Cover Story

Software development trends: Pressures and opportunities in the financial sector

Inside:
- Agile RUP for non-OO projects
- UML basics: The activity diagram
- Principles & techniques for IBM Rational ClearCase performance
- Reducing risk with the RUP Phase Plan

enter issue | mission statement | archives | subscriptions
Editor's notes:

**From the detailed to the general**

The human ability to discover general patterns by observing particular details is one big reason *The Rational Edge* occasionally focuses on software development trends in a certain industry -- telecommunications or consumer electronics, for example. Naturally, most of our readers make their living elsewhere. But we believe that a closer look at creative IT efforts in one industry can help software development planners everywhere, especially those who are wrestling with the demands of today's difficult economy.

So this month we turn to the financial services sector, where software development is beginning to play a more visible role in business success. As TowerGroup's Guillermo Kopp notes: "No one in a financial institution will be saying, 'Oh, yes, IT is busy doing projects, but nothing customers will be able to see.' Instead, you'll see increased IT spending on things like integration for all customer-facing channels and cost center automation; and, of course, electronic delivery systems like ATMs and the Web will also still play a role." Join him and IBM finance industry sales director Gary Napolitano for a **wide-ranging tour of trends in this important industry**.

And if you're wondering how vital IT capability is to financial institutions these days, here's a mental keyword search item for you: "BASEL II."

Somewhat coincidentally, the development of a billing system serves as Gary K. Evans's case study in "[Agile RUP for non-object-oriented projects](http://www.therationaledge.com/content/sep_03/index.jsp)," which explores an "object-organized" approach to COBOL and C projects. The point is, how can IT shops that support COBOL or C architectures make use of IBM Rational Unified Process,® or RUP,® and gain the same advantages that teams working with OO languages enjoy? Gary outlines a UML-based technique that you should email to your colleagues working in the public or financial sector. And speaking of the UML, Donald Bell is back with the next in his series of articles on the major Unified Modeling Language diagrams, "[The activity diagram](http://www.therationaledge.com/content/sep_03/index.jsp)." Learn about decision and
merge points, synch states, and other cool symbols.

Whether you're a seasoned software development strategist, or a relative newcomer to solutions for the application development lifecycle, there's something directly relevant to your role in this issue. Murray Cantor continues his discourse on RUP for systems engineering with a study of systems architecture. Mark Aked offers software architects using RUP a disciplined approach to RUP's Phase Plan that takes risk into consideration. Tom Milligan concludes his series on IBM Rational ClearCase® techniques with a focus on improving performance. And, the IBM Rational Rapid Developer® product management team has a new paper that describes how this new product -- for practitioners at all skill levels -- supports global e-business requirements.

Are you setting up a network infrastructure to support Internet applications? Then you'll want to check out Cisco Internet Architecture Essentials Self-Study Guide: Cisco Internet Solutions Specialist, reviewed in our Rational Reader section. In his constant pursuit of simplifying things as complex as the subject of that book, Grady Booch gives us his thoughts on the connection between elegance and simplicity in this month's Franklin's Kite.

Happy iterations,

Mike Perrow
Editor-in-Chief
UML basics

Part II: The activity diagram

by Donald Bell
IBM Global Services

In June 2003, The Rational Edge introduced a new article series by Donald Bell, IBM Global Services, called UML basics. The purpose of this series is to help readers become familiar with the major diagrams that compose much of the UML. Part I offered a general overview of these diagrams; this month, we continue the series with a close look at the activity diagram, including this diagram's complete UML v1.4 notation set.

The activity diagram's purpose

The purpose of the activity diagram is to model the procedural flow of actions that are part of a larger activity. In projects in which use cases are present, activity diagrams can model a specific use case at a more detailed level. However, activity diagrams can be used independently of use cases for modeling a business-level function, such as buying a concert ticket or registering for a college class. Activity diagrams can also be used to model system-level functions, such as how a ticket reservation data mart populates a corporate sales system's data warehouse.

Because it models procedural flow, the activity diagram focuses on the action sequence of execution and the conditions that trigger or guard those actions. The activity diagram is also focused only on the activity's internal actions and not on the actions that call the activity in their process flow or that trigger the activity according to some event (e.g., it's 12:30 on April 13th, and Green Day tickets are now on sale for the group's
summer tour).

Although UML sequence diagrams can portray the same information as activity diagrams, I personally find activity diagrams best for modeling business-level functions. This is because activity diagrams show all potential sequence flows in an activity, whereas a sequence diagram typically shows only one flow of an activity. In addition, business managers and business process personnel seem to prefer activity diagrams over sequence diagrams -- an activity diagram is less "techie" in appearance, and therefore less intimidating to business people. Besides, business managers are used to seeing flow diagrams, so the "look" of an activity diagram is familiar.

The notation

The activity diagram's notation is very similar to that of a statechart diagram. In fact, according to the UML specification, an activity diagram is a variation of a statechart diagram. So if you are already familiar with statechart diagrams, you will have a leg up on understanding the activity diagram's notation, and much of the discussion below will be review for you.

The basics

First, let's consider the action element in an activity diagram, whose official UML name is action state. In my experience, people rarely if ever call it an action state; usually they call it either action or activity. In this article, I will always refer to it as action and will use the term activity only to refer to the whole task being modeled by the activity diagram. This distinction will make my explanations easier to understand.

An action is indicated on the activity diagram by a "capsule" shape -- a rectangular object with semicircular left and right ends (see Figure 1). The text inside it indicates the action (e.g., Customer Calls Ticket Office or Registration Office Opens).

![Figure 1: A sample action that is part of an activity diagram.](image)

Because activity diagrams show a sequence of actions, they must indicate the starting point of the sequence. The official UML name for the starting point on the activity diagram is initial state, and it is the point at which you begin reading the action sequence. The initial state is drawn as a solid circle with a transition line (arrow) that connects it to the first action in the activity's sequence of actions. Figure 2 shows what an activity diagram's initial state looks like. Although the UML specification does not prescribe the location of the initial state on the activity diagram, it is usually easiest to place the first action at the top left corner of your diagram.
Figure 2: The initial state clearly shows the starting point for the action sequence within an activity diagram.

It is important to note that there can be only one initial state on an activity diagram and only one transition line connecting the initial state to an action. Although it may seem obvious that an activity can have only one initial state, there are certain circumstances -- namely, the commencement of asynchronous action sequences -- that may suggest that a new initial state should be indicated in the activity diagram. UML does not allow this. Figure 3 offers an example of an incorrect activity diagram, because the initial state has two transition lines that point to two activities.

Figure 3: Incorrect rendering of an initial state within an activity diagram. The initial state can indicate only ONE action.

With arrows indicating direction, the transition lines on an activity diagram show the sequential flow of actions in the modeled activity. The arrow will always point to the next action in the activity's sequence. Figure 4 shows a complete activity diagram, modeling how a customer books a concert ticket.
Figure 4: A complete activity diagram makes the sequence of actions easy to understand.

The sample activity diagram in Figure 4 documents the activity "Booking a Concert Ticket," with actions in the following order:

1. Customer calls ticket office.
2. Ticket rep asks what event person wants tickets for.
4. Ticket rep tells customer available seats and prices.
6. Ticket rep reserves seats.
7. Ticket rep asks for credit card and billing address.
8. Customer gives requested information.
9. Ticket rep charges credit card.
10. Ticket rep mails tickets.
The above action order is clear from the diagram, because the diagram shows an initial state (starting point), and from that point one can follow the transition lines as they connect the activity's actions.

The activity's flow terminates when the transition line of the last action in the sequence connects to a "final state" symbol, which is a bullseye (a circle surrounding a smaller solid circle). As shown in Figure 4, the action "Ticket Rep Mails Tickets" is connected to a final state symbol, indicating that the activity's action sequence has reached its end. Every activity diagram should have at least one final state symbol; otherwise, readers will be unclear about where the action sequence ends, or perhaps assume that the activity diagram is still a work in progress.

It is possible for an activity diagram to show multiple final states. Unlike initial state symbols, of which there can be only one on an activity diagram, final state symbols can represent the termination of one of many branches in the logic -- in other words, the activity may terminate in different manners.

Beyond the basics

We have covered the basic notation elements of an activity diagram, but there are still more notation elements that can be placed on this type of diagram. Although the Figure 4 diagram is technically complete, the activity modeled in Figure 4 is very simplistic. Typically, activities modeled for real software development projects include decision points that control what actions take place. And sometimes activities have parallel actions.

Decision points

Typically, decisions need to be made throughout an activity, depending on the outcome of a specific prior action. In creating the activity diagram for such cases, you might need to model two or more different sequences of actions. For example, when I order Chinese food for delivery, I call the Chinese food delivery guy, give him my phone number, and his computer will automatically display my address if I've ordered food before. But if I'm a new customer calling for the first time, he must get my address before he takes my order.

The UML specification provides two ways to model decisions like this.

The first way is to show a single transition line coming out of an action and connecting to a decision point. The UML specification name for a decision point is decision, and it is drawn as a diamond on an activity diagram. Since a decision will have at least two different outcomes, the decision symbol will have multiple transition lines connecting to different actions. Figure 5 shows a fragment of a sample activity diagram with a decision.
Figure 5: A decision point models a choice that must be made within the sequence of actions.²

As shown in Figure 5, each transition line involved in a decision point must be labeled with text above it to indicate "guard conditions," commonly abbreviated as guards.

