strong>.</td>
</tr>
<tr>
<td></td>
<td>● Ensure stability of the development and test <strong>environments</strong>.</td>
</tr>
</tbody>
</table>
| Elaboration-2 | - Develop a **skeleton solution** with broad functional coverage and depth in only the risky parts.  
- Eliminate all known non-trivial **risks**.  
- Construct a detailed **plan** for the remainder of the project. |
| Construction-1 | - Develop **80 percent of the solution** as rapidly as possible with reasonable quality.  
- Minimize **costs** by optimizing schedule and resources. |
| Construction-2 | - Develop **100 percent of the solution**.  
- Achieve **acceptable quality** of all developed portions of the solution.  
- Prepare **deployment** to the end-user and support organization.  
- Minimize **costs** by optimizing schedule and resources. |
| Transition-1 | - Improve the **quality** of the solution.  
- Ensure end-user organization **ability to use** the solution.  
- Ensure support organization **ability to support** the solution.  
- **Close out** the project. |
Do not be surprised if you notice that phase objectives, milestone exit criteria, and iteration objectives seem similar. These planning elements are all directly related to risks, and because we intend to actively attack project risks, this is to be expected.

**Partition phase effort and schedule across iterations**

In order to set boundaries for producing detailed iteration plans, we need to partition the effort and schedule allocated to each phase across the iterations within that phase. For phases containing only one iteration (in RUP probably Inception or Transition) this is easy: Allocate the total effort and schedule for the phase, as in the Inception row of Table 8. For phases with more than one iteration, however, some mathematics is required.

When partitioning schedule, it is usually easiest to divide the phase into iterations of equal length. Iterations of the same phase do not have to be the same length, but this simplifies planning. When the phase cannot be partitioned into equal parts, make the first iterations longer. The team needs more time initially to understand the phase objectives and tends to work faster during succeeding iterations. We mentioned in the previous step that iteration lengths of four to six weeks are reasonable for an average project. Small projects may have iterations of two to three weeks, whereas those for large projects may be anywhere from eight to twelve weeks.

You can partition the phase effort in much the same way. Iterations of equal length within a phase usually require equal effort. Allocate more effort to longer phases and less to shorter ones, so that the total effort per week remains relatively constant within a phase. However, effort and schedule are not always allocated proportionally. For example, an early Transition iteration may be long yet require very little effort, because the product is being reviewed by stakeholders, and only a few team members are required to incorporate feedback.

**Table 8: Phase effort and schedule across iterations**

<table>
<thead>
<tr>
<th>Phase</th>
<th>Iteration</th>
<th>Schedule</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inception</td>
<td>I1</td>
<td>4 weeks</td>
<td>600 hours</td>
</tr>
<tr>
<td>Elaboration</td>
<td>E1</td>
<td>3 weeks</td>
<td>700 hours</td>
</tr>
<tr>
<td></td>
<td>E2</td>
<td>3 weeks</td>
<td>700 hours</td>
</tr>
<tr>
<td>Construction</td>
<td>C1</td>
<td>6 weeks</td>
<td>2400 hours</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>6 weeks</td>
<td>2400 hours</td>
</tr>
<tr>
<td>Transition</td>
<td>T1</td>
<td>6 weeks</td>
<td>1200 hours</td>
</tr>
</tbody>
</table>

As with phases, it is useful to determine the effort per week (burn rate) for each iteration to avoid sharp increases or decreases in staffing requirements. Using the phase effort and schedule allocations in Table 8, you can see that the Construction phase consists of two iterations, each
with a schedule of 6 weeks and an effort of 2400 hours. Division of effort by schedule results in an average of about 400 hours (or 10 people) per week during both Construction iterations.

You now have a Project Plan that you can refine at the end of each iteration. Update phase budget and schedule allocations to reflect actual progress, and refine phase and iteration objectives to reflect previously undetected risks. The number of iterations should not change unless you have significantly under- or over-estimated the project. In that case, it is quite reasonable to believe that you will require a few iterations more or less. At the start of the project, develop a detailed activity plan for the first iteration. Many project managers neglect to do this, believing that the project team already knows what needs to be done. But having a detailed plan in place within the first two weeks of the project allows you to monitor and control the project from day one. You are likely to see deviations from the plan surprisingly early -- within the first weeks of the project -- and you can adjust the overall Project Plan accordingly.

**Iteration planning**

Near the end of each iteration, you should create a plan for the next iteration. The result of iteration planning is a fine-grained plan for a specific iteration. This plan contains the details omitted in the overall Project Plan: artifacts, activities, names, dates, and required effort.

Iteration planning consists of the following basic activities:

1. Refine iteration objectives
2. Develop product breakdown structure
3. Define evaluation criteria
4. Define iteration milestones
5. Partition effort across disciplines
6. Develop work breakdown structure
7. Staff team and assign work
8. Consolidate planning

**Refine iteration objectives**

The overall Project Plan describes objectives for each iteration, but you may need to refine them to reflect the current state of the project. As mentioned above, you should update the Project Plan at the end of each iteration, at around the same time you plan the next iteration. The refined iteration objectives should be documented in both the Project and Iteration Plans. Table 9 provides an example of initial objectives for the first iteration of the Construction phase.

**Table 9: Sample objectives for iteration #1, Construction phase**
Objective Description

Develop alpha release.
Develop 80% of the solution as rapidly as possible with reasonable quality.

Minimize costs.
Keep costs as low as possible by optimizing schedule and resources.

Refine iteration objectives by prioritizing risks and considering open issues. Sort risks in decreasing order of severity, and then target the highest priority risks for elimination. For more information on risk analysis, see the Prioritize risks section above. Table 10 shows examples of iteration objectives derived from targeting specific risks.

Table 10: Sample iteration objectives targeting specific risks

<table>
<thead>
<tr>
<th>Objective</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Achieve tool proficiency.</td>
<td>Ensure project team is proficient with the Xyz development tool.</td>
</tr>
<tr>
<td>Approve change request procedure.</td>
<td>Stakeholders understand and agree to suggested change request procedure.</td>
</tr>
<tr>
<td>Understand corporate user interface standards.</td>
<td>Acceptance testers are familiar with the contents of the corporate user interface standards.</td>
</tr>
</tbody>
</table>

Because every iteration concludes with an objective assessment of the status of the project, it is reasonable to assume that a number of open issues will be carried over between iterations. These are errors or inconsistencies that have been identified during a particular iteration, but have not yet been resolved. In an iteration plan, you must consider these open issues from previous iterations and, if necessary, update the iteration objectives to reflect significant issues.

Develop product breakdown structure

Objectives are met, risks are mitigated and issues are resolved by developing and refining artifacts (demonstration-based approach). Once you determine your iteration objectives, the next step is to define the artifacts you will produce to achieve these objectives. Group the artifacts
by discipline and assign an owner for each artifact. This does not mean that the owner will be the only person involved in producing the artifact, but the owner will be responsible for progress and the quality of the artifact. Table 11 shows an example of a product breakdown structure for a specific iteration.

Table 11: Product breakdown structure (PBS) for a single iteration

<table>
<thead>
<tr>
<th>Artifact Set</th>
<th>Artifact</th>
<th>Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Management</td>
<td>Project Plan</td>
<td>P. Menhir</td>
</tr>
<tr>
<td></td>
<td>Status Assessment</td>
<td>P. Menhir</td>
</tr>
<tr>
<td></td>
<td>Iteration Plan</td>
<td>P. Menhir</td>
</tr>
<tr>
<td>Requirements</td>
<td>Iteration Assessment Vision</td>
<td>P. Menhir</td>
</tr>
<tr>
<td></td>
<td>Use-Case Model</td>
<td>I. Understand</td>
</tr>
<tr>
<td></td>
<td>Use Case – Confirm Payment</td>
<td>L. Speccit</td>
</tr>
<tr>
<td></td>
<td>Use Case – Sell Stock</td>
<td>L. Speccit</td>
</tr>
<tr>
<td></td>
<td>Use Case – Request Material</td>
<td>L. Speccit</td>
</tr>
<tr>
<td>Analysis and Design</td>
<td>Design Model</td>
<td>M. Archie</td>
</tr>
<tr>
<td></td>
<td>Data Model</td>
<td>S. Tructu</td>
</tr>
<tr>
<td></td>
<td>Component - Trading</td>
<td>D. Velepa</td>
</tr>
<tr>
<td></td>
<td>Component - Ticker</td>
<td>O. Gramma</td>
</tr>
<tr>
<td></td>
<td>Component - UserMan</td>
<td>O. Gramma</td>
</tr>
<tr>
<td></td>
<td>Release – Alpha 1</td>
<td>D. Velepa</td>
</tr>
<tr>
<td>Test</td>
<td>Phase Test Plan</td>
<td>W. Kerr</td>
</tr>
<tr>
<td></td>
<td>Test Cases</td>
<td>T. D. Senna</td>
</tr>
<tr>
<td></td>
<td>Test Scripts</td>
<td>T. D. Senna</td>
</tr>
<tr>
<td></td>
<td>Test Evaluation Summary</td>
<td>W. Kerr</td>
</tr>
<tr>
<td>Deployment</td>
<td>Bill of Materials</td>
<td>W. Kerr</td>
</tr>
<tr>
<td></td>
<td>Deployment Plan</td>
<td>W. Kerr</td>
</tr>
</tbody>
</table>
It is optional to supply review dates for each artifact as well. We prefer to document these points in time as (minor) iteration milestones. We then use these milestones to assess the status of a number of artifacts, rather than for tracking each artifact independently.

**Define evaluation criteria**

In order to define the desired end state of the iteration, you must formulate iteration evaluation criteria. From a quality perspective, the purpose of evaluation criteria is to provide a means of measuring the success of an iteration. Within the context of planning, defining evaluation criteria is a form of the iterative project management best practice: *Begin with the end in mind.* The set of evaluation criteria should clearly define the end -- or success criteria -- of the iteration. Defining evaluation criteria (or at least a first cut) at this point in the planning process leads to a more internally consistent iteration plan, which will require less refining.

The phase objectives, milestone exit criteria, iteration objectives, risks, and open issues all have an influence on the evaluation criteria. Many project managers neglect to document iteration evaluation criteria, as they feel it adds little value. Our experience has shown that project teams provided with clear, measurable criteria know exactly what to do right from the start of the iteration. Project teams that have no iteration evaluation criteria to focus on tend to need a week or two to "figure out exactly what to do." As mentioned earlier, you should express iteration evaluation criteria as closed questions. Group the criteria by the appropriate discipline, as shown below.

**Requirements**

- Is the Vision stable?
- Did stakeholders actively participate in reviewing the Vision?
- Is the requirement set complete?
- Are all inconsistencies resolved?
- Are traceability relations and requirements attributes up to date?

**Analysis and design**

- Is the software architecture stable?
- Does the design satisfy the quality ranges defined in the design guidelines?
- Has the design of the targeted use cases been documented and reviewed?
- Have the design guidelines been adhered to?

**Implementation**
● Has a working build been delivered to the test team?
● Has the build-deploy cycle been automated?
● Does the build satisfy the quality ranges defined in the programming guidelines?
● Is there less than 10 percent rework for the upcoming iteration?

Test

● Have the requirements targeted in this iteration been incorporated?
● Have all test cases, as defined in the iteration test plan, been executed?
● Does the build reflect the expected quality?
● Have test results been analyzed and reported to the customer?

Deployment

● Has a draft deployment plan been discussed with the customer?
● Has the alpha release been delivered to the customer?
● Has a bill of materials been delivered to the customer that describes the contents delivered?
● Have the reviewers been walked through the Product Acceptance Plan?

Configuration and change management

● Have the iteration results been baselined?
● Are the reviewers aware of how to submit a change request?
● Have all submitted changes been addressed by the change control board?
● Have all assigned change requests been incorporated?
● Have all level 1 and 2 defects been resolved?

Project management

● Has the Risk List been updated and discussed with the customer?
● Has the Project Plan been updated and delivered to the customer for approval?
● Has the Iteration Plan for the second Construction iteration been delivered to the customer for approval?
● Have the results of the iteration assessment been documented and delivered to the customer?
Define iteration milestones

Milestones signify important points in time during the project and are used to protect the project from delays. Any important decision-making moments or decision input from external parties should be marked as a milestone. Within the project team, milestones mark hand-offs and synchronization points. A hand-off is when a set of artifacts with a particular state is handed over to another team.

The first step when defining milestones is to make a list of all significant events that will occur during the iteration. Consider important dates in the project environment, such as when a particular document will be approved by the board, or when the development environment will be delivered. Try to determine a milestone per discipline, such as the completion of the Vision or the deployment of the system to the acceptance test environment. The last iteration in a phase will always conclude with a major (phase) milestone. After determining in which week each milestone will fall, furnish the milestone with a date (and a detailed description of the exit criteria if necessary). See Table 12. In most cases, the easiest way to do this is to plan backwards, starting from the last day of the iteration.