Guard condition text is always placed in brackets -- for example, [guard condition text]. A guard condition explicitly tells when to follow a transition line to the next action. According to the decision point shown in Figure 5, a bartender (user) only needs to "make sure the customer is at least 21 years old" when the customer orders an alcoholic drink. If the customer orders any other type of drink (the "else" condition), then the bartender simply gets the drink for the customer. The [else] guard is commonly used in activity diagrams to mean "if none of the other guarded transition lines matches the actual condition," then follow the [else] transition line.

Merge points

Sometimes the procedural flow from one decision path may connect back to another decision path, as shown in Figure 6 at the "Customer's Age ≥ 21" condition. In these cases, we connect two or more action paths together using the same diamond icon with multiple paths pointing to it, but with only one transition line coming out of it. This does not indicate a decision point, but rather a merge.

Figure 6 shows the same decision as in Figure 5, but Figure 6 expands the activity diagram. Performing a check of the customer's age leads the user to a second decision: If the customer is younger than 21, the bartender must tell the customer to order another non-alcoholic drink, which takes our sequence back to the action "Customer Orders Drink." However, if the customer is 21 years old or older, then our action sequence takes us to the same action the bartender would follow if the person had ordered a non-alcoholic drink: "Get Drink For Customer."
An alternative approach

The second approach to modeling decisions is to have multiple transition lines coming out of an action, as in Figure 7. If this method is used, then each transition line coming out of the action must have a guard label above it, as with decisions (i.e., the diamond symbols) in the first approach. All the rules that apply to the decision symbol apply to decisions that are modeled out of an action. Personally, I do not recommend this approach, because I prefer a visual queue of the decision. Nevertheless, the UML 1.4 specification allows this approach, as shown in Figure 7, and I mention it here for the sake of completeness.

Figure 6: A partial activity diagram, showing two decision points ("Drink is alcoholic" and "Customer's age < 21") and one merge ("else" and "Customer's age >= 21")

Figure 7: Although I do not recommend this approach, the UML 1.4 specification allows decisions to be modeled as actions with guard conditions.

If you choose this second approach, you must have multiple action sequences merge into a single sequence. To do this, you connect the last transition lines in the specific sequence to the action at which the
sequence becomes one again. In Figure 8, this is illustrated with the transition line from "Tell customer to order a non-alcoholic drink" returning to the action "Customer orders drink."

**Figure 8: The second approach to modeling decisions**

**Synch states for asynchronous actions**

When modeling activities, you sometimes need to show that certain action sequences can be done in parallel, or asynchronously. A special notation element allows you to show parallel action sequences, or *synch states*, as they are officially named, since they indicate the synchronization status of the flow of activity. Synch states allow the forking and joining of execution threads. To be clear, a synch state that forks actions into two or more threads represents a de-synchronizing of the flow (asynchronous actions), and a synch state that joins actions back together represents a return to synchronized flow. A synch state is drawn as a thick, solid line with transition lines coming into it from the left (usually) and out of it on the right (usually). To draw a synch state that forks the action sequence into multiple threads, first connect a transition line from the action preceding the parallel sequence to the synch state. Then draw two transition lines coming out of the synch state, each connecting to its own action. Figure 9 is an activity diagram fragment that shows the forking of execution modeled.
Figure 9: A thick, solid line indicates a synch state, allowing two or more action sequences to proceed in parallel.

In the example shown in Figure 9, after the action "Receive Order" is completed, two threads are kicked off in parallel. This allows the system to process both the "Verify Ordered Products Are In Stock" and the "Verify Customer Has Available Credit" actions at the same time.

When you fork execution into multiple threads, typically you have to rejoin them at some point for later processing. Therefore, the synch state element is also used to denote multiple threads joining back together into a single thread. Figure 10 shows an activity diagram fragment that has two threads joining into one.

Figure 10: When parallel action sequences terminate, a synch state (thick line) is used to indicate that the multiple threads are joined back into a single thread.

In Figure 10, the "Verify Ordered Products Are In Stock" and the "Verify Customer Has Available Credit" actions shown in Figure 9 have completed processing, and the "Accept Order" action is processed. Note that the single transition line coming out of the synch state means there is now only one thread of execution.

A synch state can also be used as a synchronization point in an activity's overall action execution. In this case, it models how separate threads are forced to join before further execution can proceed. Figure 11 shows an activity diagram fragment that uses a synch state as a synchronization point.
Although it is abstract, I believe the diagram in Figure 11 provides an easy way to understand how synch states are used as synchronization points. To understand this, let's first be clear about the activity sequence here. The activity starts with all the action in one thread, and when the action "Main Thread Action XXX" is done, the activity breaks into three threads executing in parallel. In the first thread, "Thread 1 Action 1" is executed, then "Thread 1 Action 2" is executed. At the same time this first thread is executing, the second and third thread actions are being executed -- "Thread 2 Action 1" and "Thread 3 Action 1," respectively. By having the second synch state (the one on the right side of Figure 11), we wait until "Thread 1 Action 2," "Thread 2 Action 1," and "Thread 3 Action 1" are completed before proceeding. When all the previous actions are done, the right synch state synchronizes the previous three threads, then two new threads are forked, and both actions on these threads are executed in parallel.
Now, here is what is most significant about this example. Remember that when multiple threads are executing, the actions in one thread must not impact the actions executing in a parallel thread. In Figure 11 the action "Thread 1 Action 1" may be done quickly, and "Thread 1 Action 2" could begin processing before "Thread 2 Action 1" is complete. The only thing that will cause threads to wait for another parallel thread is a synch state, placed as shown in Figure 11.

In all the above examples the synch states are drawn as thick vertical lines; however, the UML specification does not require these lines to be oriented in one way or another. A synch state can be a horizontal thick line or even a diagonal thick line. However, UML diagrams are meant to communicate information as easily as possible; so a person should typically draw a synch state icon as a vertical or horizontal line; only when it makes complete sense (e.g., when running out of space on a whiteboard or piece of paper) should a synch state be drawn at an odd angle.

Swimlanes

In activity diagrams, it is often useful to model the activity's procedural flow of control between the objects (persons, organizations, or other responsible entities) that actually execute the action. To do this, you can add *swimlanes* to the activity diagram (swimlanes are named for their resemblance to the straight-line boundaries between two or more competitors at a swim meet).

To put swimlanes on an activity diagram, use vertical columns. For each object that executes one or more actions, assign a column its name, placed at the top of the column. Then place each action associated with an object in that object's swimlane. Figure 12 shows that two objects execute actions (e.g., Band Manager and Reporting Tool). The Band Manager object executes the "Selects the View Sales For My Band Report" action, and the Reporting Tool executes the "Retrieve Bands the Band Manager Manages" action. It is important to note that, although using swimlanes improves the clarity of an activity diagram (since all the activities are placed in the swimlanes of their respective executor objects), all the previously mentioned rules governing activity diagrams still hold true. In other words, you read the activity diagram just like you would if no swimlanes were used.
As we have seen so far, actions are typically executed by an object. But sometimes an action will output an object that is input to another action. The UML specification does not absolutely require this object output/input to be modeled on an activity diagram, but sometimes it is useful to do so, for the same reason it’s helpful to provide swimlanes in the diagram. To model this object flow, UML has a notation called action-object flow relationship, and includes two types of notation symbols -- object flow and object in state.

**Object flow**

An object flow is the same thing as a transition line, but it is shown as a dashed line instead of a solid one. An object flow line is connected to an object in state symbol, and another object flow line connects the object in state symbol to the next action. Figure 13 shows this action-object flow relationship.
Figure 13: Action-object flow relationship.

Note that in Figure 13, instead of transition lines between the two activities, we see action-object flow relationship symbols. This is because an action-object flow relationship between two actions implies transition to the other action, so transition lines are considered redundant.

Object in state

The object in state symbol is the rectangle in Figure 13. The first part of the object in state symbol is the underlined text. The underlined text in the rectangle is the object's class name -- in our example "Order" -- and this class would be found on one of the modeled system's class diagrams. The second part of the object in state is the text inside the brackets, which is the object's state name. Including the object's state is optional, but I recommend you do so if an action modifies the object's state. Figure 14 shows a complete activity diagram with multiple action-object flow relationships.
Figure 14: The "Place Order" action puts the Order object into the "Placed" state; then the "Verify Ordered Products Are In Stock" action moves the Order object into the "Accepted" state. 

The inclusion of action-object flow relationships does not change the way you read an activity diagram; it just provides additional information. In Figure 14, the "Place Order" action puts the Order object into the "Placed" state; later, the "Verify Ordered Products Are In Stock" action moves the Order object into the "Accepted" state. We know that these actions are modifying the Order's states, because the objects in state symbols have the states on them.

**Subactivity state**

The subactivity state represents another activity diagram that you can use when you want to nest another activity in the flow of an activity. A subactivity state is placed where an action state on an activity diagram would be located. You also draw a subactivity state in the same way as an action state, with the addition of an icon in the lower right corner depicting a nested activity diagram. The name of the subactivity is placed in the symbol, as shown in Figures 15 and 16.

Figure 15: Example of a subactivity state as drawn with IBM Rational XDE™
Note the slight difference between the subactivity state icons placed in the lower right corners in Figures 15 and 16. The UML 1.4 specification does not explicitly state what the icon should be, but instead says this:

A subactivity state is shown in the same way as an action state with the addition of an icon in the lower right corner depicting a nested activity diagram. The name of the subactivity is placed in the symbol. The subactivity need not be unique within the diagram. This notation is applicable to any UML construct that supports "nested" structure. The icon must suggest the type of nested structure.

For now, this means that there is not a standard symbol, since subactivity state icons can be different depending on who (or what tool) adds them to the activity diagram. So until firmer agreement is reached on the appearance of this icon, it is best to simply be consistent: Use the same icon every time you represent a subactivity state.

Conclusion

Like all UML diagrams, the number one purpose of the activity diagram is to communicate information effectively. One main reason to include activity diagrams in an overall system model is that they model the procedural flow of control for various activities. This is important, because this sort of model allows business people to get a better understanding of the business environment in which a system will run. Of course, activity diagrams are not limited to modeling business processes; they can also be used to model computer processes.

Typically, you will not use every notation element described in this article when you create your own activity diagrams. But you will make frequent use of the initial state, transition line, action state, and final state notation elements.

In our next installment of this series on essential UML diagrams, we will take a close look at the Class Diagram. See you later this fall.

Notes

1 In the current draft version of UML 2.0, the activity diagram will no longer inherit from the statechart diagram. However, a full discussion of this subject is outside the scope of this article.

2 In the United States, people younger than 21 years of age cannot purchase alcoholic beverages, such as beer or wine.

3 In activity diagrams, if you fork execution into multiple threads there is no requirement to rejoin the threads -- it's possible, though unlikely, to have an activity sequence that breaks off and then just terminates. An example of an activity sequence that might fork off and need
not be synchronized back would be a process for notifying an external system or third party of an event.

4 Remember that one action might take longer to execute than the other.

5 Although the text implies that every class in a UML model must appear in a class diagram, this is not required by the UML specification. It is completely possible that a class could appear on an activity diagram, but not on any class diagram.

6 Just as the class Order is modeled on one of the system’s class diagrams, the state should also be modeled on a statechart diagram.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
How are trends in financial services impacting the industry's software development?

An interview with Guillermo Kopp of TowerGroup and Gary Napolitano of IBM Software Group

Financial services companies continue to adjust to the business challenges of globalization, compliancy mandates, cross-industry convergence of service offerings, consolidation through mergers and acquisitions, and building and retaining customer loyalty -- all of which they have felt since the 1990s. These well-established market drivers, coupled with severely curtailed earnings in the current economic downturn, have radically altered income structures on which the industry has long depended. The loss of income formerly generated by investments in the equity markets has left banks, investment firms, and insurance companies with an imperative to cut operating expenses dramatically.

As a consequence, IT priorities in these institutions have shifted from long-term, discretionary projects aimed at improving customer service, to more strategic efforts geared toward lowering cost of ownership. IT must also cope with a new regulatory emphasis on mitigating operational risk, in response to global political realities. Hampered by business processes that have evolved around vertical integration and isolated silos of data -- and by the need to merge disparate systems across organizations -- these companies are moving rapidly to integrate their application portfolios to reduce cost, manage risk, and present a unified face to customers.

To better understand how these pressures in the financial services sector are affecting software development, we asked Rational Edge reporter Scott Cronenweth to interview two industry experts: Guillermo Kopp, director of financial services strategies and IT investments, and emerging...
technology solutions at TowerGroup; and **Gary Napolitano**, director, finance industry sales, for IBM's Software Group.

**Scott Cronenweth:** In this era of shrinking budgets and intensified pressure to innovate, what trends now loom largest among IT departments in the financial services industry?

**Guillermo Kopp:** The key trend I see is a shift from sledgehammer cost-cutting to a more strategic approach that hones operational efficiency. Financial institutions have begun to focus on business process improvement, for which IT is obviously a prime enabler. This means that IT capabilities -- people, applications and infrastructure -- must align closely with the operational strategies of the business. This alignment should drive fundamental spending decisions, such as which systems to maintain or enhance, and which to re-engineer or replace.

**Gary Napolitano:** IT spending is very different than it was a few years ago. The industry dynamics at work today are forcing companies to defer large, discretionary projects that in former times were commonplace. Then, everybody was talking about achieving ultimate levels of customer loyalty and customer relationship management. That's clearly not happening today. Instead, financial services institutions are taking on smaller and much wiser projects that move them a step closer to their strategic goals.

**SC:** Yes, but legacy transformation projects also cost money. We know these companies are strapped for cash, so where are these dollars coming from?

**GK:** A lot of companies are redirecting routine maintenance expenditures to more strategic re-engineering and replacement efforts. Based on our research, legacy replacement today constitutes about 11.8 percent of total IT expenditures globally. By 2009, it will be running at roughly 14.7 percent. This steady increase in spending will come not so much from incremental funding, as from rerouting maintenance budgets toward new development.

What we also see happening in the industry is a growing recognition that legacy extension and replacement can drive operational efficiency. TowerGroup research shows that every dollar you spend on modernizing your IT infrastructure can increase opportunities for improved operational efficiency up to seven-fold. Why? Simply because outmoded systems constrain business operations. Remove those constraints, and companies can realize benefits not only in their back offices, but also in the front office; the payoffs are truly end-to-end. This means transforming some pieces of the existing infrastructure -- as opposed to merely "re-wrapping" them for connectivity purposes -- and those that fail to add value to current business strategies may be candidates for outright replacement.

**GN:** Yes, and again, we should emphasize that in this industry, the spending right now is mostly about integration, not about building all-new
applications. Financial services companies have huge, existing operational infrastructures, and they still spend orders of magnitude more money on maintenance than on development. Most of our customers want to add onto their existing base, not rip and replace.

But, as Guillermo said, upgrading the business application portfolio is more important than ever. In the insurance world, for example, many companies have been losing money on core business operations such as underwriting, because their business processes are badly antiquated. In the past that almost didn't matter, because there was so much money to be made in the equity markets. You could write a premium or policy, invest the money you made from it, and get outstanding returns; you never had to worry about overall profitability. Today, a glaring hole in one of your business processes will be much more evident -- and destructive -- whatever industry you're in. That vulnerability is driving a change in IT priorities, not just in insurance, but across all financial services markets.

Plus, as you work toward greater efficiency, you also have to factor in new requirements around operational resilience. How do you prepare your IT infrastructure -- as well as your business -- for unplanned events like a terrorist attack? Now there are major regulatory activities across the financial services industry designed to mitigate operational risk, such as the global BASEL II initiative for banking, and the Patriot Act here in the US to prevent money laundering.

---

**SC:** How do these efforts to address operational risk affect IT departments?

**GN:** Well, BASEL II is setting an important precedent: As it unfolds over the next couple of years, it will represent the first time that the technology infrastructures of private businesses are scrutinized for operational risk. Soon regulatory bodies may be scrutinizing the operational risk profile of every major organization, whatever the industry. Under the terms currently proposed for BASEL II, if a bank's risk is higher than the average for the industry or exceeds a predefined standard, the institution is forced to hold more capital in reserve. Then they can't invest that capital, and this has a direct negative impact on their bottom line. So right now, in banks around the globe, the IT people are charged with hardening the technology infrastructure while at the same time streamlining for efficiency.

---

**SC:** Isn't there a natural synergy between these two objectives? When you streamline business processes and eliminate unplanned redundancies, don't you inherently reduce risk and improve resiliency?

**GN:** Absolutely. Mitigating operational risk will drive the streamlining of business processes, and that will ultimately reduce operational costs and improve efficiency. BASEL II, and perhaps other initiatives like it, will just
compel companies to focus on this and do it faster.

**GK:** I agree that these regulatory mandates present an opportunity for financial services institutions. Those that pursue risk management pragmatically and proactively will reduce operational costs and improve their operational risk profile at the same time. How does that look from the IT side? I think it will force a lot of data integration, among other activities. Concerns like information security will also grow in importance. But to address both costs and risks effectively, you need to look beyond tactical disaster recovery, beyond data backup or hardware redundancy.

A firm's goal should be *business continuity* -- creating more resilience in its day-to-day operations. How do you handle everyday disruptions? Or unexpectedly high transaction volumes? With a growing emphasis on real-time activities and a growing interdependence among firms due to factors like outsourcing and convergence of services, you need alternative pathways for processes to flow down if something goes wrong. If a hardware component fails, or a piece of the network cuts out, there should be an alternative path for data so transactions will keep moving. Really, this goes beyond IT to the overall management of the business; it requires not just solid technology support, but also carefully planned business processes. It requires looking at the right metrics and putting the right tools in place to manage both the core business infrastructure and the application portfolio.

**GN:** Most people equate operational resiliency with infrastructure robustness. But it also depends on your applications and artifacts -- and your development process. If there were a disaster, how would you recover your key development assets? Would your IT team have the skill and the code artifacts to retrace and rebuild a mission-critical system if they had to? You need to put excellent process controls in place -- including change control and management processes -- to do that effectively.