Table 12: Iteration milestones, dates, and owners

<table>
<thead>
<tr>
<th>Milestone</th>
<th>Type</th>
<th>Date</th>
<th>Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Week 12 – March 17th – March 21st</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Review Use Case Sell Stock</td>
<td>Minor</td>
<td>2003-3-19</td>
<td>I. Understand</td>
</tr>
<tr>
<td>Review Phase Test Plan</td>
<td>Minor</td>
<td>2003-3-19</td>
<td>W. Kerr</td>
</tr>
<tr>
<td>Deliver Use Case Sell Stock</td>
<td>Minor</td>
<td>2003-3-20</td>
<td>L. Specct</td>
</tr>
<tr>
<td><strong>Week 13 – March 24th – March 28th</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Review Use Case Confirm Payment</td>
<td>Minor</td>
<td>2003-3-26</td>
<td>I. Understand</td>
</tr>
<tr>
<td>Deliver Use Case Confirm Payment</td>
<td>Minor</td>
<td>2003-3-27</td>
<td>L. Specct</td>
</tr>
<tr>
<td>Report Status</td>
<td></td>
<td>2003-3-28</td>
<td>P. Menhir</td>
</tr>
<tr>
<td><strong>Week 14 – March 31st – April 4th</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Review Deployment Plan</td>
<td>Minor</td>
<td>2003-4-2</td>
<td>W. Kerr</td>
</tr>
<tr>
<td>Review Use Case Request Material</td>
<td>Minor</td>
<td>2003-4-2</td>
<td>I. Understand</td>
</tr>
<tr>
<td>Deliver Use Case Request Material</td>
<td>Minor</td>
<td>2003-4-3</td>
<td>L. Specct</td>
</tr>
<tr>
<td>Review Test Set</td>
<td>Minor</td>
<td>2003-4-4</td>
<td>W. Kerr</td>
</tr>
<tr>
<td><strong>Week 15 – April 7th – April 11th</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Report Status</td>
<td>Minor</td>
<td>2003-4-11</td>
<td>P. Menhir</td>
</tr>
<tr>
<td>Deliver Development Release</td>
<td>Minor</td>
<td>2003-4-11</td>
<td>D. Velepa</td>
</tr>
<tr>
<td><strong>Week 16 – April 14th – April 18th</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Review Deployment Plan</td>
<td>Minor</td>
<td>2003-4-14</td>
<td>W. Kerr</td>
</tr>
<tr>
<td><strong>Week 17 – April 21st – April 25th</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Review Test Evaluation Summary</td>
<td>Minor</td>
<td>2003-4-23</td>
<td>W. Kerr</td>
</tr>
<tr>
<td>Review Project Plan</td>
<td>Minor</td>
<td>2003-4-22</td>
<td>P. Menhir</td>
</tr>
<tr>
<td>Review Iteration Plan</td>
<td>Minor</td>
<td>2003-4-23</td>
<td>P. Menhir</td>
</tr>
<tr>
<td>Deliver Alpha Release</td>
<td>Minor</td>
<td>2003-4-24</td>
<td>D. Velepa</td>
</tr>
<tr>
<td>Iteration Assessment</td>
<td>Major</td>
<td>2003-4-25</td>
<td>P. Menhir</td>
</tr>
</tbody>
</table>
In Table 12, notice that only the Iteration Assessment milestone is a *major* milestone. All other milestones in this iteration are minor. The milestone type is related to the freedom to change the planned milestone date. The project manager can change minor milestones, but major milestones relate to important decision-making moments and are owned by the customer.

**Partition effort across disciplines**

At this stage, we have determined what needs to be achieved during the iteration, the artifacts that must be produced in order to achieve these objectives, and the important dates during the iteration. We now need to start considering how much effort will be necessary to execute the iteration. We have already determined the effort and schedule for each iteration and documented it in the Project Plan. What we need to do now is partition the effort for the iteration across the disciplines. Each discipline is a group of logically related activities.

Table 13 provides an indication of how much of the total effort of an iteration within each phase should be allocated to the disciplines. Once again, we emphasize that this should serve only as a guideline and may be adjusted according to the specific project.

**Table 13: Estimate of effort by discipline for a single iteration**

<table>
<thead>
<tr>
<th>Discipline</th>
<th>Inception</th>
<th>Elaboration</th>
<th>Construction</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Modeling</td>
<td>10%</td>
<td>5%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Requirements</td>
<td>20%</td>
<td>20%</td>
<td>5%</td>
<td>5%</td>
</tr>
<tr>
<td>Analysis &amp; Design</td>
<td>10%</td>
<td>20%</td>
<td>15%</td>
<td>5%</td>
</tr>
<tr>
<td>Implementation</td>
<td>5%</td>
<td>20%</td>
<td>30%</td>
<td>20%</td>
</tr>
<tr>
<td>Test</td>
<td>5%</td>
<td>10%</td>
<td>20%</td>
<td>35%</td>
</tr>
<tr>
<td>Deployment</td>
<td>5%</td>
<td>5%</td>
<td>10%</td>
<td>20%</td>
</tr>
<tr>
<td>Config. &amp; Change Mgmt.</td>
<td>5%</td>
<td>5%</td>
<td>5%</td>
<td>5%</td>
</tr>
<tr>
<td>Project Management</td>
<td>20%</td>
<td>10%</td>
<td>10%</td>
<td>5%</td>
</tr>
<tr>
<td>Environment</td>
<td>20%</td>
<td>5%</td>
<td>5%</td>
<td>5%</td>
</tr>
</tbody>
</table>

The result of this activity is a top-down estimate of the effort required to perform all the activities for a specific discipline in this iteration. This information is provided to the teams for developing their own (bottom-up)
detailed activity plans. They also need estimates of their budget in order to focus only on doing only what is necessary. Then they can estimate what percentage of their total capability they can afford to devote to activities for each discipline. Table 14 shows one way of doing such an estimate.

### Table 14: Percentage of total effort allocated to each discipline for a single iteration

<table>
<thead>
<tr>
<th>Discipline</th>
<th>Effort (hours)</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>200</td>
<td>9%</td>
</tr>
<tr>
<td>Analysis &amp; Design</td>
<td>350</td>
<td>15%</td>
</tr>
<tr>
<td>Implementation</td>
<td>850</td>
<td>36%</td>
</tr>
<tr>
<td>Test</td>
<td>500</td>
<td>21%</td>
</tr>
<tr>
<td>Deployment</td>
<td>200</td>
<td>9%</td>
</tr>
<tr>
<td>Project Management</td>
<td>200</td>
<td>11%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>2350</strong></td>
<td></td>
</tr>
</tbody>
</table>

Develop a work breakdown structure

The artifacts we defined above, combined with our idea of how much effort we can devote to each discipline, do not give us enough to execute the iteration. We still need to identify the activities we need to perform. It is useful to group these activities by discipline, because then we can take advantage of the partitioning we have just done. The result is a work breakdown structure (WBS), consisting of three levels: iteration, discipline, and activity. RUP itself is a very good source for defining activities, and there are distinct advantages to using a standardized WBS (not discussed here).

For each discipline, consider the iteration objectives, risks, open issues, and artifacts you need to produce, and create a list of activities based on this. For larger projects with different teams, it may make sense to let each team define its own activities. Of course, the project manager should check the results to make sure they are complete.

Do not forget to include reviews and project management activities. Some project managers prefer to define a fourth WBS level, by making each general activity product-specific. For example, you can define the *analyze use cases* activity below in terms of sub-activities: one for each use case that will actually be analyzed during the iteration. The contents of the resulting WBS should look something like Figure 2.
Of course, it is not a requirement to document the WBS graphically. You can use a text editor or spreadsheet, as we will show in the next section.

**Partition effort across activities**

Using the partitioned effort per discipline as a guide, you can now allocate the amount of effort required for each activity. Although you may feel that some knowledge of the black arts is necessary to conjure up this level of detail, in fact, this work breakdown structure is based primarily on common sense. The product breakdown (developed during the step Define deliverables) specifies which artifacts you must work on. The iteration evaluation criteria provide insight into what is required for these artifacts at different points in time, and the iteration milestones indicate when you should deliver them. Typically, you do not perform an entire activity within one week; most are spread out over two or more weeks.

You may notice that the totals per discipline are not exactly the same as the values you estimated earlier. This is not a problem; the top-down partitioned values simply serve as guidelines. However, you will need to resolve large discrepancies between the top-down and the bottom-up
Table 15 is a detailed activity plan, which is the most important result of iteration planning. It replaces the Gantt chart traditionally used for this level of planning. Although it does not capture dependencies between activities, you can still control the schedule effectively because the iteration is of short duration and has clearly defined milestones and evaluation criteria.

### Table 15: Detailed activity plan for a single iteration

<table>
<thead>
<tr>
<th></th>
<th>Hours per week</th>
<th>Totals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td><strong>Week nr.</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Nr</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Activity</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>A Requirements</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AA Manage Dependencies</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>AB Detail Use Cases</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>AC Review Use Cases</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td><strong>B Analysis &amp; Design</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BA Analyze Use Case</td>
<td>20</td>
<td>16</td>
</tr>
<tr>
<td>BB Design Use Case</td>
<td>30</td>
<td>26</td>
</tr>
<tr>
<td>BC Design Class</td>
<td>26</td>
<td>34</td>
</tr>
<tr>
<td>BD Design Database</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>BE Review</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td><strong>C Implementation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CA Implement Components</td>
<td>48</td>
<td>92</td>
</tr>
<tr>
<td>CB Rework</td>
<td>38</td>
<td>4</td>
</tr>
<tr>
<td>CC Deliver Release</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td><strong>D Test</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DA Develop Test Plan C1</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>DB Review Test Plan C1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>DC Develop Test Cases</td>
<td>48</td>
<td>16</td>
</tr>
<tr>
<td>DD Develop Test Scripts</td>
<td>24</td>
<td>56</td>
</tr>
<tr>
<td>DE Review Test Set</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>DF Test Release</td>
<td></td>
<td>80</td>
</tr>
<tr>
<td>DG Analyze Test Results</td>
<td></td>
<td>56</td>
</tr>
<tr>
<td><strong>E Deployment</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EA Develop Deployment Plan</td>
<td>16</td>
<td>36</td>
</tr>
<tr>
<td>EB Review Deployment Plan</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>EC Define Bill Of Materials</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>ED Deliver Alpha Release</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>F Project Management</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FA Refine Project Plan</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>FB Refine Iteration Plan C1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td><strong>Totals</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Hours per week</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Totals</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: The table assumes a single iteration of short duration with clearly defined milestones and evaluation criteria, enabling effective schedule control despite not capturing dependencies.
If you check the total amount of effort per week, you will see that it matches quite satisfactorily with the 400 hours per week we estimated earlier. Of course, you would try to keep the burn rate relatively constant, so when there are significant differences, you may need to shift the milestones slightly.

Staff team and assign work

We have almost completed our detailed iteration planning, except that we do not yet know anything about the size or constitution of the project team. We have calculated that the burn rate for this iteration is approximately 400 hours per week, or a 10-person team working 40 hours per week. Since this is a Construction iteration, some roles are not required full time. We can therefore safely assume that one to three people will be working part-time, which gives us a team size of eleven to thirteen people.

From the activity Partition effort across disciplines, we know how the effort is distributed over the disciplines. This information gives us a clue about the number of people required per discipline. For example, implementation takes about 30 percent of the effort, or about 720 hours in six weeks. This gives us 120 hours per week, or the equivalent of three developers. We can make similar calculation for the requirements, design, and testing roles. The result is an organization breakdown structure, shown in Table 16.

Table 16: Organization breakdown structure (OBS)

<table>
<thead>
<tr>
<th>Quantity</th>
<th>Role</th>
<th>Team Member</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Project Manager</td>
<td>P. Menhir</td>
</tr>
<tr>
<td>1</td>
<td>Architect</td>
<td>M. Archie</td>
</tr>
<tr>
<td>1</td>
<td>System Analyst</td>
<td>I. Understand</td>
</tr>
<tr>
<td>1</td>
<td>Requirements Specifier</td>
<td>L. Speccit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FB</th>
<th>Refine Iteration Plan C1</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>FC</td>
<td>Review Iteration Plan C1</td>
<td>6</td>
</tr>
<tr>
<td>FD</td>
<td>Develop Iteration Plan C2</td>
<td>10</td>
</tr>
<tr>
<td>FE</td>
<td>Assess Status</td>
<td>8</td>
</tr>
<tr>
<td>FF</td>
<td>Assess Iteration</td>
<td>12</td>
</tr>
<tr>
<td>FG</td>
<td>Manage Project</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td><strong>Totals</strong></td>
<td><strong>412</strong></td>
</tr>
</tbody>
</table>
During the previous step (*Partition effort across activities*), we calculated the amount of effort required per activity. The next step is to plan the work for each team member, by determining how much of this effort each person will spend.

There is a simple way of doing this when you do not have fancy planning tools. Simply create a worksheet in a spreadsheet tool to describe the overall activity planning, and then duplicate this worksheet for each team member. In each individual worksheet, you only describe the required effort per activity for that individual person. Then, you can sum the effort across all team members and compare this to the partitioned effort of the overall activity planning.

Last but not least, notice that steps *Partition effort across activities* and *Staff team and assign work* must be performed a number of times in order to make the product breakdown, work breakdown, and organization breakdown structures consistent. The artifacts defined in the product breakdown structure will be produced by project team members defined in the organization breakdown structure, by performing the activities defined in the work breakdown structure.

### Consolidate planning

The purpose of the final step is to verify consistency among objectives, deliverables, milestones, and activities. The following questions must be
answered with a firm YES!

- Are the deliverables consistent with the iteration objectives?
- Are the iteration evaluation criteria consistent with the iteration objectives?
- Are the milestones consistent with the deliverables?
- Are the activities consistent with the deliverables?
- Have all significant differences between top-down and bottom-up estimates of effort been resolved?
- Does the team constitution reflect the required amount of effort per discipline?

In this article we have presented a sequence of planning activities, but keep in mind that, in reality, many activities are actually performed simultaneously, and some are repeated until a plan evolves that satisfies the questions listed above. Once you have gained sufficient experience with the techniques outlined here, you will find yourself performing these steps as one indistinguishable blur. Until then, however, it is useful to be consciously aware of which step you are performing and to follow the guidelines described here.

**Conclusion**

Applying new techniques for the first time, without experience to fall back on, may be intimidating. The measures you gain from experience are the "black magic" project managers often refer to. With the techniques described in this article, you can acquire that "black magic." By taking an educated guess and carefully monitoring the first weeks of the project, you can refine these *guesstimates* to represent project-specific indicators.

You need courage and belief to plan and manage uncertainty. Jules Verne's *belief* led to the 1865 publication of *De la Terre à la Lune*, a novel about a fantastic journey to the Moon. In 1969 Neil Armstrong had the *courage* to become the first person to touch the Moon's surface. In the upcoming decades we will be heading for Mars. With *belief* and *courage*, maybe you can plan a bold project yourself and explore new frontiers.

**References**


Dave West, "Planning a Project with the Rational Unified Process." Rational
Notes

1 These are projects in which many interrelated activities are performed concurrently, rather than sequentially (as is typical of linear engineering projects). Software development and research projects are examples of concurrent engineering projects.


4 Many claim that estimation is an art form. With the right tools and techniques, you may not become Van Gogh, but you will sell many paintings.

5 A figure of 350 or 400 conveys your uncertainty more realistically than a figure of 367,45.

6 Technically, phases could be performed in parallel, or overlap somewhat, but this is usually unnecessarily difficult to manage.


8 Between 1,000 and 20,000 person-hours with a schedule of one half year to two years.


For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
IBM Rational Rapid Developer: More control for architects over code, design, and deployment

by Joseph G. Noonan
Software Quality Engineering Manager
Rational Software
IBM Software Group

Application development has never been easy. As software has increased in importance, so has its complexity. In an effort to reduce this complexity and increase ease of use, software tool providers have endeavored for many years to create rapid application development (RAD) tools that allow developers to quickly create applications. The first iterations were not actually tools, but rather libraries that provided developers with key functionality, mostly focused on generating user interfaces. These offerings eventually expanded to encompass sophisticated libraries for communications and key functions. However, although they accelerated development, these libraries were proprietary and still required significant skill to use correctly.

In order to make RAD more practical, and open programming to a broader base of programmers, tool vendors developed products such as Microsoft® Visual Basic and Sybase® PowerBuilder, which provided an entire environment for the developer. The supporting languages were simple to learn, and the tools themselves provided accelerators such as GUI designers that allowed the user to "draw" the user interface by dragging and dropping controls onto a virtual canvas and then attaching actions in the form of code functions to the controls. These environments quickly generated converts, and many millions of lines of code were created using their tools.

But problems persisted. Although these tools were used to create many critical applications, they did little to control the methods developers used. For the most part, applications were neither model-based nor founded on strong architectures. Therefore, developers could (and did) design their
applications however they saw fit. The resulting applications were usually unstructured, poorly designed, and naturally difficult to maintain. A general lack of code-sharing practices meant that applications often delivered the same basic function sets in several different (and sometimes conflicting) ways. In addition, when organizations wanted to expand the scope of an unstructured application beyond a single department, the application was difficult to scale, given the tool set used to create it. Also, the underlying technologies of these tools sets were proprietary, so they were difficult to integrate with other technologies.

To combat this chaos, Rational Software Corporation, now IBM Rational, part of IBM Software Group, has provided the software development industry with a much more standards-based approach to the application development lifecycle. This approach includes languages (UML), tools (IBM® Rational Rose,® IBM® Rational ClearCase,® IBM® Rational ClearQuest,® and IBM® Rational XDE,™ to name a few), and industry leading best practices (via Rational Unified Process,® or RUP®). In May of 2003, IBM® Rational® Rapid Developer joined this distinguished list of Rational resources and technologies. Years in the making, Rational Rapid Developer provides an architected approach to RAD, based on standards such as J2EE. The new approach is called "architected RAD," and it provides a variety of accelerators and controls over development of applications. With Rational Rapid Developer, less experienced developers can join forces with highly skilled architects and senior developers to create solidly designed applications.

The Gartner Group recognizes tools of this nature as Architected RAD or "ARAD" tools. Some characteristics of these tools include:

- Support for industry standards such as J2EE, UML, XML, SOAP, and MDA.
- Use of industry standard design and construction patterns, such as MVC2.
- Engineered quality in constructed code.
- Architectural consistency across teams of developers.

According to Gartner,

There's no better way to improve productivity, reduce cost, and ensure scalability and performance of applications, while still putting methods in place that are "minimally invasive."

For most enterprises and applications, ARAD should prove to be a near-ideal approach to balancing speed and cost with "just enough"application quality and performance. 1

This article will introduce the major attributes of IBM Rational Rapid Developer that enable architectural and construction control, thereby ensuring high-quality business applications while giving development teams great flexibility in technology, construction, and deployment.
Architected RAD: Importing a UML base model

The heart of every application is the base model that describes it -- a domain model, class model, logical model, entity model, or the like. In all instances, this base model provides a view of the key classes and their relationships, and IBM Rational Rapid Developer supports this view graphically with its UML-based Class Architect (see Figure 1).

![Figure 1: UML class model in IBM Rational Rapid Developer](Click to enlarge)

You can develop this model in several ways. You can simply use the friendly drawing tools to drag, drop, and connect classes and packages on the canvas of the "Class Architect" feature within Rational Rapid Developer. More often than not, however, the development team will want to leverage existing assets. The architect and senior developers can import these assets into the application project in Rational Rapid Developer from a couple of sources, the first being an existing database. Using the database import wizard from the Tools menu, the architect can connect to the database and then import the table and relationship definitions by selecting from the list of tables and views. This can be done as many times as needed over the development cycle. In addition, changes the
architect makes to the model in Rational Rapid Developer can be propagated to the database or into DDL scripts that the database administrator can run. Rational Rapid Developer also provides facilities to import definitions of legacy assets, such as IMS or VSAM database definitions.

An architect can also create a class or information model by using the model synchronization feature in IBM Rational Rapid Developer (see Figure 2). This feature connects to existing models created in either the IBM Rational Rose or IBM Rational XDE modeling tools, which are mainly geared for senior developers and architects; they provide a very rich modeling environment based on the Unified Modeling Language (UML) standard. Using the connectivity feature to access these existing models in Rational Rapid Developer, an architect can push a fully developed class model down to users and give them a jumpstart on development. Once the initial model is imported, these users can also initiate incremental updates; as the architect modifies the model, they will receive the changes without having to re-import the entire model.

![Class Model Synchronization Tool - Preview](image)

**Figure 2: IBM Rational Rapid Developer offers access to UML class models in IBM Rational Rose or IBM Rational XDE**

In addition, the architect can easily filter out elements that are not ready for use or that are not relevant to the application his team wants to develop.
Architected RAD: Maintaining design and branding consistency

IBM Rational Rapid Developer supports a very rich user interface design environment. Developers and graphic designers can lay out an application’s user interface using drag-and-drop controls and special nestable grids to manage and display complex content.

Rational Rapid Developer also supports the creation of a style repository that can help ensure uniform page layouts and consistent branding if multiple contributors will be creating applications and adding content to a Web site. A style repository contains details of color schemes, font styles, themes (explained below), editing and navigation controls, and page styles. A graphic designer can create a series of templates offering variations in style, color, design elements, and so forth, that still maintain branding consistency for the site.

In Figure 3, the graphic designer has created a style repository with specifications for buttons, tabs, grids, and labels, as well as some document templates. Another development team can import this style repository into another application they are creating with Rational Rapid Developer, and it will determine what controls and styles that development team may use.

In addition, the designer can create themes, using the Theme Architect, for the style repository. Themes consist of color sets, image sets, and font sets that can be named and then used either specifically for a given page or loaded dynamically as a user option.
Architected RAD: Maintaining control over code

IBM Rational Rapid Developer has many features that provide greater control and consistency for code while making coding easier for developers.

Code generation for multiple technologies

Through its combination of base, interaction, and user interface models, IBM Rational Rapid Developer can generate the bulk of the code for all n-tiers of a J2EE application. Currently, it can generate applications for a wide variety of deployment technologies, which the architect/developer can select from a dropdown menu. Moving an application from one vendor technology to another -- or from an older version of a technology to the latest version -- is as easy as selecting another menu option and clicking the Construct button. This means that developers can have one development environment on their desktop (e.g., Tomcat with SQL Server), and a completely different environment for production (e.g., IBM WebSphere with IBM DB2), without worrying about the differences between those technologies.

From an application maintenance perspective, this capability ensures that development teams can spend the vast majority of their time solving business problems by adding new features and capabilities to applications, rather than on porting code from one technology to another.

Code construction patterns

A unique feature of IBM Rational Rapid Developer is that it allows architects and senior developers to define design patterns for constructing the application. Rational Rapid Developer has a specific set of default patterns for classes, pages, messages, and components (Web services) that are loaded with each application (see Figure 4). For each category, architects and designers can either choose one of these default patterns or mix and match to achieve a specific result.
Each pattern has a named descriptor that can be assigned to a page, message, or component. Patterns can be transactional or non-transactional, and they can support dynamic SQL or stored procedures. They can also support entity beans, JSPs (JSP Model II), or servlets. This allows the architect to select the patterns that are best for the application and enforce their use by developers.

**Code templates**

Rational Rapid Developer includes a coding environment called Logic Architect. This is where the programmer actually writes the business logic. In order to streamline this process and provide consistency among the coding done by different developers, Logic Architect provides a series of code templates that perform common business application processing tasks. The developer picks a template and responds to the wizard’s questions for that template, and then Logic Architect populates the template with appropriate code (see Figures 5 and 6).
Figure 5: Code templates for Logic Architect
For example, if the developer selects a template for calling a stored procedure, he or she enters the database descriptor name and the name of the stored procedure; then, Logic Architect inserts the code automatically. This is an open system that allows the architect to both modify existing code templates and add new ones. The bottom line is that the architect or senior developer can maintain control over the code developers use across the project.

Security control

IBM Rational Rapid Developer also provides advanced features for more sophisticated developers that extend access to custom security providers and components. It supports role-based security as well as data encryption and permits the addition of customized security mechanisms that comply with the J2EE JCE (Java Cryptography Extender) standard. This allows architects to either create their own security mechanisms in the form of a JCE provider or leverage existing third-party JCE implementations.

Control via custom components

Another way that architects can control development is by employing components in the form of Java packages, or EJ Bs. These services-oriented components can represent common business logic that Rational Rapid Developer users can call from their custom methods.

Architected RAD: Deployment control
Most development tools provide a local environment in which developers can deploy their applications for testing. However, the actual target production environment is usually significantly more complex, and you need specialized skills to determine how to partition the application for deployment. IBM Rational Rapid Developer provides architects and senior developers with an advanced tool, the Partition Architect, to create partition and deployment definitions for the application. Rational Rapid Developer provides a "local" model that defaults to Apache Tomcat, which is an open source servlet/JSP engine. Rational Rapid Developer also supports deployment onto a variety of J2EE platforms, such as IBM WebSphere (versions 3.5, 4.0, and 5.0). Partition Architect allows segmentation of the application into different tiers (presentation, business, and data) as well as along functional lines (buyer, seller, administrator, etc.). See Figure 7.

![Rational Rapid Developer](image)

**Figure 7:** IBM Rational Rapid Developer has Partition Architect, which provides advanced deployment control.

Click to enlarge

**Conclusion**

Unlike RAD approaches of the past, IBM Rational Rapid Developer provides architects and senior developers with a great deal of control over application development. Architects can ensure architectural consistency by controlling the code generation process, ensure layout and branding consistency through the style repository and templates, and ensure deployment success through advanced partitioning controls. They can also provide less experienced developers with templates and tools that make
their work faster and more efficient. The end result is a quality application that is built using the best practices for the target platform, and a team that works well together and has a head start on the next project.

Notes


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

**Waltzing with Bears: Managing Risk on Software Projects**
by Tom DeMarco and Timothy Lister

Dorset House, 2003
ISBN: 0-932633-60-9
Cover price: US$27.95
144 pages

Demarco and Lister are very well-known for their classic, *Peopleware*, which should be mandatory reading for all software managers. Their new book, *Waltzing with Bears*, should be mandatory reading for software managers, project managers, product marketing managers -- in fact, anyone involved in the decision-making process for funding software projects. It's just as pithy and to the point as *Peopleware*, and well worth the investment of a few hours of time. When you're done reading it, you'll probably be tempted to buy a few more copies to give to coworkers.

Early in my own career, a wiser and more experienced team leader gave me the advice "Always pad your schedules." This book demonstrates exactly why this was excellent advice, and gives concrete techniques for estimating the most probable delivery dates for a project. Team leaders will love these techniques, as well as the online Web tool the authors make freely available, which helps you calculate how much to pad a schedule to deal with the usual emergencies.

Fundamentally, the book is about how to use risk management techniques to plan for, and cope with, the most common IT project killers: unrealistic schedule estimations, feature creep, employee turnover, lack of stakeholder agreement, and underperforming contributors. The authors' examples (particularly their detailed analysis of the software fiasco at the Denver International Airport) bring home that most "surprises" that occur during software projects are actually foreseeable, and you can plan for them in advance.

Echoing the IBM Rational message, Marco and Lister argue that iterative development is an excellent risk reduction technique, provided that the iterations are properly planned. In a chapter on that topic, they show that iterative development is also an excellent approach for quantifying return-on-investment; and it often serves to expose "bells and whistles" for what they really are: extras that could be dropped from the project if necessary.
One of the most valuable pieces of advice that I took from Waltzing with Bears was that "Costs and benefits need to be specified to equal precision." In other words, if the value proposition for a project is "We gotta have it," then the cost estimate should be "It's gonna be expensive."

Marco and Lister also make the thought-provoking argument that many IT projects cost much more than the benefit they produce, largely because no one does a proper return-on-investment analysis before the company decides to undertake the project. I especially like their conclusion that most death march projects yield almost no benefit at all. After all, they reason, if a project were really critical to a company's survival, wouldn't the company invest everything it had to make sure the project finished on time and was executed properly? But if you've ever worked on such a project, you know that isn't how companies behave.

Waltzing with Bears is very readable (I read most of it in a morning at the car dealership) and memorable. Everyone who has ever worked on a software project that went over schedule will recognize his or her organization somewhere in this book. If you're a project manager, you probably won't want to use all the techniques in this book immediately. However, maybe you will make your team start doing some risk mitigation up front and calculating return on investment while planning the next project. This alone will dramatically reduce your odds of ending up with a death march project. Your managers will thank you for that. And so will your team!

- **Beth Benoit**
Senior Software Developer
Rational Software
IBM Software Group

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

*Writing Better Requirements*
by Ian F. Alexander and Richard Stevens

Addison-Wesley, 2002  
Cover price: US$34.99  
176 pages

Whether you are new to writing requirements or an old pro, this book is a must read. For novices, the authors provide a step-by-step methodology for writing requirements. For the pro, the layout of the requirements gathering and writing process is an excellent refresher. Do you want to achieve greater clarity when using common terms or improve the structure of your requirements document? This book has the required level of detail to help.

The first chapter covers the preliminaries and defines common terms such as user, customer, and different names used for requirements. It finishes with a description of requirements writing within the larger context of system development that is, unfortunately, buried in the text and treated too lightly. All too often, project teams do not really understand the discipline of requirements gathering and writing. This sometimes results in animosity from team members who mistakenly think their agendas conflict with that of the analysts and resent the amount of time these activities require. A separate chapter expanding upon the relationship between requirements and the rest of the development process would have helped readers understand that writing requirements is a crucial activity; it affects the entire project team as well as the quality of the product.