**SC:** Greater efficiency and resiliency don't necessarily net you more business or address customer demand. While financial services companies whip their technology infrastructure into shape and streamline their business processes in response to these regulatory initiatives, what are they doing to generate revenues?

**GN:** Even though companies have significantly cut back their original, large-scale CRM projects, the mantras of customer care and customer profitability are still resonating across all financial services industries, because customer demands are increasing all the time. Institutions are just finding less expensive ways to make improvements. Take the banking world: Customers want the ability to bank anytime, anywhere, and through any medium. And they want to do this almost for free; they bristle at the idea of additional costs. Plus, they're always shopping around; they want more, and they'll shift their loyalty, temporarily, to whatever institution can supply it. This isn't surprising: When their friendly
local bank gets swallowed up by a regional or national giant, customers are often wary of the new institution.

As for the giants, they're trying to strengthen their branding and hang on to those customers they acquire. Many firms are moving to redesign and integrate their systems, from the glass to the back office, and everywhere in between. For the customer, this delivers a common look-and-feel for every interaction they have with that bank. It also creates a common, integrated view of the customer that all bank employees can access, so they can serve customers more efficiently and improve the customer experience. Initially, institutions focused integration efforts on service delivery channels -- ATMs and the Internet. Now, the emphasis is shifting to integrating applications and data. If you're a big regional bank, and you've just acquired a small local savings and loan, you want to be able to pull up and process all the data for any of that S&L's customers at any branch in your region as soon as possible -- and you want the screen to look exactly like all the other screens for your related applications. If they can access a truly integrated view of your customers, your employees can deliver better customer care, which leads to greater profitability.

**GK:** And that integrated view becomes a business differentiator. As all banks typically offer essentially the same products and services, those that take a more holistic view of the customer and provide a consistent customer experience across channels have a great competitive advantage. They can get deeper into the relationship with the customer -- and deeper into the customer's wallet. Leading institutions are getting very good at this. What they're aiming for is the kind of well-established, trusting relationships you might see at local community banks. And by creating an application set that provides a holistic view of the customer, IT can support this "personal touch."

Whether we're talking about a local bank or a big investment firm, an integrated view of the customer is going to be critical to sustained growth. This customer-centric emphasis will drive purposeful IT projects that are well-aligned with business strategy, as I said earlier. No one in a financial institution will be saying, "Oh, yes, IT is busy doing projects, but nothing customers will be able to see." Instead, you'll see increased IT spending on things like integration for all customer-facing channels and cost center automation; and, of course, electronic delivery systems like ATMs and the Web will also still play a role.

**SC:** In the development world, some financial services organizations have a reputation for a "not invented here" attitude. Is the trend toward integration also moving companies toward standards-based, packaged applications?

**GN:** It depends on the organization. You can't really say that's happening across the board. The big companies are still heavily into "growing their own," and it's not just because of attitude: They really do have very unique requirements. I will say, though, that there is a higher level of "buy" versus "build" right now than I've ever seen, especially in the US. In
Europe this is less the case, but the point is, companies are willing to consider the possibility of buying now. Their attitude is definitely changing.

**GK:** You also see a shift away from doing everything in-house when you look at how business processes are managed and controlled today. For example, many financial services companies are now using Web Services, not only to pool their internal efficiencies and avoid duplication of function, but also to reach beyond the firewall and connect with other institutions that can fulfill some of those functions as a best practice.

The distinction between the way institutions handle origination of mortgage loans and the way they service them is a classic example. Some companies that have a great market presence or great local focus do a great job at origination. But they may not have the economies of scale or the core competencies to be the best-in-class at servicing the loan. So they decompose that function and connect to a third party at the servicing end, via Web Services. Alliances and partnerships of this sort are taking hold very strongly. We see them not just in retail banking but also in the insurance industry and in securities and investments -- wherever there are specialized functions and competencies. Even wholesale banking is going this way.

**GN:** There's a lot of talk in the industry about the concept of infrastructure resources as demand-driven "utilities." Whether the source is internal or external, financial services companies are looking at sharing resources -- both within their own firm and between firms -- to drive efficiency. But some people misinterpret the "utility" model; they assume it translates to cost reductions and requires outsourcing. And that may not be true. Take an internal utility concept like dynamically allocated server capacity implemented within a firm. If you configure your resources differently, you may find all the computing power you need right within your own enterprise. That means you don't have to spend money to buy more, but you won't actually save money -- because the resources you already have won't cost you any less.

**GK:** Rationalizing legacy systems is also part of this utility concept. Large institutions are asking, "Why have literally tens of financial reporting applications or other versions of the same functionality? Couldn't we create an internal utility to provide that function across the enterprise"? And many of them are seeing the value in launching those projects now.

In addition, more are realizing that there's no shame in using some external components to drive value creation at the application level. It's a matter of core competencies. Increasingly, IT groups are trying not to let affection for their proprietary internal systems deflect them from focusing on what's best for customers.

**SC:** Does this receptivity toward outside resources extend to development activities as well? There has been a lot of press lately on how US companies are outsourcing application development overseas. Is the financial services industry moving in this
GN: Offshore development is extremely exciting, extremely hot, and extremely hard to do, all across the industry. Almost every major firm is at least looking at it, even if they're not doing it yet. Right now there are a lot of places where it's working, but there are also a lot where it isn't. Many firms don't have that much experience with outsourcing development. They want to move it offshore because they figure they'll pay less for programmers in China or India, or wherever. But they forget how hard it can be to establish well-coordinated management and communication processes, from the business side of the house right on through to the software side. When you transfer development from onshore to offshore, you are inherently increasing your risk. So you need an absolutely clear path that lets you begin mitigating risks early on. You need a common visual language that can close cultural gaps and ensure unambiguous communications, plus clearly expressed requirements and airtight change management controls. You also need common artifacts, and the right metrics to measure progress and stay within your timeline, your cost structure, your thresholds for defects, and so on. All these things are critical for successful development under any circumstances, but when it's distributed across cultures and time zones, tight process control is a life-or-death proposition.

GK: Actually, the numbers for IT outsourcing are already huge in this industry. At financial services companies, outsourcing in all its forms accounts for about one-third of total IT spending globally. This means that institutions already are handing over a hefty percentage of their IT budgets to third parties, whether for development, ASPs, service bureaus, or the like (not including hardware).

And I don't see this as simply a move to get cheap labor. What organizations are trying to buy when they outsource is better productivity. Regarding development, this translates to better dependability in hitting milestones and meeting the organization's real business needs. Why not favor those groups, whether internal or external, that can do it right? However, some projects you just can't outsource -- it's a matter of control and confidentiality. For certain applications that are mission critical or handle sensitive customer information, institutions will continue to keep all development activities internal.

SC: Let's drill down further into the development process issues Gary mentioned. Even with a strong process in place, is it realistic to think you can manage complexity across large-scale, multi-organizational projects while emphasizing cost efficiency?

GK: I don't think it's just a dream if you select projects and plan for them the right way. To reiterate my previous point about aligning IT and business strategies, you should evaluate every potential development project against corporate goals. If your project passes that test, then you establish appropriate cost/benefit metrics, do an impact assessment, and identify priorities, interdependencies, risks, and so on. Then you keep this...
focus as you drive the definition of requirements, and on into the project lifecycle. The key is to make IT a strategic partner in meeting business goals, rather than a passive recipient of business directives. And if you're outsourcing part of your development, you have to infuse that message into the service provider's organization -- through training, mentoring, monitoring, and excellent communication.

**GN:** Of course, the more organizations that are involved, the tougher it is to keep them all on the same track. In the past, the big issue was centralization versus decentralization. But this new world of development requires a *federation* of IT resources. That means you need the ability to seamlessly link all your different silos of data and business rules via reusable processes that you can monitor and manage. A federated approach reduces not only cost but also complexity, especially when you're integrating across applications.

I believe you can make that approach successful with a repeatable, standardized development process: a standardized methodology for establishing requirements; a standardized framework for business modeling; standardized code generation and application delivery environments; standardized test cycles; and so on, until you have complete lifecycle management. You can't streamline processes you can't repeat. The more repeatable and efficient your development process, the more likely you'll be able to successfully share major system components, skill sets, and data across projects. Standardization across tools and process really adds value and helps control costs for the company.

**GK:** Another reason financial services companies have embraced offshore development is that many of the service providers have very mature development capabilities. They've embraced the Capability Maturity Model® and have excellent project management and quality assurance capabilities. I'm a strong advocate of Six Sigma™ improvement efforts and other professional project management disciplines. I think others in the industry recognize that you can't manage IT development by accident, so they look for service providers with top notch process controls.

**SC:** Looking forward, do you think companies who invest in all the things we've said they need -- to streamline their business processes, integrate their application portfolio, and tighten up their software development methodology -- will achieve and sustain competitive advantage? The investment required to do these things is substantial; will the ROI be great enough to offset that?