In subsequent chapters, the authors do a good job of describing a process for writing better requirements, providing useful techniques for each process step. The most difficult step in the requirements process is gathering proper, well-articulated requirements from stakeholders, and the techniques presented in the chapter on requirements gathering are particularly useful. Conducting interviews, holding workshops, and observing users at work are some of the techniques the authors identify and describe. In many cases users do not really know what their requirements are, so augmenting the interview or workshop with a prototype usually spurs their imagination. Regardless of the format interviewers chose for gathering requirements, the authors encourage
them to keep asking the "Why?" question until the real requirement is revealed. Other useful questions are "So what?" and "Who cares?"

The exercises in these chapters are also well presented and well thought out -- every chapter has at least one exercise that assigns the reader tasks based on ideas and techniques covered in the text. For example, the exercises in Chapter 4 -- "Other Sources of Requirements" -- provide the reader with sample specifications. The task is to read the specification, extract statements from the text, and categorize these statements according to their type (User Requirements, System Specification, Design Elements, etc.). The first exercise identifies statements that need categorizing, and the second exercise is "free-form" -- the reader must pick out the statements for categorizing. Completing all of the exercises gives the user a practical way to learn and experience the process of gathering and writing requirements.

A chapter on how not to write requirements adds some humor to the book. For example, "The battery low warning lamp shall light up when the voltage drops below 3.6 volts, and the current workspace or input data shall be saved," is an example of including multiple requirements in one statement. (I don't think the two requirements are related, but you never know -- they just might be!) The example, "The fire alarm shall always be sounded when smoke is detected, unless the alarm is being tested or the engineer has suppressed the alarm," points out that "let-out" clauses, such as those that begin with "unless," can be dangerous. Actually, a fire alarm built to this requirement would be a real danger! (I am sure I have written a few laughable requirements in my lifetime, too.)

In addition to the few minor weaknesses I've cited, the authors include a single snippet of the European Space Agency's Software Development Standards but then do not reference it again. This left me frustrated. I felt they should either have provided more information and excerpts as an Appendix or simply not mentioned the standards at all.

Overall, however, I am confident that this book will help all who read it not only to improve their ability to write requirements but also to convey the importance of good requirements writing to other team members. I have found it particularly useful to give people examples of bad requirements. In our daily speech we use all sorts of constructs -- let-out clauses, wishful thinking, rambling, and so forth -- that we must avoid when writing requirements, and the examples this book provides are a great way to demonstrate the confusion they can cause.

As a product manager, I constantly write requirements documents; I know that I will use this book as a reference to gather and categorize them, and to lay them out clearly.

-Peter Hanschke
Rational Software Canada
IBM Rational Software Group
Delivering packaged application solutions with Rational Unified Process

by Dennis Moya
Consulting IT Specialist Manager
Rational Software
IBM Software Group

Project teams adopting Rational Unified Process,® or RUP,® often have many practical questions about applying and adapting its guidance to their particular problem space. This has been especially true for packaged application implementations, which historically have been dominated by waterfall-based development practices and a unique perspective on the nature of requirements. This article presents an overview of problems related to packaged-based development, and describes an implementation through the RUP phases and key process activities.

Business applications are increasingly at the center of an organization's interaction with its customers. Packaged applications, such as Customer Relationship Management (CRM) and Enterprise Resource Planning (ERP) systems, are a common way of delivering these complex systems. However, enterprise-class business applications often exceed the scope of a single package; IT staff developers need effective practices for delivering such applications, including techniques for integrating a packaged solution with existing applications and infrastructure to produce larger and more capable systems.

The Rational Unified Process provides software development best practices to handle these difficult projects, but its guidance is stated in general terms that can be difficult to relate to the packaged application domain. Teams often ask: What are the most important activities? How should we adapt the guidance on requirement specifications? How can we realize the implementation benefits promised by the package? This article addresses
such questions. But before we begin describing how a packaged application implementation would proceed through the RUP phases and the key activities within each phase, let us set the stage with a brief discussion of business application development and packaged application frameworks.

Note that an in-depth discussion of RUP is beyond the scope of this article; if you would like more information, see the References listed below.

**Understanding business needs**

Today's IT organizations are increasingly concerned about aligning their technology investments with corporate strategies and business objectives. *Most business application development projects arise from the realization that there is a misalignment between the needs of the business and the technologies intended to support these needs.* There may be new business goals or opportunities, the technology may be aging or growing unreliable, or the current automation might just be inadequate.

*An understanding of business needs should drive the creation of a new software solution.* To ensure project success, you must:

- Define the business capability you want to achieve.
- Determine the technology needed to deliver that capability.
- Define a plan to transform the organization from its current state to the one you define as an end goal -- within the business constraints of cost, time, and quality.

A solid understanding of business needs not only ensures project success; it also ensures that the project will improve alignment between the two key domains within an organization:

- **The business domain**, *defined by the business purpose, goals, and strategies.* This is where people collaborate and execute business activities that fulfill the overall mission. They leverage technology to perform specific activities and manage information.

- **The technology domain** *is the world of systems and applications that business workers use.* It also includes the infrastructure that provides interoperability and availability of these resources across organizations and geographies.
Since the business perspective is the driver in this process, business modeling and/or domain analysis should play a key role in the development lifecycle. The scope and focus of the business modeling activities will depend on the complexity of change you're trying to introduce into the business and how well the existing business practices are documented and understood.

**Choosing the right packaged application**

Once they have specified business goals, many development organizations turn to packaged applications to provide part of their overall solution. A packaged application provides a predefined solution within a well-defined business area. Underlying this solution is the package framework -- the architectural foundation that determines the general behavior and extent of the out-of-the-box functionality. The framework includes some key elements:

- Built-in processes and rules implementing a vendor's recommended practices within the business function.

- The major abstractions representing the business objects (e.g., Accounts, Contacts, Orders); the user interface elements that allow you to create, modify, and relate objects to each other; and the elements that provide process and communication control.

- A set of mechanisms that control the interaction and exchange of information with other systems and applications; this allows you to integrate functionality to provide a broader set of capabilities.

These elements typically allow some configuration capability so you can better align the application with business needs. *Changes that go beyond...*
The recommended configuration parameters represent a customization of the framework, and can affect the package's maintainability and performance.

Figure 2 shows how the application framework relates to the business and technology domains of an organization. Based on this model, the packaged application development process then focuses on:

- Identifying business operations that the application can deliver effectively.
- Understanding how the application will interact and exchange information with other systems to deliver the broader capabilities, and how the integration technologies can support these interactions.
- Mapping the logical business and information entities to their representations (abstractions) in the application.

**Figure 2: The packaged application framework bridges the organization's business and technology domains.**

When using a packaged application, there should be a good alignment between the package's core processes and the organization's desired business practices. This helps reduce effort compared to traditional custom development projects. If there are differences that go beyond configuration, the organization should be prepared to adapt its business practices to the package in order to minimize risk and reduce the cost of implementation. If the alignment is poor, it will be difficult to realize the packaged application's benefits; customization can compromise the application's long-term viability and demand enough effort to negate the cost benefits of starting with a package. To minimize this risk, you must conduct a well-defined package evaluation and selection process. To do
this effectively, you must have a detailed description of the desired business environment so that you can evaluate all aspects of the package's functionality and framework in relation to each business goal.

**Using RUP for a packaged application development project**

Now that you understand the importance of choosing a packaged application suited to your business goals, let's see how RUP can take you through that process as well as the rest of your project lifecycle. RUP divides the software development lifecycle into four phases, which we will summarize below, from a packaged application project perspective:

- **Inception** describes the scope of the problem and your vision of a solution. For a packaged application you would describe the business processes underlying the problem, identify the packaged offerings that can support some or all of those needs, and identify at least one packaged solution that supports the buy-versus-build option.

- In **Elaboration** you would find all the necessary components of the solution and describe how they will fit together. You would select the packaged offering that best fits the business; identify the package responsibilities; identify the key requirement gaps that will drive configuration; and configure the key application objects and mechanisms that deliver the core functionality.

- During **Construction**, you would build upon the elaboration framework and expand the functionality to its final form. You would configure the application objects, business processes, and rules based on the requirement gaps you identified during Elaboration and the early stages of Construction.

- **Transition** includes all the preparation and activities you would need to deliver the application to the end users. This typically involves migrating the configured package to the production systems, enabling the interfaces with other systems, and loading any legacy data required for operation.

Each phase represents a refinement of the solution, from the initial idea through to the final product. The conclusion of each phase represents a major milestone that involves a review of the project's viability.

- In Inception you establish the business case for the project.
- In Elaboration you demonstrate a viable architecture that meets the constraints of cost, quality, and effort.
- At the end of Construction you examine the readiness of the application for production use.
You complete the development cycle with the successful Transition of the application to the user community.

**Iterations**

With RUP, you deliver an intermediate product in each phase through a series of iterations; each iteration improves your understanding of the problem, its requirements, and the corresponding solution. *Phases and iterations do not focus on a single development activity such as analysis or design; rather, each one includes all of the activities the team must perform to evolve its understanding to the next level.* This is a significant difference from traditional waterfall approaches. It requires a highly integrated team that brings together the business, application, and infrastructure expertise to discover conflicts and determine trade-offs early in the development lifecycle.

**Reducing risks**

Plan iterations to systematically resolve the significant risks or concerns of each phase of development. For packaged applications, the *first concern is to establish the functional boundaries of the package relative to the rest of the application or legacy environment.* This involves finding the best distribution of business responsibility among systems that have complementary or overlapping functionality, and identifying the business services, data exchanges, and interfaces needed to support their interaction. You should evaluate the various options within the constraints of the platform, technology, and integration standards that the IT organization adopts.

The second level of concern is the *extent to which the package will support the assigned responsibility.* Here we have *two broad categories of risk:* the structure and relationships of the key application objects, and *the business rules that govern the activities within a business process.* For example, in Figure 3 we see that the Account, Product, and Contact objects impact all areas of the Customer Relationship Management (CRM) system, and are likely to be the elements that tie together the process flows. The first iterations should be targeted at understanding how these elements interact and support the combined perspectives of Marketing, Sales, and Service.
Evolution through incremental releases

When you consider the complexity of an enterprise-class packaged application, delivering a full set of functionality in a single development cycle may not be practical. Instead, you can divide development into a series of releases, with each release expanding on the breadth or depth of system capabilities. This provides a number of benefits:

- **Smaller sets of functionality can be delivered faster and more frequently.**
- There is more control over the amount of change that any one release introduces into the business and infrastructure.
- You can refine the business rules and degree of automation based on experience, avoiding unnecessary precision and detail that waterfall approaches often introduce.
- There is greater flexibility to adapt to changes in the business environment.

With this approach you need to decompose the scope of the business problem into manageable units of work, and then organize these units into a release roadmap. A common practice with business applications is to partition capabilities by function or departments, such as sales or
marketing. While this can deliver highly optimized solutions within a functional area, it usually fails to provide an integrated solution to the real business problem.

The alternative is to look at the business processes as a "value chain," or functions that incrementally add substance to the end product or goal. For example, the CRM system in Figure 3 has separate marketing and sales components. Although we can develop marketing campaigns separately from the sales functionality, the real value of the CRM system is to apply the history recorded with each sales opportunity to the creation of campaigns that produce high quality opportunities.

Understanding the value chain allows us to focus the details of requirements and analysis within the processes of immediate importance to us, and proceed with less precision around the requirements further out in the evolution (see Figure 4). The attention is on detailing the requirements and system behavior needed to fulfill the process responsibility, and providing the foundation for future work that delivers the next level of capability.

**Figure 4: Requirements are refined as you progress through each evolutionary development cycle.**

But this perspective, by itself, is not sufficient to determine the release roadmap. A packaged application can consist of several modules that may need to be deployed as a unit. You may need to hide functions of a module that are not fully implemented from the user interface, or increase the functional scope in order to deploy the module. Legacy systems will also constrain the sequence in which you can replace or deliver functionality. You may need wrappers, temporary interfaces, or a broader set of functionality before you can retire a system or integrate it with the package. The development roadmap will reflect the best alignment for the boundaries of the process, application, and legacy systems (see Figure 5).
The constraints of application components and legacy systems will influence the sequence in which you can deliver new functionality.

Development activities

The following sections highlight key activities and concerns in RUP phases as they relate to packaged application development.

Inception

Broadly speaking, the purpose of Inception is to mitigate business risks. In this case, you need to demonstrate that package-based development represents a viable approach to the business problem. The first concern is to express stakeholder needs in terms of business processes that will deliver the desired business environment. You should detail the critical business scenarios within the process enough to understand the participating business workers, and their responsibilities and interactions. The resulting business use cases support a number of activities:

- You can use them to derive application-independent test cases that support the packaged application evaluation and selection process.
- Once the packaged application is selected, you can use the business use cases to derive the system requirements needed to support the business.
- You can use them as input for developing workload models that drive performance and scalability testing.
- They support end-user acceptance testing.
Your next concern is to identify the commercial package offerings that you can potentially apply to the business problem. As you identify business scenarios, you can survey these packages and match their capabilities against business needs. This provides not only a short-list of packages that should proceed to formal evaluation, but also the initial functional boundaries for the package (for example, that it will support sales lead generation through quotes).

From the general capabilities of these packages, you can evaluate which business responsibilities they should support, and which ones should remain within the scope of existing systems or human workers. From this division of responsibilities you can then identify the interactions among systems and consider the technologies needed to support them. You can also introduce and add the requirements and standards of the IT infrastructure to the packaged application evaluation criteria. You should refine this analysis throughout Inception to develop one or more candidate solutions to the business problem (see Figure 6).

The functional requirements for the package will be constrained by the application framework, so defer them until you select the final application. **Inception will be complete when you have a clear picture of the boundaries for the packaged application's responsibilities, and can offer at least one candidate solution that meets the business constraints.**

---

**Figure 6:** During Inception you develop your vision for the packaged application solution, based on the organization's business needs.

**Elaboration**

In *Elaboration* you need to complete the package selection and, based on the package's capabilities, describe the system architecture. Use the
business models and test cases from Inception to identify the packaged application that offers the best functional fit with your organization's needs and process models. You should develop a prioritized list of evaluation criteria during Inception, or before the formal selection process begins. Before you begin system development, resolve any changes in assumptions or process conflicts that you identify during the evaluation and update your business models.

Once you select a package, the critical activity is to finalize the division of business responsibilities across systems and workers. This determines the functional scope of the package and interfaces that you will need to develop. Refine the functional boundaries for the package that you proposed in Inception to produce an overall blueprint for the application and its interfaces.

You derive system requirements from the business responsibilities you allocate to the package. During Elaboration the focus is on the scenarios that deliver critical business capabilities; for these scenarios the business models should detail activities and business rules associated with workers' responsibilities. Review the models to determine what specific actions the application can support, and then capture these actions in a system use case. Note that you should determine the requirements in the context of the package framework, where you can set realistic expectations for stakeholders and consider trade-offs for needs that go beyond the tailoring the package normally offers. You can use a "vanilla," or nonconfigured, system to walk stakeholders through the package's out-of-the-box functions, so they will understand its limits and capabilities before deriving formal requirements.

With packaged applications, analysis and design activities focus on "gap analysis," which involves identifying the application objects responsible for a behavior and determining the configuration needed to meet the requirements of the associated use case. This is essentially a mapping process, in which you align business objects and needs with the application's available objects and mechanisms. These "requirement gaps" provide a high-level design that drives the configuration of the application framework. They are linked to the original requirement and are the basis for determining the level of effort for an iteration.

The primary targets of configuration are the application's Business Entities and User Interface (UI) objects. You can modify a business entity to extend its attributes and relationships, and control visibility and access to data through the UI. Using the context the business model provides, combined with knowledge of the application environment, you can identify the application objects related to a set of requirements. The roles, operations, and type of business data manipulated in a use case will point to relevant application screens, and from those you can determine the underlying business objects. Figure 7 shows this mapping for a simple case for recording account activities in a Siebel CRM system. The discovered application objects are captured in the analysis model, and traced back to logical entities in the business model to record how they are realized in the packaged application.
Figure 7: The business model helps identify the application screens and objects responsible for the business behavior.

Other configuration areas include enforcing workflow or business policies, integration with other business applications, and migrating data from legacy systems (see Figure 8). The mechanisms to implement these will vary across commercial packages, and that will influence how you describe or represent the requirement gap. In some cases, a significant amount of application integration activity might pose a significant architectural risk to the system. You should attack such areas early in the Elaboration phase.

Examine the requirement gaps to identify changes with the greatest impact on the application. Some of the most important cases include:

- Business objects common to multiple process or functional areas (see Figure 3).
- State machines that define the lifecycle of an object.
- Attributes that define object hierarchies, assignment, or workflow processing.
- Nonpersistent data and integration objects representing information exchange between systems.

In general, the foundation for all the application capabilities lies in the attributes and relationships of the business objects supporting the requirements. A major advantage of packaged applications is that these structural changes are relatively easy to implement and make visible through the user interface, providing an executable environment in which the project can better validate the system's key architectural assumptions.
In summary, the Elaboration phase focuses on assessing the gaps between the desired system and the packaged application’s out-of-the-box behavior. You assess the extent and complexity of the requirement gaps to estimate the overall effort required to complete the configuration during the Construction phase. **Elaboration is complete when you have assessed that you can handle the outstanding technical risks satisfactorily during the remainder of the project, without significant risks to schedule or budget.**

![Diagram](image)

**Figure 8: During Elaboration you define application boundaries and identify abstractions and mechanisms that will realize desirable system behavior.**

**Construction**

*During Construction you implement configuration changes you identified in your requirement gap analyses; in particular, this involves fine-tuning the application’s business rules, user interface, and workflows. Configuration is significantly faster than writing code, so you can invest resources you would normally apply to traditional development in testing the correctness and completeness of the configuration, and providing interim releases for stakeholder evaluation. As you approach completion of the requirements, you should demonstrate stability through consistent reductions of defects and change requests.*

A main concern of Construction is to **organize the work represented by requirement gaps into efficient units that minimize conflicts among developers working on related activities.** The best way to achieve this is to **align the work with the architectural elements of the package framework.** Understanding the package configuration tools and how they expose the elements of the framework will help you determine the most effective approach to organizing the configuration efforts. For example, in Figure 3, dividing the work along sales and marketing lines can lead to contention about changes to the Account object.

At the same time, it should be easy to relate a gap back to the original use case, so the developer can understand the full context of the change. **Capturing and maintaining traceability between system requirements and requirement gaps should be an integral part of the requirements**
management process. This will prove vital in assessing the cost of implementing a given requirement, and determining the impact of changing a requirement later in the development cycle.

Figure 9 presents a basic requirements structure that provides traceability between requirements and gaps, and supports organizing the gaps by either application objects or type of configuration. You can also relate the gaps to work orders used to track the progress of Construction activities.

Data integration deserves special attention during this phase. In Elaboration we mapped application objects and attributes across the systems; now we need to identify and implement the data transformations needed to accommodate differences among these attributes. This applies to data originating from interactions with other systems as well as from the retirement of a legacy system. The activity can be particularly troublesome if you are applying new integrity constraints to the system. In the case of legacy migration, scripts and in-house or purchased tools are typically required for the Extraction, Transformation, and Load (ETL) of enterprise data. You should plan and successfully complete formal and extensive testing before moving on to the Transition phase.

At the close of Construction you will have finished configuration of the requirement gaps you needed to fill in order to fulfill the package requirements, and implemented the scripts and components you need to integrate the application into the legacy environment. The system is now ready to be deployed to a test
environment for stakeholder evaluation.

Transition

The Transition phase deals with delivery of the solution to the user community. The evaluation releases delivered to stakeholders during Construction are now replaced with a formal end-user acceptance test (UAT), and a go/no-go decision for a rollout to the production environment.

From the development point of view, this is a fine-tuning phase. You may need to make minor adjustments to successfully complete UAT, but the application functionality is now complete. You should include final versions of usage models and the most efficient navigation paths in the training materials. You should plan and, to the extent possible, verify the transfer to the production systems through a series of dry runs on the test platforms. During this period you should also plan the production hand-over to the customer and train operational support personnel.

The deployment perspective deals with the logistics of the rollout, based on factors such as number of users and locations affected; volume and number of legacy data sources; availability of support resources; and training schedules. These logistical activities can extend over a period of time, but the end of Transition is not necessarily tied to their completion. Rather, Transition ends when you have completed enough activities for the project to be measured relative to the release success criteria.

Key considerations

Packaged application projects that are adopting RUP should keep the following points in mind:

- **Take a gradual approach to adopting RUP.** As we discussed earlier, don't expect to develop and apply a complete process in day one. The workflows and activities should accommodate the experience and culture of your team. Focus on the phases and milestones immediately ahead of you, and refine the process with each iteration and development cycle.

- **Tailor RUP to your project needs.** As we noted, RUP is a framework that you should adapt to reflect your development project's particular needs. Add your domain expertise to those areas that provide only general guidance, and hide the sections that don't apply to your project.

- **Build an integrated and collaborative team.** A highly integrated and collaborative team is key to the successful adoption of RUP and the delivery of a package-based solution. Both RUP and the packaged applications embody multiple perspectives that continuously balance and converge; a complete understanding of these relationships is typically beyond the capability of any one person or function.
• **Create artifacts that everyone understands.** In creating artifacts, always consider the needs and background of those who will be consuming the specifications. Producing complex UML models will not help if a stakeholder or developer is not knowledgeable about UML.

• **Educate your team about iterative development.** The iterative approach to development we discussed above is significantly different from traditional development approaches. Initially, it may be a challenge for the team. For each iteration, make identifying and prioritizing risks an important and visible process, so that the team will understand how and why the intermediate milestones are defined.

**Conclusion**

Packaged applications offer a well-defined set of integrated business processes that include well-defined mechanisms for integration with the broader enterprise environment. Potentially, using a packaged application can give you more agile delivery of business capability and simplify your IT organization's application integration problems. But you may not realize these benefits if you:

• **Approach the project as a custom development undertaking.** Ignoring the constraints of the package framework poses a risk to the application's maintainability and robustness. Instead, work within that framework and carefully consider the trade-offs between compromising basic functionality and adapting the underlying business process.

• **Assume that the business can adapt its model to the package framework.** You should do a thorough assessment of requirement gaps and their business impact, and then take measures to bridge those gaps.

• **Deploy the application as vertical stacks of functionality.** If you delay considering the cross-functional scope of your business processes until integration, you are at risk of discovering architectural flaws when it is difficult and costly to fix them, and potentially compromising delivery of higher business value.

In this article we've described how to apply the Rational Unified Process to the delivery of a packaged application in a way that avoids these and other development pitfalls. At the core of our approach are these key principles:

• **The business perspective drives the solution.** Express stakeholder needs in terms of business processes and responsibilities that will deliver the desired business value. This provides a context in which you can evaluate the suitability of a package as part of the solution.

• **Establish the architecture before building.** Leverage your
business models to establish the package's functional boundaries relative to the rest of the application or legacy environment. Evaluate the interfaces you'll need to support interaction among systems within the constraints of the platform, technology, and integration standards adopted by the IT organization.

- **Derive system requirements in the context of the package framework.** Examine business activities to determine the degree of process automation you can achieve within the recommended limits of package configuration. Capturing and maintaining the traceability between requirements and functional gaps should be an integral part of the requirements management process.

- **Build iteratively.** Evaluate the requirement gaps to identify changes that will have the greatest impact on the application. Plan configuration activities accordingly, and provide interim releases for stakeholder evaluation. These measures promote the early discovery of flaws, and are key in managing scope and reducing risk.

RUP can address all phases of the packaged application development lifecycle. But if your team is new to RUP, you will want to incrementally adopt and refine it as you progress through your development effort. This will allow the team to transition from traditional development while preserving the good practices that allow them to apply their domain expertise effectively.

**References**

http://www.rational.com/products/rup/whitepapers.jsp

Per Kroll and Philippe Kruchten, The Rational Unified Process Made Easy. Addison Wesley, 2003. (Detailed discussion of RUP.) Also see the collection of whitepapers at:
http://www.rational.com/products/rup/whitepapers.jsp

Pan-Wei Ng, "Effective Business Modeling with UML: Describing Business Use Cases and Realizations." The Rational Edge, November 2002. (Great introduction to business modeling.)
http://www.therationaledge.com/content/nov_02/t_businessModelingUML_pn.jsp

Maria Ericsson, "Developing Large-Scale Systems with the Rational Unified Process." Rational whitepaper, 2000. (Perspective on large-scale system development that can be useful for complex package implementations.)
http://www.rational.com/products/rup/whitepapers.jsp

"The Rational Unified Process for Systems Engineering." Rational whitepaper, 2001. (Perspective on large-scale system development that can be useful for complex package implementations.)
http://www.rational.com/products/rup/whitepapers.jsp
"Evolutionary Process for Integrating COTS-intensive Systems (EPIC)." Cecilia Albert and Lisa Brownsword, Software Engineering Institute, 2002. (Describes a RUP-based approach to Commercial Off-The-Shelf (COTS) software development from the COTS-based Initiative at the Software Engineering Institute (SEI). This can be particularly useful when dealing with more formalized package acquisition and procurement processes.)

http://www.sei.cmu.edu/publications/documents/02.reports/02tr005.html

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
"Use-Case Diagrams"
(Chapter 4*)

from Learning UML: Communicating Software Design Graphically
by Sinan Si Alhir
(O'Reilly, 2003)

"The Unified Modeling Language is a language for communicating about systems: an evolutionary, general-purpose, broadly applicable, tool-supported, and industry-standardized modeling language for specifying, visualizing, constructing, and documenting the artifacts of a system-intensive process." So writes Sinan Si Alhir in his new tutorial for this marvelous language, which was conceived by Rational Software Corporation's three amigos: Grady Booch, James Rumbaugh, and Ivar Jacobson.

Sinan Si Alhir's book focuses on mastering UML essentials in an orderly fashion. Featuring an example-driven approach and an evolving project management system case study, it progressively introduces and demonstrates the application of key concepts.

Use cases are a powerful first step in applying the UML to a software development project. In Chapter 4, "Use-Case Diagrams," the author describes how these visual representations are used to model system functionality and drive iteration planning. Covering fundamentals such as using actors, communicating associations, and depicting dependencies, the chapter is written in simple, straightforward language that is accessible to a broad audience.

*Chapter 4 posted in its entirety by permission from O'Reilly (www.oreilly.com).
CHAPTER 4

Use-Case Diagrams

This chapter focuses on use-case diagrams, which depict the functionality of a system. First, I introduce use-case diagrams and how they are used. Next, I discuss actors and use cases. Finally, I go over various relationships relating to actors and use cases. Many details that were not fleshed out in Chapter 2 are more fully elaborated here, and throughout the chapter I include suggestions relating to use-case diagrams.

Use-case modeling is a specialized type of structural modeling concerned with modeling the functionality of a system. You usually apply use-case modeling during requirements activities to capture requirements that define what a system should do. Use-case modeling typically starts early in a project and continues throughout a system development process. It is usually done as a series of workshops between users and analysts of a system in which ideas can be explored and requirements may be fleshed out over time.

As a use-case driven process uses use cases to plan and perform iterations, it is important to understand how use cases are related to one another, including what use cases have in common, which use cases are options of other use cases, and which use cases are similar to each other. Given that every project has limited resources, you can use this information about use cases to determine how best to execute a project. Use cases that are common to two or more other use cases need only be implemented once, and then they can be reused. Use cases that are options of other use cases may be implemented at a later time, and use cases that are similar allow us to implement one use case that may be reused. An understanding of how use cases are related allows users and analysts to negotiate and reach agreement concerning the scope and requirements of a system.

A use-case diagram may be considered a “table of contents” for the functional requirements of a system. The details behind each element on the use case diagram may be captured in textual form or using other UML modeling techniques. All the use-case diagrams and their associated details for a specific system form the functional requirements of the system. However, the UML does not provide any explicit guidance on how to capture the textual details, but focuses more on the notation.
Actors

As discussed in Chapter 2, an actor is a user or external system with which a system being modeled interacts. For example, our project management system involves various types of users, including project managers, resource managers, human resources, and system administrators. These users are all actors.