**GN:** That's right; we have been talking about substantial investments, and you can't do it all at once. If you're an IT manager in a financial services company, your first priority is the ongoing optimization of business processes while you refine your understanding of what customers need and want. Next, you work on optimizing your application portfolio across the organization through integration, replacement, and consolidation. Meanwhile, if you don't already have one, you should be instituting a
process framework and also addressing issues around operational risk, infrastructure resiliency, data security -- the whole works. Any new business process applications you build should be based on open standards and a common, flexible infrastructure that you can dynamically manage.

But if the economy continues to lag, integration will remain the key activity for IT, and process management can support that. It's a matter of how fast it can happen. Remember, most of these companies have been living in a silo-driven world, where Function A or Region B did its thing, and if they passed the file over to Function C you were lucky. The winners will leverage competitive advantage by knowing their business goals and being able to integrate applications as seamlessly and quickly as possible into the existing portfolio.

The question is, in lean times, will company managers have the discipline to think clearly about what IT needs in order to support the business, and continue funding IT improvements? A "pay-by-the-drink" systems model can really save companies money, but few are set up for it right now. They need an application portfolio that puts all the necessary flexibility, monitoring, metering, and virtualization capability in place. They can certainly get there with a development process that enables them to deliver and manage all this integrated capability, but it will take time and patience.

**GK:** Alliances and partnerships can also provide some of the flexibility that Gary is talking about. So can Web Services, XML, and other means to decompose, wrap, and integrate your strategic application modules using standard interfaces. Ideally, this can take place not just at the application level but also at the level of business semantics. What type of transaction do you use? What is the meaning of the data that's being modified? What are the integrity rules?

These changes, too, require a disciplined development process. They will happen because ultimately they will reduce total operating costs. But again, the effort required to produce this kind of transformation is great; it won't happen overnight. And IT can't drive the process without support from the business side.

Many firms today are trying to "save money" by decomposing function and creating alliances, but they won't actually achieve cost savings unless they also pull internal efficiencies that make outsourcing arrangements, in effect, self-funding. Internal savings start to happen the moment you begin creating an IT infrastructure that is better aligned with business operations.

---

**Notes**

1 A proposed replacement for the current 1988 Capital Accord, designed to allow banks and regulators to better evaluate the risks that banks face today, and to encourage sound banking practices. For more information
visit http://www.bis.org/publ/bcbsca.htm.

2 A model for judging the maturity of an organization's software processes and for identifying key practices required to increase the maturity level. For more information visit http://www.sei.cmu.edu/cmm/cmm.html.

3 A methodology for continuous improvement in business processes. For more information visit http://www.isssp.com/.

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

How IBM Rational Rapid Developer supports global e-business

In order to meet the growing demand for globalized transactional Web applications, software development organizations need to explore new, highly productive environments for creating and globalizing Web products. The solution is IBM Rational Rapid Developer®, which offers an "architected rapid application development" (ARAD) approach for modeling and delivering optimized, scalable applications for multinational businesses.

"IBM Rational Rapid Developer introduces a revolutionary approach to the design and implementation of Web-based, n-tier applications. And now, the architected rapid application development (ARAD) paradigm used by Rational Rapid Developer can be leveraged for the development of globalized Web applications. No other solution provides the speed of application delivery, ease of maintenance, or abstraction from the typical complexities of building n-tier, globalized solutions that Rational Rapid Developer does."

-Claudia Dent
VP, Product Management
IBM Rational software

Click here to read the full paper!

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
Globalization and Creating Multi-Locale Applications

Raul Ortega
ortegar@us.ibm.com
Table of Contents

The Business Opportunity: Globalizing Web-based Systems........................................................................................................3

The Technical Problem: Technical Challenges to Globalization..................................................................................................4

The Solution: IBM® Rational® Rapid Developer..........................................................................................................................5

Rational Rapid Developer Development Process.........................................................................................................................5

1. Application Functional Requirements ............................................................................................................................................5
2. Import Existing Models and Databases.......................................................................................................................................5
3. Rational Rapid Developer Modeling System...............................................................................................................................6
4. The Agile Application Model..........................................................................................................................................................7
5. Technical Requirements...............................................................................................................................................................7
6. Rational Rapid Developer Construction System .......................................................................................................................7
7. Deployed Code..................................................................................................................................................................................7
8. Summary..........................................................................................................................................................................................8

Globalizing a Web Application..........................................................................................................................................................8

Before Globalizing an Application – Sizing up the Problem..........................................................................................................8

What Languages Will Your Application Need to Support?.............................................................................................................9
Will You Need to Consider Special Languages?..........................................................................................................................10
Are You Migrating an Existing Application or Are You Creating a New Global-Ready Application?......................................................10
What Libraries Should Your Application Use?................................................................................................................................11
What Do You Need to Know About Your Host Development Environment?........................................................................................11

Application Design Strategies..........................................................................................................................................................11

Page Development Strategies...........................................................................................................................................................11
Response Locale Strategies.................................................................................................................................................................12
Browser Encoding Strategies...............................................................................................................................................................13
Database Strategies................................................................................................................................................................................14
Currency Transaction Strategies..........................................................................................................................................................16
Time Zone Strategies.............................................................................................................................................................................17

Internationalization...........................................................................................................................................................................18

Externalizing Application Strings........................................................................................................................................................18
Encoding Issues.....................................................................................................................................................................................21
User Interface Issues.............................................................................................................................................................................23
Formatting Issues..................................................................................................................................................................................24
Managing Images..................................................................................................................................................................................26
Managing Error Messages.................................................................................................................................................................26
Other Internationalization Features..................................................................................................................................................27

Localization.......................................................................................................................................................................................27

Design Time vs. Runtime Testing.......................................................................................................................................................28
Translation..........................................................................................................................................................................................28
Translation Granularity.........................................................................................................................................................................29

Construction.....................................................................................................................................................................................29

Conclusions.....................................................................................................................................................................................30
Introduction

In order to meet the growing demand for globalized transactional Web applications, we need to explore new, highly productive environments for creating and globalizing Web products. The solution is Rational Rapid Developer®, that offers a model-driven, architected rapid application development (ARAD) approach to delivering optimized, n-tier, scalable, global applications.

The Business Opportunity: Globalizing Web-based Systems

IBM’s vision for a new era in business computing is defined by highly adaptive companies that can instantly sense and respond to changing conditions and make business operations easily accessible to others. In IBM, this vision is called e-Business On Demand.

The conditions and operating environments for these companies have become increasingly global in nature. The key to survival for many of these companies lies in their ability to operate successfully in global markets. In order to exploit global markets, these companies must take into account the diversity of languages and cultures throughout the world.

Global products result in the efficient and effective deployment of applications worldwide with their ability to reach a wide range of users regardless of language or cultural preferences, local business practices or conventions. Integrating multilingual information and global business processes, end-to-end, across the company, and across borders with key partners, suppliers and customers is becoming an imperative, as is responding with speed in any language to any customer demand, market opportunity or external threat.
Globalization of Web applications has been slowed or deterred to some extent by the complexity of creating new global applications or transforming existing applications to meet the needs of global users.

Consider the following:

- By 2005, 70% of Web users will speak a primary language other than English. (GlobalReach 2003)
- According to IDC, 2001, companies that do nothing to globalize their Web sites project an average of 12% of their revenue will come from foreign sources in 2002 while companies that do globalize project that 30% of their revenue will come from foreign sources.
- Globalization markets are poised for growth once we are over the economic slump. As more global organizations compete for international audiences, customer expectations will be raised, looking for providers who “speak their language.” (ABI Report, Oct 2002)

The time to embrace the global e-Business On Demand vision is now.

**The Technical Problem: Technical Challenges to Globalization**

While the opportunity may be obvious, globalization of Web applications has been slowed or deterred to some extent by the complexity of creating new global applications or transforming existing applications to meet the needs of global users.

The following challenges have slowed the globalization process within many companies:

- Information about globalization is hard to get and can be confusing
- Training the entire team in the complexities of globalization is difficult and costly
- Implementing globalization is costly and time-consuming
- Translating a Web application can be cumbersome, costly, and inefficient
- Determining the effect of translation on page design and layout is awkward and time-consuming, because it can only be done at run-time in the current development paradigm
- Creating and managing text strings across languages is complex
- When translating large Web sites into multiple languages, managing the workflow can be overwhelming
The Solution: IBM® Rational® Rapid Developer

IBM Rational software introduces Rational Rapid Developer—the solution to creating Web applications and providing localized versions in a highly productive environment.

By combining powerful visual modeling, RAD techniques and automated construction, Rational Rapid Developer enables a broad class of developers to rapidly build complex e-business applications that are portable to all leading deployment technologies. Rational Rapid Developer-based applications are reliable, scalable, secure and readily modifiable to accommodate changes in technology and business objectives. Leveraging mainstream development skills, you can deploy and maintain enterprise-class applications that integrate with an extensive range of legacy systems. Time-to-market and cost of ownership are greatly reduced.

After a brief introduction to the Rational Rapid Developer development process and paradigm, this white paper focuses on the role of IBM's Rational Rapid Developer in facilitating the globalization of Web applications.

Rational Rapid Developer Development Process

The Rational Rapid Developer development process consists of two main steps:

1. Visually modeling your application using the Rational Rapid Developer Modeling System
2. Automatically constructing the code using the Rational Rapid Developer Construction System.