Actors follow the type-instance dichotomy first discussed in Chapter 2 and applied to classes and objects in Chapter 3. You can use the UML to talk about classes of actors, as well as specific actors of a class. When speaking of a class of actors, it’s customary to use the terms actor or actor class. Thus, while you might think of an actor as a specific thing, in the UML, an actor really represents a class of things. When speaking of a specific actor of a class, use the term actor instance.

An actor is external to a system, interacts with the system, may be a human user or another system, and has goals and responsibilities to satisfy in interacting with the system. Actors address the question of who and what interacts with a system. In the UML, an actor is shown as a “stick figure” icon, or as a class marked with the actor keyword and labeled with the name of the actor class.

Figure 4-1 shows various actors associated with the project management system:

A project manager
   Responsible for ensuring that a project delivers a quality product within specified time and cost, and within specified resource constraints

A resource manager
   Responsible for ensuring that trained and skilled human resources are available for projects

A human resource
   Responsible for ensuring that worker skills are maintained, and that quality work is completed for a project

A system administrator
   Responsible for ensuring that a project management system is available for a project

A backup system
   Responsible for housing backup data for a project management system

An actor instance is a specific user or system. For example, specific users of the project management system include Jonathan, Andy, Si, Phillip, Nora, and so forth. An actor instance is shown similar to an actor class, but is labeled with the actor instance name followed by a colon followed by the actor class name, all fully underlined. Both names are optional, and the colon is only present if the actor class name is specified. Actor instances address the question of who and what specifically interacts with a system.
Figure 4-2 shows various actor instances of the actor classes in Figure 4-1, including Jonathan and Andy who are project managers, Si who is a human resource, Phillip who is a system administrator, Nora who is an unspecified type of actor instance, a backup system named BackupSys1.0, and other actor instances. Just as it’s possible to have an actor of an unspecified type, such as Nora, it is possible to have actors such as HumanResource for which a type is specified, but not a name.

Because actors are external to a system and interact with that system, they define the boundary or scope of the system. For example, given the actors shown in Figure 4-1, we know exactly who and what will interact with the project management system. If we don’t define our actors, we may fall into the trap of endlessly debating whether we have identified all the users of the system and all the other systems that interact with the system. Consequently, because every functional requirement should be of interest to at least one user (otherwise, why would we build the system to provide the functionality?), without identifying actors, we have no way of knowing whether we have identified all the functional requirements of the system. An actor may also
represent a resource owned by another project or purchased rather than built. For example, the backup system must be provided by another project, and it may be purchased from a vendor or built rather than purchased. Independent of how it is developed, we are interested in interacting with it as a resource.

**Use Cases**

As discussed in Chapter 2, a use case is a functional requirement that is described from the perspective of the users of a system. For example, functional requirements for the project management system include: security functionality (such as allowing users to log in and out of the system), inputting of data, processing of data, generation of reports, and so forth.

Use cases follow the type-instance dichotomy first discussed in Chapter 2 and applied to classes and object in Chapter 3. You can use the UML to talk about classes of use cases, and you can use the UML to talk about specific use cases of a class. When speaking of a class of use cases, it’s customary to use the term use case or use-case class. Thus, while you might think of a use case as a specific thing, in the UML, a use case really represents a class of things. When speaking of a specific use case of a class, use the term use-case instance.

A use case defines a functional requirement that is described as a sequence of steps, which include actions performed by a system and interactions between the system and actors. Use cases address the question of how actors interact with a system, and describe the actions the system performs.

In the UML, a use case is shown as an ellipse and labeled with the name of the use-case class. Use cases may be enclosed by a rectangle that represents the boundary of the system that provides the functionality. Figure 4-3 shows various use cases associated with the project management system, including functionality to manage projects, manage resources, and administer the system.

![Figure 4-3. Use cases](image-url)

A use-case instance, often called a scenario, is the performance of a specific sequence of actions and interactions. For example, a specific actor instance that uses security functionality to log in or out of the project management system represents a...
scenario. Other possible scenarios might include a user who enters data, a user who requests processing to be done on the data, or a user who requests that the system generate a report.

A use-case instance is shown similarly to a use case but labeled with the use-case instance name followed by a colon followed by the use-case name, all fully underlined. Both names are optional, and the colon is present only if the use-case class name is specified. Use-case instances address the questions of how actor instances specifically interact with a system and what specific actions the system performs in return. Use-case instances are not commonly shown on use-case diagrams but rather are simply discussed with users to better understand their requirements.

Because use cases are performed by a system, they define the functional requirements of the system. For example, given the use cases shown in Figure 4-3, we know generally what type of functionality must be provided by the project management system. Other use-case diagrams may refine and decompose these use cases. If we don’t define our use cases, we may fall into the trap of endlessly debating whether we have identified all the functionality of the system.

Each use case is composed of one or more behavior sequences. A behavior sequence is a sequence of steps in which each step specifies an action or interaction. Each action specifies processing performed by the system. Each interaction specifies some communication between the system and one of the actors who participate in the use case. For example, a login use case may have the following behavior sequence:

1. The system provides a user interface to capture login information.
2. The user enters her username.
3. The user enters her password.
4. The system validates the username and password.
5. The system responds appropriately by allowing the user to continue, or by rejecting the user, depending on whether her username and password are valid.

Steps 1, 4, and 5 are actions that the system performs, and steps 2 and 3 are interactions between the system and user. It is fairly simple to see that step 4 is an action, but you may be wondering why steps 1 and 5 are actions rather than interactions, when the system is interacting with the user by presenting a user interface to the user or by responding appropriately with a message window or something similar. Steps 1 and 5 are actions rather than interactions because the system is simply taking action to show a user interface or a response message window that the user can easily choose to ignore. However, if this were not a human user but another system that would receive some communication to which it must reply, these would be interactions rather than actions. Quite often, trying to classify each step as either an action or interaction is not necessary; rather, it is more important to consider a use case as a dialog between actors and the system, to understand how actors interact with the system, and to understand what actions are the responsibility of the system.
Actions and interactions may also be repeated, conditional, or optional. For example, the Manage Project use case may have a behavior sequence for finding a project on which to work, and the following three succeeding behavior sequences:

- One for managing projects involving employees only
- One for managing projects involving consultants only
- One for managing projects involving both employees and consultants

An instance of the Manage Project use case involves actor instances and finding a project and managing it using one of the three available behavior sequences based upon the type of project selected. Behavior sequences, such as the one shown earlier for the login use case, are commonly captured in textual form, but may also be captured using behavioral modeling techniques, as discussed in Part III. However, the UML does not provide any explicit guidance on how to capture behavior sequences.

Because use cases result in some observable value to one or more actors, they must allow actors to achieve their goals. After all, each actor has goals in interacting with a system. Use cases don’t represent actor goals, but instead represent functionality that enable actors to achieve their goals. For example, the use cases shown in Figure 4-3 enable the actors shown in Figure 4-1 to achieve their goals in the following ways:

- To ensure that a project delivers a quality product within the specified time, cost, and resource constraints, a project manager may use the Manage Project use case.
- To ensure that trained and skilled human resources are available for projects, a resource manager may use the Manage Resource use case.
- To ensure that a project management system is available for a project, a system administrator may use the Administer System use case, which may involve a backup system.

The project management system does not offer any functionality to enable human resources to achieve their goals. Note that such functionality is outside the scope of the system.

**Communicate Associations**

Figure 4-1 shows actors associated with the project management system and Figure 4-3 shows use cases associated with the project management system, but how are actors and use cases related? A specialized type of association, called a communicate association, addresses the question of how actors and use cases are related and which actors participate in or initiate use cases. (Associations are discussed in Chapter 3.)

As discussed in Chapter 2, a communicate association between an actor and a use case indicates that the actor uses the use case; that is, it indicates that the actor communicates with the system and participates in the use case. A use case may have
associations with multiple actors, and an actor may have associations with multiple use cases. A communicate association is shown as a solid-line between an actor and a use case.

Figure 4-4 shows that a project manager participates in managing projects, a resource manager participates in managing resources, and a system administrator and backup system participates in administering the system.

![Figure 4-4. Actors and use cases](image)

A navigation arrow on an association pointing toward a use case indicates that the actor initiates the interaction with the system. Figure 4-4 shows that a project manager initiates the interaction with the project management system to manage projects, and a resource manager initiates the interaction with the project management system to manage resources.

A navigation arrow on an association pointing toward an actor indicates that the system initiates the interaction with the actor. Figure 4-4 shows that the project management system initiates the interaction with the backup system to back up project management data.

Rather than use two arrows when either the system or the actor may initiate an interaction, navigation arrows on both ends of such an association are dropped. Figure 4-4 shows that either a system administrator or the system may initiate an interaction to administer the system. The system administrator might initiate an interaction with the system to back up the data, or, for example, the system might initiate an interaction with the system administrator informing the actor that system resources are low.

Be aware, however, that a lack of navigation arrows may simply result from a modeler choosing not to specify anything about the initiation of an interaction. Thus, with respect to Figure 4-4, you can’t be absolutely certain that either actor can initiate a system administration interaction. It could be that the system administrator only can initiate the interaction, and the UML modeler simply chose not to specify initiation in this one case. It simply depends on the modeling guidelines the modeler is using.
**Dependencies**

A model may have many use cases, so how do we organize the use cases that define what a system should do? And how do we use this information about use cases to determine how best to execute a project while considering how use cases are related to one another, including what some use cases might have in common, and also taking into account use cases that are options of other use cases? Specialized types of dependencies, called include and extend dependencies, address these questions; dependencies are discussed in Chapter 3. The next few sections discuss these specialized types of dependencies.

**Include Dependencies**

Perhaps we wish to log the activities of project managers, resources managers, and system administrators as they interact with the project management system. Figure 4-5 elaborates on the use cases in Figure 4-4 to show that the activities of the project manager, resource managers, and system administrators are logged when they are performing the use cases shown in the diagram. Thus, logging activities are common to these three use cases. We can use an include dependency to address this type of situation by factoring out and reusing common behavior from multiple use cases.

![Figure 4-5. Use cases with common behavior](image)

An include dependency from one use case (called the base use case) to another use case (called the inclusion use case) indicates that the base use case will include or call the inclusion use case. A use case may include multiple use cases, and it may be included in multiple use cases. An include dependency is shown as a dashed arrow from the base use case to the inclusion use case marked with the include keyword. The base use case is responsible for identifying where in its behavior sequence or at which step to include the inclusion use case. This identification is not done in the UML diagram, but rather in the textual description of the base use case.

Figure 4-6 refines Figure 4-5 using include dependencies. The Log Activity use case is common to the Manage Project, Manage Resource, and Administer System use cases, so it is factored out and included by these use cases.
You can use an include dependency when a use case may be common to multiple other use cases and is therefore factored out of the different use cases so that it may be reused. The Log Activity use case in Figure 4-6 is included in the Manage Project, Manage Resource, and Administer System use cases. Consequently, you must analyze and develop that use case before you develop the three use cases that depend on it.

**Extend Dependencies**

Projects are made of activities, and activities are made of tasks. Figure 4-7 elaborates the Manage Project use case in Figure 4-4, and shows that a project manager may manage projects by maintaining the project itself, its activities, or its tasks. Thus, maintaining the project, its activities, and its tasks are options of managing a project. You can use an extend dependency to address this situation by factoring out optional behavior from a use case.

An extend dependency from one use case (called the extension use case) to another use case (called the base use case) indicates that the extension use case will extend (or be inserted into) and augment the base use case. A use case may extend multiple use cases.
cases, and a use case may be extended by multiple use cases. An extend dependency is shown as a dashed arrow from the extension use case to the base use case marked with the extend keyword. The base use case is responsible for identifying at which steps in its behavior sequence the extending use cases may be inserted.

Figure 4-8 refines Figure 4-7 using extend dependencies. The Maintain Project, Maintain Activity, and Maintain Task use cases are options of the Manage Project use case, so Manage Project is factored out and extends those three use cases.

![Figure 4-8. Simple extend dependencies](image)

The location in a base use case at which another behavior sequence may be inserted is called an extension point. Extension points for a use case may be listed in a compartment labeled “Extension Points” where each extension point is shown inside the compartment with an extension-point name followed by a colon followed by a suitable description of the location of the extension point in the use case’s behavior sequence. Locations may be described as being before, after, or in-the-place-of a step in the base use case’s behavior sequence. For example, the Manage Project use case may have a behavior sequence for finding a project on which to work followed by an extension point named Project Selected followed by another behavior. The Project Selected extension point may be described as occurring after a project is found but before it is actually worked on.

An extend dependency is responsible for defining when an extension use case is inserted into the base use case by specifying a condition that must be satisfied for the insertion to occur. The condition may be shown following the extend keyword enclosed within square brackets followed by the extension point name enclosed in parentheses. For example, other use cases may be inserted into the Project Selected extension point just described for the Manage Project use case. Such behavior sequences may include reviewing and updating project information, or selecting a specific version of a project before managing the details of the project in the succeeding behavior sequences.

Figure 4-9 elaborates on the Administer System use case in Figure 4-4 using extend dependencies. It shows that a system administrator is offered two options—starting
up the system or shutting down the system—at the extension point named Administration Functions, which is described as being available on the administration menu of the user interface. Figure 4-9 further shows the following:

![Figure 4-9. Extension points and extend dependencies](image)

- The Startup System use case is available as an option at the Administration Functions extension point of the Administer System use case. The Startup System use case has two extension points named Before and After. The Before extension point is described as occurring before the startup functionality is performed by the system, and the After extension point is described as occurring after the startup functionality is performed by the system. These extension points are used as follows:
  — The Restore Data use case is available as an option at the Before extension point of the Startup System use case. Before starting up the system, the system administrator may restore data from the backup system to the project management system’s database that was previously archived.
  — There are no options described for the After extension point of the Startup System use case.

- The Shutdown System use case is available as an option at the Administration Functions extension point of the Administer System use case. The Shutdown System use case has two extension points, named Before and After. The Before extension point is described as occurring before the shutdown functionality is
performed by the system, and the After extension point is described as occurring after the shutdown functionality is performed by the system. These extension points are used as follows:

- The Backup Data use case is available as an option at the After extension point of the Shutdown System use case. After shutting down the system, the system administrator may back up data from the project management system’s database to the backup system for later retrieval.

- There are no options described for the Before extension point of the Shutdown System use case.

The extension points just described allow us to insert behavior into the Startup System and Shutdown System use cases before or after they perform startup or shutdown processing for the project management system. The extend dependencies reference these extension points to indicate where use cases may be inserted inside one another, and also to indicate the conditions that must be satisfied for such an insertion to occur. Naturally, data is restored before the system is started up and data is backed up after the system is shut down.

Use an extend dependency when a use case is optional to another use case. Because the Maintain Project, Maintain Activity, and Maintain Task use cases extend the Manage Project use case, the Manage Project use case must be developed before the others; otherwise, the other use cases won’t have a use case to extend. Likewise, the Administer System use case must be developed before the Startup System and Shutdown System use cases, Startup System must be developed before Restore Data, and Shutdown System must be developed before Backup Data. However, once Administer System is developed, Startup System and Shutdown System may be developed in parallel or concurrently, because they are not directly related.

**Generalizations**

Actors may be similar in how they use a system; for example, project managers, resource managers, and system administrators may log in and out of our project management system. Use cases may be similar in the functionality provided to users; for example, a project manager may publish a project’s status in two ways: by generating a report to a printer or by generating a web site on a project web server.

Given that there may be similarities between actors and use cases, how do we organize the use cases that define what a system should do? And how do we use the information about similarities between actors and use cases to determine how best to execute a project? Specialized types of generalizations, called actor and use case generalizations, address these questions. Generalizations are introduced and discussed in Chapter 3. The next two sections discuss these specialized types of generalizations.
**Actor Generalizations**

Figure 4-10 shows that project managers, resource managers, and system administrators may log in and out of the project management system. Thus, logging in and out is common to these actors. Actor generalizations address such situations by factoring out and reusing similarities between actors.

Figure 4-10. Similar actors

An actor generalization from a more specific, or specialized, actor to a more general, or generalized, actor indicates that instances of the more specific actor may be substituted for instances of the more general actor. An actor may specialize multiple actors, and an actor may be specialized by multiple actors. An actor generalization between actors is shown as a solid-line path from the more specific actor to the more general actor, with a large hollow triangle at the end of the path connected to the more general actor.

Figure 4-11 refines Figure 4-10 using actor generalizations between actors. A human resource initiates the Login and Logout use cases. Project managers, resource managers, and system administrators are human resources.

Use an actor generalization between actors when one actor is similar to another, but has specific interactions in which it participates or initiates. For example, any human resource may log in and out, but project managers, resources managers, and system administrators make more specialized use of the project management system. Because the Project Manager, Resource Manager, and System Administrator actors are specialized Human Resource actors, they benefit from the use cases in which the Human Resource actor is involved. Therefore, by developing the Login and Logout use cases, we provide the functionality described by those use cases for all the actors of our system.
Use-Case Generalizations

Figure 4-12 shows that a project manager may publish a project’s status in two ways: by generating a report to a printer or by generating a web site on a project web server. Thus, publishing a project’s status and all the processing involved in collecting and preparing the data for publication is common to these use cases. You can use a use-case generalization to address this situation by factoring out and reusing similar behavior from multiple use cases.

A use-case generalization from a more specific, or specialized, use case to a more general, or generalized, use case indicates that the more specific use case receives or inherits the actors, behavior sequences, and extension points of the more general use case, and that instances of the more specific use case may be substituted for instances of the more general use case. The specific use case may include other actors, define new behavior sequences and extension points, and modify or specialize the behavior sequences it receives or inherits. A use case may specialize multiple use cases, and a use case may be specialized by multiple use cases. A use-case generalization between use cases is shown as a solid-line path from the more specific use case to the more general use case, with a large hollow triangle at the end of the path connected to the more general use case.
Figure 4-13 refines Figure 4-12 using use-case generalization between use cases. The Generate Report and Generate Website use cases receive the Project Manager actor, behavior sequences, and extension points of the Publish Status use case.

![Use-case generalizations](image_url)

You can use a use-case generalization between use cases when a more specific use case is similar to a more general use case but involves other actors or has specialized behavior. For example, a project manager may publish a project’s status using a report or a web site, but a printer is involved only if a report is generated, and a project web server is involved only if a web site is generated. Because the Generate Report and Generate Website use cases specialize the Publish Status use case, Publish Status must be developed before the other use cases; otherwise, the other use cases won’t have a use case to specialize.

It is important to understand the difference between include and extend dependencies and use-case generalization. An inclusion use case does not have knowledge of the base use case that includes it, an extension use case does not have knowledge of the base use case that it extends, and the Maintain Activity use case in Figure 4-8 has no knowledge of the use cases that it extends, so they can’t involve the actors of the base use case in their behavior sequences. For example, the Log Activity use case in Figure 4-6 has no knowledge of the use cases that include it. However, a more specific use case receives or inherits the actors, behavior sequences, and extension points of its more general use case, so it can involve the actors of the more general use case in its behavior sequence. For example, the Generate Report use case in Figure 4-13 has knowledge of the Publish Status use case and may involve the Project Manager actor in its behavior sequence. An inclusion use case must be developed before its base use cases, an extension use case must be developed after its base use cases, and a more specific use case must be developed after its more general use cases.
Exercises

1. Describe Figure 4-14: identify actors and use cases, and describe the relationships among actors and use cases.

![Use-case diagram for the project management system](image)

Figure 4-14. Use-case diagram for the project management system

2. Update the diagram shown in Figure 4-14 stepwise to show the following details. After each step, check your answers against the solutions shown in Appendix B:
a. A human resource may manage his professional development plan. When managing his professional development plan, the human resource may manage his profile. In managing his professional development plan, he will have access to a training schedule provided by a training database, and all the information pertaining to his professional development plan will be stored in a professional development plan database. Both databases are not part of the project management system.

b. When a resource manager manages resources, she will have access to resource functions that are provided on a resource menu. One option is to manage a resource’s profile. This is the same functionality used by human resources to manage their profiles.

c. A project manager or system administrator, acting as an email user, may send email using an email system that is not constructed as part of the development effort (it may have been purchased). When sending email, the manager may select a secure option wherein his whole interaction with the system is secured using a purchased encryption engine. Note that this is not simply encrypting the sent email message, but encrypting the entire interaction as the email is composed and sent.

d. An email system may be used to receive email messages. When a message is received, the email user is informed. Likewise, an email user may query the system to check whether new messages have arrived, depending on how often her system is set up to check for new messages.

e. When sending or receiving email, the system logs the transaction. This functionality for logging transactions may be used by other use cases in the future.

3. Given Figure 4-14 and the solution to question 2, describe the general order in which the use cases shown in the resulting diagram must be developed; that is, what use cases must be developed before other use cases (independent of users prioritizing the use cases)?
Get a robust collection of new developer resources -- free!

If you're a developer who wants to stay on top of the latest technology, then be sure to sign up to receive the latest Developer PowerPack from IBM Rational. Get a robust collection of resources: market-leading evaluation software, technical articles, extensive artifact libraries, Webinars, expert presentations, reference posters, and software demos. View what you want. Explore your technical interests. Click here.
IBM Rational software joins IBM training portfolio

IBM's acquisition of Rational Software has expanded and complemented the IBM software portfolio to create five brands in the IBM Software Group. These include DB2 Information Management, Lotus, Tivoli, WebSphere, and now Rational software. As a leading provider of application development software, Rational supports what IBM does best -- providing infrastructure software and tools to help customers create open, complete software development environments for building on demand e-business and technical software products and systems.

There are many ways to learn concepts, develop skills, and gain the specialized knowledge needed to master the art of software development. IBM IT Education Services is now pleased to add training for the IBM Rational Suite to our portfolio to help meet all of your skill-building needs.

Offerings mapped to the Rational development tool areas, supporting the phases of the AD lifecycle:

- Automated Testing
- Configuration Management
- Rational Unified Process
- Requirements and Analysis
- Visual Modeling and Development

Why come to IBM for training?

IBM IT Education Services provides you with a far-reaching, flexible portfolio of technical training and education services to support mission-critical IT skills.

- Single source for global coverage
- Comprehensive portfolio of training solutions for IBM and non-IBM products
• Expertise in leading-edge IT subject areas

• Highly experienced technical trainers with subject matter expertise

• Public classes at convenient locations nationwide

• Face-to-face classes at our site or yours

• Collaborative technologies

• E-learning and blended learning options

• Flexible pricing

IBM offers a comprehensive training curriculum to help you build and maintain your Rational software skills. The courses are designed and taught by experienced course developers with close ties to Rational's product developers, helping to ensure that classes contain the latest information, including tips, techniques, and trends. Job-specific training paths take the guesswork out of developing a training plan. Each training path visually outlines the logical sequence of courses necessary for you to build and maintain proficiency in a particular job function or product area. You can receive Rational training from one source. Most of our classroom offerings feature the advantage of hands-on labs, which provide a risk-free practice environment. In addition, by incorporating the power of networking and Web-based technology, we can make cost-effective, flexible learning options available at your desktop.

Instructor-led training

IT Education Services offers a variety of Rational software classroom training solutions. Most courses include hands-on labs or exercises. Classes are available on a public or private basis, at our site or yours.

Education at your desktop

Can't take time away from work? Limited or no training budget? Our Web-based training provides anytime access to courses teaching foundational knowledge and basic concepts of Rational tools and methodologies. You can complete courses according to your own schedule, as they are available 24 hours a day, 7 days a week over the Web. This training is available on Rational Developer Network, a customer-only learning and support Web site.

Visit our frequently updated Web site for detailed information, including:

• Complete course descriptions
Course schedules and locations

New course information

Up-to-date training paths

News about other training offerings

Links to other useful and informative Web sites

To enroll in IBM Rational training, visit ibm.com/training/us/catalog/rational or call 1-800-IBM-TEACH (1-800-426-8322).
The privilege and responsibility of software development

by Grady Booch
Chief Scientist
Rational Software

As developers, we've all had our share of bad days. Days in which our operating systems, networks, workstations, and coworkers conspire against us to suck all productivity out of the air. Days in which our bosses or their bosses or our customers hammer us for errors done or for functions left undone. Days that turn into nights and back into days again as we chase some elusive gnome from our system.

These are the days of living as a netslave, a microserf.

After abiding such days during which we labor to build artifacts that live in the realm of nanoseconds, sometimes we long for a life with "no unit of time shorter than a season."

Still, most of us come to the profession of software development deliberately, typically because we like to create things from pure thought, things that give life to our machines and that matter to our organizations, perhaps even to the world. For others, software creeps up behind us and grabs us by the neck; although we may secure an uneasy truce with it even though we may not be code warriors, we still require some degree of development skills so that we can wrestle that software to the ground and direct it to carry out our will. Either way, as an intentional or as an accidental developer, we build things that the rest of the world needs and uses and yet are often invisible to them.

For this reason, it is both a privilege as well as a responsibility to be a software developer.

It is a privilege because -- in spite of some inevitable dark days -- we
collectively are given the opportunity to create things that matter: to individuals, to teams, to organizations, to countries, to our civilization. We have the honor of delivering the stuff of pure intellectual effort that can heal, serve, entertain, connect, and liberate, freeing the human spirit to pursue those activities that are purely and uniquely human.

At the same time, we have a deep responsibility. Because individuals and organizations depend on the artifacts we create, we have an obligation to deliver systems of quality in a manner that applies scarce human and computing resources intentionally and wisely. This is why we hurt when our projects fail, not only because each failure represents our inability to deliver real value, but also because life is too short to spend precious time on constructing bad software that no one wants, needs, or will ever use. As professionals, we also have a moral responsibility: Do we choose to labor on a system that we know will fail or that may steal from a person their time, their liberty, or their life? These are questions that do not have a technical answer, but rather are ones that must be consciously weighed by our individual belief system as we deploy technology to the world.

Thus, software development is ultimately a human activity, not only because it emanates from the human intellect, but also because it requires the cooperative activity of others to make it real.

As professionals, we therefore constantly seek better ways to deliver quality software that matters, simply because our task is too complex to squander our time and our energy. This is why we look at successful projects and analyze why they were successful and similarly look at failed projects so that we may learn from their mistakes. We then codify all these lessons learned in the best practices and processes that constitute our industry's tribal memory, such as found in the RUP. For the same reason, we agree upon common notations such as the UML that help us communicate and reason about our systems.

In this pile of best practices, object-oriented development stands out as a proven technology that has been used to successfully build and deliver a multitude of complex software-intensive systems in a variety of problem domains.

Still, the demand for software continues to rise at a staggering rate. The ever-growing capabilities of our hardware, together with an increasing social awareness and economic value of the utility of computers, create tremendous pressure to automate systems of even greater complexity. The fundamental value of object-oriented development, with its well-defined notation and process, is that it releases the human spirit to focus its creative energies upon the truly demanding parts in the crafting of useful, quality systems in a timely, predictable, and repeatable fashion.

(This article is excerpted from the forthcoming third edition of *Object-Oriented Analysis and Design with Applications*, from Addison-Wesley).

Notes


*NOTE:* This article was originally published on Rational Developer Network, the learning and support channel for the IBM Rational customer community. If you are an IBM Rational customer and have not already registered for your free membership, please go to www.rational.net

For more information on the products or services discussed in this article, please click [here](#) and follow the instructions provided. Thank you!
Migrating a J2EE project from IBM Rational Rose to IBM Rational XDE Developer v2003

by Steven Franklin

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

If you've been using IBM® Rational Rose® in a J2EETM (JavaTM 2 Platform, Enterprise Edition) project and have been wondering whether and how to switch to IBM® Rational® XDE™ Developer (known until v2003 as simply Rational XDE), this five-part series of articles should help you decide. I'll use the same approach as in an earlier article series I wrote, and use a fictional project (the ASDI project) as the basis for showing how to make the transition from Rational Rose to Rational XDE Developer. I'll assume you have some previous knowledge of Rational XDE, and I'll update you on some of the new features that have been introduced in Rational XDE Developer v2003 -- Java Platform Edition. Although familiarity with my earlier articles isn't required, you may find them helpful (for example, for background on Rational XDE); see "Earlier Articles" for more information.