1. Application Functional Requirements
   Functional requirements are the starting point of your application. They are typically captured in the use case models and other document repositories.

2. Import Existing Models and Databases
   Although Rational Rapid Developer excels in the development of "greenfield" applications, in which there are no existing assets to incorporate into the new system, the reality of today's enterprise application development is that new applications typically must integrate existing databases, UML models and legacy applications and components of various types.
A frequent requirement in enterprise application development is to harness business functionality that has been modeled in one of the popular UML modeling tools such as IBM Rational Rose. Rational Rapid Developer enables development shops that have adopted UML modeling as a standard to accelerate the implementation phase of the development life cycle by importing UML class diagrams into Rational Rapid Developer and rapidly moving to executable applications. This is accomplished using the open, industry-standard XML Metadata Interchange (XMI) interface, along with a native synchronization feature with Rational Rose and IBM Rational XDE.

Rational Rapid Developer can also save tremendous amounts of time by allowing you to import a database schema from one of the leading relational databases into a class model. This is also known as database reverse engineering. Rational Rapid Developer can also import IMS, VSAM, and CICS files from a legacy environment.

Additionally, any existing Rational Rapid Developer application can be imported into a new application to start the development process. This feature allows you to create your own libraries of "components" that can be reused across the organization. The benefit of these types of components is that they are not tied to any single deployment technology or platform. Instead, they are component models that can be generated at build time into actual code in the desired deployment technology.

3. Rational Rapid Developer Modeling System
Application development begins with functional requirements. The Rational Rapid Developer Modeling System is used to create a detailed model of your application that meets these requirements. Modeling starts at the inception phase of the application when requirements are being gathered. It carries over into the elaboration (design) phase when a high-level model of the application is created. Finally, during the implementation phase, detailed parts of the application are modeled. The modeling involves all aspects of the application: business objects, user interface, messaging and integration. The modeling process does include some hand coding, but this coding is limited to the essential business logic needed to implement business rules and processes, and is typically less than 5% of the total application code.
4. Agile Application Model
The result of the modeling step in Rational Rapid Developer is an agile application model. This model reflects the entire functional needs of your application. At this point, it is a virtual application, about to be generated by the Rational Rapid Developer Construction System.

5. Technical Requirements
It is important to note that the technical requirements of the application are independent of the functional requirements. Technical requirements identify the specific n-tier deployment technologies that are used to deploy your application. Technical requirements drive the design of the deployment model and include the capacity, performance, scalability and availability needs of the application. Technical requirements are specified before you construct the application.

6. Rational Rapid Developer Construction System
The Rational Rapid Developer Construction System constructs and deploys your application model into high-quality, industry-standard n-tier code within minutes. The complexity of n-tier applications is removed from application development and is encapsulated as an engineering discipline in the construction system. Construction is optimized to your specific deployment technologies.

You invoke the Rational Rapid Developer Construction System in all phases of application development, for incremental construction during implementation, and for full construction during system testing and deployment.

7. Deployed Code
The code constructed by Rational Rapid Developer is a rendering of your modeled application, optimized for your selected deployment platform. The code is a constructed artifact of the application but not the actual application. Therefore, modification and maintenance are performed within Rational Rapid Developer and not directly on the constructed code. You can, however, override constructed code components for the various tiers to achieve specific results.

For new versions of your applications, you change your application model and re-construct it to generate new deployment code. After each iteration, the previously deployed code is discarded. As your business needs evolve, you will
IBM® Rational® Rapid Developer Globalization and Creating Multi-Locale Applications

8

Rational Rapid Developer provides an environment in which you can model and construct the typical set of activities that are associated with globalizing a Web application.

have a new set of functional requirements to reflect these needs. You use the Rational Rapid Developer Modeling System and modify the application model to meet the new functional requirements. After making all the requisite changes, you re-construct the code, test and debug, and re-deploy the application.

8. Summary

New technologies are rapidly emerging and new versions of existing technologies will continue to be released. To keep pace, businesses need to quickly adapt their existing applications to new technologies and business requirements without losing their investments in current systems. Rational Rapid Developer allows you to design and maintain applications as an application model, rather than at the technology level. Your application model can be used to rapidly redesign, reconstruct and redeploy applications for the latest technologies.

Globalizing a Web Application

Now that you know a little about developing applications with Rational Rapid Developer, let’s discuss the globalizing capabilities that have been added to Rational Rapid Developer with the Globalization Model.

There are many issues to contend with when globalizing an existing Web application or when creating a new one. The following sections:

1. Illustrate the degree to which globalization affects an application.
2. Highlight, whenever possible, the role of Rational Rapid Developer in facilitating globalization activities.

Rational Rapid Developer provides an environment in which you can model and construct the typical set of activities that are associated with globalizing a Web application. You can do this with minimal knowledge of the underlying complexities associated with the implementation platform.

Before Globalizing an Application – Sizing up the Problem

To set the stage for globalization, you must first define the problem. The nature of the problem affects and influences the set of activities and the application design that is required to accomplish your globalization goals. This section discusses general, high-level topics that are just as important as the actual implementation activities.
What Languages Will Your Application Need to Support?

The Internet provides worldwide accessibility for your application. You need to consider the languages that must be supported for your application to succeed.

In an enterprise intranet scenario, your company operates in specific countries, so it's easy to determine the languages that your application must support. However, with products such as eBay or Amazon, the reach is more unpredictable and widespread and your application needs to support many languages.

Rational Rapid Developer's Globalization Model provides you with a convenient mechanism to define any number of locale descriptors. Locale descriptors are used to identify cultural characteristics such as language, formatting, sorting, images, and so on. Once defined, they are available throughout your application.

In the following illustration, the Globalization Model shows that the application supports five languages.
Will You Need to Consider Special Languages?

Some languages pose greater challenges than others. The right-to-left characteristic of Hebrew and Arabic requires special attention to layout and presentation fonts.

In Rational Rapid Developer, the right-to-left attribute can be easily specified in the model without the need to get into the details and complexities of the code. The attribute can be associated with any HTML tag on the page. For example, in the following example, the “dir = rtl” attribute has been applied to the page BODY tag. Rational Rapid Developer will integrate this tag with the HTML produced for the page.

Are You Migrating an Existing Application or Are You Creating a New Global-Ready Application?

The complexity of migrating an existing application depends on the degree of thought and effort devoted to globalization when the application was first created. Major effort is required to migrate already existing applications that have not been designed with globalization in mind. From externalizing strings, to providing appropriate calendars, to the creation of formatters and much more, a large amount of code must change.

With Rational Rapid Developer applications, changes are made to the application model and the application is generated, as required. The process of migrating applications in which no thought was given to globalization is much easier in applications developed in Rational Rapid Developer, compared to manually coded applications. Rational Rapid Developer generates Unicode-enabled code, externalizes strings, and generates property files (you won’t have to create and manage property files manually). Rational Rapid Developer also generates calendars, formatters, and the requisite code to handle time...
zones. Virtually any feature that is required for globalization can be modeled in Rational Rapid Developer and the appropriate code generated.

**Localization Libraries**
Although Java includes built-in internationalization services, your development team will most likely require the use of a library such as IBM's ICU (International Components for Unicode). The library provides a full range of services for supporting internationalization. It provides cross-platform C, C++ and Java APIs. The Java version of ICU builds upon the internationalization features in Sun's Java.

With Rational Rapid Developer, you can leverage powerful libraries such as ICU in custom code that is generated for an application.

Rational Rapid Developer also provides a set of internationalization methods that are at a higher level of abstraction than libraries such as ICU. Using the Rational Rapid Developer internationalization methods reduces complexity while providing platform independence. These methods, and many more, are provided in the Rational Rapid Developer Framework APIs.

**What Do You Need to Know About Your Host Development Environment?**
The globalization features of the host operating system for your development environment are important to the development of the application. Features such as the ability to set up input locales and the ability to set up browser language preferences will prove to be valuable.

**Application Design Strategies**
This set of application design strategies will help you plan the implementation of your Web application.

**Page Development Strategies**
It is often the case that a Web application is created for one culture or locale and then ported to a set of other required cultures. The application is duplicated and then localization changes are made to the copy of the application. While initially appearing easier, consider the problems that are introduced with this methodology. Propagation changes across the set of localized applications will prove to be costly, time consuming and error prone.
Rational Rapid Developer provides an environment that enables you to easily create a single page layout that serves up or produces pages in any language that your application supports. Notice in the following screen capture of the Rational Rapid Developer Page Architect that you can cycle through each language supported by your application. This provides you with a view of the page in each language, at design time. You can easily design the application so that the language in which the page is displayed is selected dynamically at runtime.

This is a unique feature that differentiates Rational Rapid Developer from any other development environment. However, it should be noted that you will not lose the flexibility of creating a page for a specific language, if that is what is required.

Response Locale Strategies
With the ability to define a single, consistent layout for a set of languages, how will you produce the page with the correct language for a given user?

Rational Rapid Developer provides the ultimate flexibility in defining the response locale for a page. This setting establishes the language for page creation, and thus the language provided to the browser. You can either set the page to a fixed locale or you can define a callback method to set the response locale. While the fixed option provides a static language page, the callback method enables you to define whatever strategy is needed to set up the response locale, and thus the language.
As an example of one strategy, suppose that your application supports English, Spanish, and Japanese. If the request comes in from the UK, you will respond with an English page. If the request comes from India, you may also want to respond with the English page. You can define a lookup table in the callback method that dynamically determines the response locale.

You could also establish the response locale based on the user preferences that have been stored in a session variable at user login. Yet another strategy could allow the user to change the application language or cultural settings at will.

**Browser Encoding Strategies**

As demonstrated above, Rational Rapid Developer enables you to establish the language for the page that will be delivered to the browser. This solves only part of the problem. When the page is delivered to the browser, the browser must present the page with the encoding that matches the language delivered. This enables the browser to correctly interpret and present the data. For example, if you try to present Hebrew strings with a Western European encoding, the browser will not be able to interpret the Hebrew strings correctly and the page will not display Hebrew.
As information (e.g., strings, numbers, dates, times) is presented or entered using a browser, the interpretation of the data is based on the encoding that has been set in the browser. You can usually use two types of encodings:

- **Code pages, which handle a limited set of characters (such as Western European)**
- **Unicode, which handles all characters**

Like the response locale, you can design a strategy that sets the correct encoding in the page response. Setting the encoding in the page response in turn sets it in the browser encoding before the page is interpreted and presented.

As shown in the preceding illustration, Rational Rapid Developer provides several ways to set the encoding. The UTF-8 setting represents a Unicode encoding that is a safe way to represent most languages. You can also set the encoding using a callback method (for full flexibility), or by the encoding that has been associated with the page locale.

**Database Strategies**

Database design can be grouped into the following categories:

- **Database structure**
- **Data encoding**
- **Data conversion**

**Database Structure**

Database structure or schema determines how you organize multi-lingual data within an individual database. Will you store data for a specific language in a table devoted to that language, or will you have a column in a table that identifies the language? You may decide to store data from different languages in different databases that mirror each other in structure. The implications of the database structure affect the code that stores, retrieves, and processes data. Maintenance is also affected.
While this is mostly an application database design issue, Rational Rapid Developer provides for properties in the model to specify table attributes that should contain multi-lingual data. By selecting the Unicode checkbox in the text attribute specification, an attribute is created in the database with the Unicode data type specification. For example, the Unicode version of the varchar data type is the nvarchar data type.

![Figure 5. Unicode Data Type Specification.](image)

Data Encoding

Like the encoding that was specified for the browser, the encoding for the database establishes the languages that are correctly stored in the database. Different database vendors handle encoding differently. Rational Rapid Developer understands the encoding capabilities of the supported databases and allows you to make the appropriate encoding selections from the model. The following example of the IBM DB2 database property sheet shows the DB2 properties that are available to define the encoding.
If you are migrating from a single language application to an application that supports multiple languages, you need to consider how to migrate the already existing data without compromising data integrity.

**Currency Transaction Strategies**

Currency transactions add complexity. The requirements may affect the database structure because of the need to store additional information (e.g., the date and time when a transaction was recorded, the base currency and potentially other attributes of the transaction). Local laws that regulate financial transactions need to be incorporated into any financial processing.
These are mostly application design issues. Rational Rapid Developer facilitates the incorporation of a currency design by allowing you to focus on the application and not the underlying complex technology.

The ability to easily accept Web services into your application also allows you to leverage currency conversion services.

**Time Zone Strategies**

Presenting and storing date and time data can be difficult in the Web application domain. The time that you display or enter on a Web page depends on the problem that you are trying to solve in the application domain.

What time will you display? Do you display the local time? For example, when you place an order in New York at 9AM Eastern Standard Time, what time will you display for the order date / time on a page displaying that order in California or Hong Kong?

How do you store the data when users in different time zones enter local date / time data? Will you store the local time qualified by some time zone code?

Once you have determined the application design for time zone requirements, you simply provide the Rational Rapid Developer-based application with the time zone for data storage and the default application time zone used to present or enter date / time data. The system does the rest.
You can have a finer granularity of control for time zones by setting a specific time zone for a specific field displayed on a page. This is shown in the following example.

![Figure 8. Time Zone, Field Properties.](image)

**Internationalization**

Once you have made your strategy decisions, the next step is to complete the implementation. For new applications, this means following a set of rules that dictate how you implement aspects of your application. For existing applications, this involves making modifications to your application that enable you to provide local support, or to localize the application without having to modify code.

In the Rational Rapid Developer paradigm, the part of the definition about not changing code takes on a different meaning since Rational Rapid Developer will generate as much as 95% of the total application code.

**Externalizing Application Strings**

When asked what it means to “globalize” an application, most developers will utter the words “externalize your strings” as the first and sometimes only order of business. While there is much more to globalization, externalizing strings is an important part of getting it right.
Of course, if you took the time and effort to externalize your application strings when creating the first version of the application, you would be in a better position to localize. However, it is often the case that globalization is a lower priority than “getting the thing done” and it is overlooked.

For hand-written applications that have not been designed for globalization, this means going through the code and replacing references to strings with functions that fetch the strings from some external string storage mechanism. This can be done manually or through search and replace utilities. This means that you must “touch” all of your code – a process that can introduce errors into your application.

With Rational Rapid Developer, the process of externalizing strings falls into two categories:

1. **Model the strings that need to be translated.**

   Controls such as labels need to present the correct translated string based on the language requested for a page. A collection of translated strings (strings that represent the same meaning in different languages) are assigned a key. The key is then assigned to the control. At runtime, when the page is presented in a given language, the label makes a request for the translated string that matches the requested language.

   You can create keys by first designing a page in a language of choice and specifying the controls that require keys. In the following example, the context menu is on a label and the menu selected creates a key for this control. Notice that other controls that have been associated with keys have a small key indicator on the control.

   Keys with their translations are stored as part of the model within the page definition.
Figure 9. Specifying Individual Text Strings that need Translation.

Figure 10. Using the Translation Editor to Create Keyed Strings.

You can also create keys independently from controls, and then assign the keys to controls.
2. Construct the runtime mechanism that provides access to the externalized strings once they have been translated.

Once keys have been created in the model, and you construct the application, Rational Rapid Developer creates resource bundles. The use of resource bundles is a standard mechanism to provide translated strings to an application at runtime. Resource bundles are name/value pairs that define the key and the translated value for a specific language. Resource bundles are created for each language that you have defined in the application Globalization Model.

The process of externalizing strings involves changes to the model, not to the code. Since the code is generated, the chances for errors are greatly reduced.

Encoding Issues

Setting the encoding involves making decisions about the how data is to be interpreted at various conversion points within the application architecture. The diagram below shows the different elements in an n-tier Web application architecture. The conversion points are:

1. Between the browser and the JVM (application server) – this is for data and for information passed as query string parameters,
2. Between the application server and the database (through JDBC)
3. Between the application server and other data sources such as messaging transports.

Figure 11. Encoding Conversions within the Application Architecture.
You can see from the diagram that you can use Unicode or code page encoding. Encoding can be set at different points in the application architecture. If you don’t take care to match the encodings between conversion points, your data will not be interpreted correctly.

Because there are different technologies involved in Web architectures, setting the encoding may be dependent on other systems. For example, some databases require that you set the encoding when the database is created. XML documents that are sent through message transports require the encoding to be set in the XML header. The following illustration shows the settings that are available for XML documents from the Rational Rapid Developer Message Model.

The data that is passed between the browser and the application server is the data that makes up the page, and optionally parameters that are passed on the URL. Since this data can be in multiple locales, you need to be able to set an encoding property for the data and for the parameters on the URL.

To set the data encoding, you need to set it on the browser. For example, you can set the encoding for Microsoft Internet Explorer using an Encoding submenu on the View menu. The trick, however, is that you want the page being delivered to the browser to set the encoding for the browser, not the end user. Rational Rapid Developer enables you to control this setting with the encoding property on the page property tab, as shown in the following illustration.
User Interface Issues

A common approach to localization of Web pages is to create a new page for each language. Unfortunately, this creates a large number of pages that need to be maintained. When a modification is made to one page, it will most likely need to be made to the corresponding pages in different languages. This is a maintenance nightmare and an error-prone methodology.
Your internationalization goals should be to be able deliver multiple languages from a single page. This greatly reduces the number of pages and the amount of maintenance required when modifying the application. Rather than making similar modifications for each page that represents a given language, you make a set of changes and the application delivers the page in the requested language.

Rational Rapid Developer facilitates this more efficient methodology by allowing you to view the same page in different languages. During runtime, the page appropriate for the request is delivered to the browser.

**Formatting Issues**

Formatting in a globalized application can be very complex. The goal is to provide one set of code that reflects whatever locale or cultural settings required for an instance of a page that is being delivered to a browser.

Java has a robust API to handle this situation; however, navigating through all of the classes and learning how to use the API can be tedious. Rational Rapid Developer provides you with a Formatting Model that allows you to create format descriptors for each locale and each formatting type. Formatters are available for numbers, currency, percent, date, time, and date-time.

The Rational Rapid Developer Formatting Model is shown in the following illustration. You can see that every locale has a default format for each formatting type. You can also create your own format descriptor for any format type.
The Formatting Model enables you to establish a set of format descriptors. You need to use these descriptors to control the presentation or the data entry format for specific data items on a page.