Of all the feedback I received on the ASDI project articles and my later articles on Rational XDE, the most popular questions were "Should I switch from Rational Rose to Rational XDE?" and "How do I transition my team from Rational Rose to Rational XDE?" This series will address those questions.

Concern over moving to a new tool
The most popular series to date has been the ten-part "Applying Rational Tools to a Simple J2EE-Based Project," which shows how a fictional J2EE project could start at a basic level of process and evolve into a more efficient and mature team with the help of the IBM® Rational Unified Process® (RUP®) and other Rational software tools. Although the sample project (for a company called Audiophile Speaker Design, Inc., or ASDI) is fictional, it's based on my personal experiences and observations. My aim in writing these articles was to:

- show how the RUP and other Rational software tools could be used by a talented team on a fast-changing project
- share some of the lessons learned by myself and others in integrating the tools into actual projects

Since writing that series, I've written two more, both on the subject of Rational XDE ("First Encounters with Rational XDE Professional" and "Data Modeling in Rational XDE Release 2"), in which I discuss some of its interesting features and its strengths and weaknesses as an integrated environment.

This set of articles will continue with the project where it stands at the end of the earlier ASDI article series: the team has released their product and is now maintaining the software. They're facing a potentially major revision of the project, and they also want to sell their product to other customers. The tools and process they used during implementation were acceptable, but now they feel there are some improvements that can be made. They turn to Rational XDE Developer as a way to improve the integration in their development environment. I'll show how they make that transition with a legacy model and system in place. I'll discuss their concerns, mistakes, and benefits realized as a result of using Rational XDE Developer, and I'll look at the effort required, the features lost or gained, and the impact of the changes on the team.

After reading these articles, you should be able to:

- understand Rational XDE Developer's role in different types of projects
- see how Rational XDE Developer and Rational Rose differ and how they can coexist
be comfortable with the process of migrating from Rational Rose to Rational XDE Developer

understand how Rational XDE Developer can integrate with other tools already in use on your project

Below is a roadmap to this series. Each article will include this list, linking to the other articles as they become available.

- Part 1: introduction to the team and the challenges they face
- Part 2: how Rational XDE Developer fits into the evolving project, and its impact on process; the transition from Rational Rose to Rational XDE Developer
- Part 3: Rational XDE Developer's collaborative and usability enhancements
- Part 4: Rational XDE Developer's advanced features and how they benefit the ASDI project
- Part 5: a review of lessons learned; how to tailor Rational XDE Developer usage to specific types of projects; a summary of the article series

A final note: Keep in mind that the ASDI project is fictional and that real-world requirements will be different. To determine whether Rational XDE Developer fits your needs, try downloading and using an evaluation copy of it. Only by using the tool can you really get a feel for its value; these articles express only one perspective.

Revisiting and Evaluating the ASDI Project

The earlier series of articles, "Applying Rational Tools to a Simple J2EE-Based Project," lays out the fictional premise that we're a software company, Lookoff Technologies Incorporated, that was hired by Audiophile Speaker Design, Inc. (ASDI) to meet their burgeoning IT requirements. ASDI wanted us to work with them to reengineer their legacy system. Although they were a very talented team, their process was rigid and not conducive to the rapid changes and the reengineering task that they were about to undertake.

The ASDI project was carried out by a small team and was limited by both schedule and budget. Quality had to be top-notch, since this system would go into production as the supply chain management software for this speaker design company.

The system went into production as hoped, and ASDI was very pleased with the finished product. We then solicited a follow-on support and maintenance contract from them. As ASDI's business grew, they required more maintenance and support from us. Eventually, it reached the point where we had two developers on the ASDI site supporting all IT operations for the system and network. Our customer's speaker business was very successful, and online operations increased their revenue, largely through
We responded as quickly as we could when the customer made software change requests. However, the process that had led us through a successful 60-person-month development project was not well suited to our on-site maintenance and enhancement work. With such a small development team, our analysis and design artifacts were getting out of date, and sometimes we were making quick changes without understanding the full impact, consequently introducing bugs and instabilities into the production system.

It was important to ensure that the production system ran smoothly and reliably, since it was a critical portion of the revenue chain for the company; however, the customer was making frequent, substantial change requests. With increasingly outdated documentation, it was becoming more and more difficult to make changes. There was some concern that our mature framework was evolving into a system that would be hard to maintain.

Having a cumbersome software product worked against our long-term strategy to market the product to other small businesses in the high-tech industry. Although tailored to a speaker design company, the system was actually a flexible product that could be of great interest to small- and mid-sized inventory-based companies.

We set out to identify the problems in our current process by looking for areas in which the tools were impeding our productivity. We needed to figure out just where our design and development time was being spent. We knew we had to keep the design model up to date, but there were clearly some problems that were keeping that from happening.

Although we knew our problems couldn’t be solved only through tools, we hoped there were some tools that could at least make our development and model maintenance easier. When we learned about Rational XDE Developer, we thought it justified further investigation. We had already significantly tailored our environment with Rose Extensibility Interface (REI) scripts, and we knew the Rational software tools (namely Rational Rose, IBM® Rational® PurifyPlus, and IBM® Rational® ClearCase®) inside out. Although nervous about moving to a new integrated tool, we decided to weigh the tradeoffs.

**Analyzing the ASDI Development Process**

During the initial development contract, the ASDI project had been run according to the Rational Unified Process (depicted in Figure 1). This process worked very well, and led to the holy grail of software delivery: we came in under budget and ahead of schedule. Moreover, the customer was very happy with the end product.
During implementation, the project followed the steps shown in Figure 2. The highlighted steps are those that we expected to focus on as we moved into maintenance and enhancement. In reality, however, our maintenance process ended up focusing heavily on the "Update Code" and "Mini-Builds" tasks, with mini-builds going straight to production!

Our carefully planned process seemed to be weakening as a result of schedule pressures and the fact that although each change was "just a small change," there were a lot of them. With better planning, we could have implemented the changes more efficiently and effectively.

It may seem as if our original delivered product lacked sufficient analysis.
This could be, but I believe the main reason for our weakening process was the rapid growth of our customer's business combined with their desire to add significant new functionality. We had deferred implementing certain features during the original contract due to a lack of budget or priority, and some of these features were now being addressed under the maintenance phase. The customer's "wish list" was much longer than the current budget would allow, and we also had ambitious goals for our product. It was time to tighten up the process and address some of the technological deficiencies that we had identified.

We decided to dedicate some research and development to refactoring the system, to ensure that it could easily be tailored and internationalized for different customers and markets. Refactoring is the task of reorganizing, reducing, and removing code to improve cohesion and minimize coupling. Basically, it involves cleaning up the code to adhere to good functional or object-oriented principles. (You can find out more about it at Martin Fowler's site.) Refactoring is an important skill in software engineering, particularly as more of us inherit legacy systems that require substantial improvements.

So, in addition to our existing maintenance team (again, with two developers -- one senior and one intermediate), we proposed having an R&D team with additional developers for doing refactoring and more; Figure 3 shows this organization.

![Figure 3: Maintenance team and proposed R&D team](image)

We established the R&D team and gave it these specific goals:

- Assess the existing ASDI code and model.
- Improve the quality of the code.
- Work out an arrangement that allows concurrent access to the source repository both off-site at ASDI and on-site at the R&D lab.
- Factor out all customer-specific business logic and constraints so
that we can easily tailor this system to different customers.

- Internationalize the software to support non-English-speaking customers.

**The Bad News**

The R&D team first reviewed the project with the original development team members and the maintenance team. A number of deficiencies came to light as the result of these early conversations.

- **Too many tools** -- The team was finding it cumbersome to deal with so many tools: database modeling software, Rational Rose, Rational PurifyPlus, a third-party Java IDE, IBM® WebSphere® Studio's administration and deployment tools, and Rational ClearCase. Although it was possible to integrate some of these tools, the developers still found that they had to switch frequently between tools to do their job.

- **Outdated design** -- The team found that the RTE was not convenient; right or wrong, they were not incorporating every last method signature update into the design. Over time, the implemented system was departing further and further away from the last design snapshot. Without a good design and associated impact analysis tools, it was becoming difficult to judge the complexity and merits of future requirements changes. Furthermore, the R&D team was concerned that their refactoring work would be difficult without an up-to-date system model.

- **Steep learning curve** -- New team members were overwhelmed with having to learn Rational Rose, the Unified Modeling Language (UML), the development environment, the business domain, and the system we had built. We thought we had scheduled enough time with a two-week hand-off, but that proved to be insufficient.

- **No time for R&D** -- The team also wanted to try some new technological and process advances, but there never seemed to be enough time or budget for that. With a long list of small fixes always in the queue, it was difficult to justify time for prototypes and software evaluations.

**The Good News**

In its discussions with the development and maintenance team members, the R&D team also heard about the positive aspects of the ASDI project -- but even then there were some negative observations.

- **ClearCase Windows integration** -- The team was pleased with the ClearCase Windows integration, and felt that this level of integration needed to be extended to other tools they were using.

- **Rational Rose and REI** -- Rational Rose, particularly its script customization through REI, was a definite plus. The maintenance
team used REI scripts to automate a number of tasks (although they used them less and less as they fell behind in their model updates).

- **Rational SoDA** -- The development team was very pleased with IBM® Rational® SoDA®. The maintenance team, however, didn't know SoDA well enough to be able to quickly define new reports tailored for their maintenance activities, and this further reduced their need to maintain and use the model.

- **RUP** -- The RUP had been very effective during implementation. The maintenance team recognized the value of the RUP during the maintenance phase; however, for reasons that were not yet understood, they were not following RUP principles and practicing good model management habits.

- **Early prototyping** -- The development team felt that their early prototyping efforts had been key to the success of the ASDI project. The maintenance team wanted to continue these efforts to ensure that the system evolved with the company. Lookoff was willing to take the plunge and invest some R&D to enhance the product.

**Wish List for R&D**

In the end, the maintenance team lead came up with the following wish list for the R&D team:

- Simplify synchronization of the model with the system as changes happen.
- Find a strong IDE that ties in with the model and the existing source change management system (ClearCase).
- Reduce the number of required tools.
- Support multiple partitioned models to manage the larger project.
- Improve reuse across projects and developers by sharing code snippets, patterns, and templates.

**Identifying Room for Improvement**

The R&D team hoped that Rational XDE Developer could address a number of the issues they faced, and they wanted to explore the tool further before making any recommendations. Since the product was quite new at the time, they found the "First Encounters with Rational XDE Professional" articles to be helpful as an overview. They listed their major requirements and concerns related to the ASDI project, and tried to identify those that could be met by Rational XDE Developer (Table 1).

**Table 1: Matching Rational XDE Developer capabilities to project deficiencies**
<table>
<thead>
<tr>
<th>Concern</th>
<th>Rational XDE Developer capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronizing the design with every little code change took a lot of effort.</td>
<td>Rational XDE Developer automates synchronization, thereby ensuring that code updates are worked into the design and that design updates are applied to the code.</td>
</tr>
<tr>
<td>The analysis artifacts were outdated.</td>
<td>Although updates to analysis cannot be automated, the analysis model is integrated into the environment, so it's easier to make changes to it. Rational XDE Developer's support for multiple models (with integration and traceability between models) enables concurrent model management.</td>
</tr>
<tr>
<td>ClearCase integration was adequate within Windows Explorer, but it would be better to have it integrated with the team's third-party IDE.</td>
<td>Rational XDE Developer integrates tightly with ClearCase, making ClearCase a transparent part of the IDE while providing change management within the tool.</td>
</tr>
<tr>
<td>The data modeling tool was a separate application. The development team would sometimes modify SQL or the schema directly rather than go to the single PC that ran the database modeling software.</td>
<td>Although Rational Rose Enterprise has data modeling, for cost reasons the project ended up purchasing only Rational Rose Professional J Edition (which also provides some data modeling functionality). Rational XDE Developer includes database modeling functionality within it, avoiding the need for a separate tool.</td>
</tr>
<tr>
<td>Rational Rose capabilities were more complex than the team required.</td>
<td>Rational Rose has significant functionality and comprehensive UML support; Rational XDE Developer provides similar modeling capability, but the model can be set up so that only a subset of the capabilities is visible to the developer.</td>
</tr>
</tbody>
</table>
The system needed a significant amount of refactoring.

Built-in refactoring features in Rational XDE Developer and WebSphere Studio Application Developer make it easy to rename entities, move and extract code to and from methods, encapsulate field access, pull methods up into a parent class, and more.

Useful code snippets and patterns had emerged during implementation but had not transferred properly to the maintenance team.

Rational Rose has some capabilities in this area, but Rational XDE Developer further tightens up the integration of code templates and patterns.

Summary

We knew we had let our process lapse, and we were also not watching for tool advances as closely as we could have. It's always tempting to let things slide like this at the tail end of a development project, but if the product is important to you or your customer, it can't be excused. In our case, the product was important to us and our customer. We foresaw that we could market the system to multiple customers, so it had to be easy to maintain and adaptable to change.

Our existing approach was satisfactory, but our standards were now very high, and we wanted to be sure to maintain our software quality. Maintenance had slowly eaten away at the coupling and cohesion of the design, and the system was no longer the same as the one captured in the latest system model. We wanted to fix that, and we saw that a combination of technology and process could help us.

Rational XDE Developer was one tool that we saw as a potential candidate. We hoped that migrating to it would be safe, since we were moving from one tool to another from the same vendor. We also hoped that by using Rational XDE Developer we would decrease the number of tools and the complexity of mundane tasks.

Part 2 will look at Rational XDE Developer in more detail and show how a project can smoothly make the transition from Rational Rose to Rational XDE Developer after the project is underway.

References

Articles by the author:

- "Applying Rational Tools to a Simple J2EE-Based Project" (a ten-part series)
- "First Encounters with Rational XDE Professional" (a six-part series)
● "Data Modeling in Rational XDE Release 2" (a two-part series)

References available via Rational Developer Network

---

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