The following illustration shows how you can do this within the property sheet for a control. By simply assigning a format descriptor to a control, Rational Rapid Developer will know the formatters, calendars, and other items to create when constructing the underlying code.
Besides presentation, you should also be aware that formatting descriptors are used to set the format for data that is being entered into the page and its subsequent validation. Validation can be in the form of client validation through Client Side JavaScript and Server side validation.

Managing Images
Images must also be handled correctly in a multi-lingual environment. Rational Rapid Developer provides a flexible yet easy way to prepare for the localization of images. You can specify the relative location of a set of images that need to be presented on a page, based on the response locale for the page. Notice in the following illustration that the relative path of the image is specified with a macro - `<%CurrentLocale%>`. This macro is evaluated at design time or at runtime. This gives you the ability to create locale-based folders that store sets of localized images.

Managing Error Messages
You can't complete the globalization of your application without preparing error messages that can be presented in multiple languages.

Rational Rapid Developer facilitates the globalization of error messages in several ways:

- By providing a model-based mechanism for internationalizing client-side JavaScript messages for data entry forms.
The following illustration shows the properties that can be set for a required data entry field. The string associated with the key “testpage_L1011” will be delivered as a client side dialog box when you attempt to submit a form without entering something for the required field. Rational Rapid Developer creates and delivers the dialog box localized for the response locale for the page at the time the error occurs.

- By providing a general interface that can be used in any place in your custom methods.

```
ctx.getResourceString("G12")
```

The method returns a localized string based on the page response locale.

**Other Internationalization Features**

Rational Rapid Developer offers a robust set of features to deal with the challenges presented by globalization. Other available features include:

- **Sorting**
- **Selection**
- **Section 508 Help**
- **Help**
- **Multiple Languages on a single page**
- **Internationalized enumerations**
- **Localized auto-constructed images.**

**Localization**

Once you have prepared the application for different languages, you will need to localize the application.
The process of localizing an application can be managed through Globalization Management Systems (GMS). These systems manage the process of localization through a pre-defined workflow. The translation is one step in the workflow.

Rational Rapid Developer provides several features that greatly facilitate the translation process. They are:

- *The ability to view translation and localization impact at design time*
- *The management of translation strings into your application*
- *The granularity offered for the integration of translations*

**Design Time vs. Runtime Testing**

When developing a localized application, the designer / developer / tester / translator (members of the development team) must be able to test the pages in multiple languages. Not only are they testing the pages for proper translation, but they are also focusing on layout consistency issues. Often, the developer must wait until the page can be viewed at runtime to catch any layout conflicts.

With Rational Rapid Developer, you can detect layout conflicts at design time by toggling between different languages in the Page Architect page-authoring module. Design-time review saves time by catching layout defects earlier in the process.

**Translation**

To translate your application strings to multiple languages, Rational Rapid Developer offers a convenient method of exporting keyed strings from your application to a variety of formats. The resulting file is sent to translators.

In the following illustration, four strings that need to be translated into Japanese are selected in a specific page. The strings will be sent to the translators in XLIFF format. You also can select TMX, Excel, or CSV export formats.
Once the translated files are returned from the translators, you simply place it in a “Translated” directory in the application directory. Rational Rapid Developer automatically imports the strings and integrates them back into the model. You can inspect the translation and optionally select the translated strings that you want to incorporate back into the model.

### Translation Granularity

In most translation scenarios, you create an application in one language and then send out all the application strings to several translators. Since it is so easy to prepare strings to be translated and to incorporate the translations back into the model, Rational Rapid Developer gives you the choice to translate on a page-by-page basis at any time during your development process. Your development process can include a check to see if a page is ready for translation. As long as the translators are available, they can provide the translations as different parts of the application are ready for translation. There is no need to wait until the entire application is completed. There may be reasons to wait for the complete application to be finished but at least with Rational Rapid Developer, you are not trapped into this workflow due to the complexity of localizing hand-coded applications. You now have a choice.

### Construction

At any point in the process you can construct your entire application or individual pages, messages, components, or Web services. Rational Rapid Developer takes the model specifications and constructs optimized code for the target platform.
Conclusions

There are many issues involved in application globalization. Rational Rapid Developer addresses these issues and makes it far easier to create and maintain applications that can be delivered in any number of languages. The Rational Rapid Developer model-driven, architected RAD development environment ensures productivity in design and implementation. It provides an abstraction from the complexity involved not only in building an n-tier Web application, but also in building an application that supports multiple languages.
Rational Unified Process for Systems Engineering

Part II: System architecture

by Murray Cantor
Principal Engineer
Rational Brand Services
IBM Software Group

Last month we began a three-part series to provide an overview of the latest evolution of Rational Unified Process for Systems Engineering,® or RUP SE.® RUP SE is an application of Rational Unified Process,® or RUP,® software engineering process framework. 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.

Part I included a discussion of systems, the challenges facing the modern systems developer and how RUP SE addresses them, RUP SE Unified Modeling Language (UML)-based modeling and requirement specification techniques, and the use of UML semantics. This month, in Part II, we will focus on system architecture and introduce the RUP SE architecture framework, which describes the internals of the system from multiple viewpoints. Part III, to be published in October, will cover requirements analysis and flowdown, an introduction to the method for deriving requirements, and specifications for the elements of the RUP SE framework. This will include a description of the Joint Realization Method, a novel technique for jointly deriving the specification of architectural elements across multiple viewpoints. Part III will also include a discussion of RUP SE programmatics.

Editor's note: The RUP SE v1 Plug-In was made generally available in 2002, and v2 of this plug-in was made available in June of 2003. Although the information in this series is consistent with v2, the articles do discuss a few possible extensions to the process framework. Please note that the RUP SE Plug-In -- v1 and v2 -- is downloadable from IBM Rational Developer Network (authorization required).
Definitions

A clear understanding of RUP SE is impossible without a grounding in several terms and concepts. [Other standards may define these terms differently; what we strive for here is internal consistency.]

**System decomposition:** Successful system engineering relies on the ability to reason about many things at once. System-level decomposition is one powerful technique for accomplishing this.

A system may be decomposed in two ways:

- Into further systems using logical decomposition; this is the so-called "systems of systems" decomposition.
- Into system components that make up the delivered system.

**System model dimensions:** A RUP SE system model has two dimensions, which allow for separation of concerns by different teams involved in the design and construction of the system.

- **Viewpoint dimension:** the context for addressing a limited set of quality concerns.
- **Model level dimension:** UML diagrams that capture a specific level of design detail.

**Model:** A representation of a system, including views that capture all areas of concern, levels of specificity, and model entity relationships.

**Model level:** The level of abstraction at which each model may be constructed, from the more general -- hiding or encapsulating detail -- to the more specific -- exposing more detail and explicit design decisions.

**Viewpoint:** As the name implies, a viewpoint is a notional "position" from which some aspects or concerns about the system are made visible, implying the application of a set of concepts and rules to form a conceptual filter. To understand a system, it is usually not sufficient to examine the actual system itself, which is why models are constructed to represent the various viewpoints involved.

**View:** A projection of a model level that shows entities that are relevant from a particular viewpoint. These projections will typically be illustrated by diagrams of some kind. The intersection of viewpoint and model level (of abstraction) will contain (or at least identify) views of model(s) relevant to that viewpoint (concern) at that level of abstraction.

**Viewpoints**

The RUP SE framework provides a set of viewpoints, as expressed in Table 1.
<table>
<thead>
<tr>
<th>Viewpoint</th>
<th>Expresses</th>
<th>Concern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worker</td>
<td>Roles and responsibilities of system workers</td>
<td>• Worker activities</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Automation decisions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Human/system interaction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Human performance specifications</td>
</tr>
<tr>
<td>Logical</td>
<td>Logical decomposition of the system as a coherent set of UML subsystems</td>
<td>• Adequate system functionality to realize use cases</td>
</tr>
<tr>
<td></td>
<td>that collaborate to provide the desired behavior</td>
<td>• Extensibility and maintainability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Internal reuse</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Good cohesion and connectivity</td>
</tr>
<tr>
<td>Physical</td>
<td>Physical decomposition of the system and specification of the physical</td>
<td>• Adequate physical characteristics to host functionality and meet</td>
</tr>
<tr>
<td></td>
<td>components</td>
<td>supplementary requirements</td>
</tr>
<tr>
<td>Information</td>
<td>Information stored and processed by the system</td>
<td>• Sufficient capacity to store data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Sufficient throughput to provide timely access to the data</td>
</tr>
</tbody>
</table>
The viewpoints in Table 1 are some of the most common ones for software-intensive systems. Many system architectures require additional viewpoints that are domain-specific: safety, security, and mechanical viewpoints, for example.

Viewpoints represent different areas of concern that must be addressed in the system architecture and design. If there are system stakeholders or experts whose concerns are important to the overall architecture, there likely is a need for a set of views to capture their design decisions.

It is important to build a system architecture team with staff members whose skills will enable them to manage the various viewpoints. The team might include business analysts and users who take primary responsibility for the worker viewpoint, software architects who attend to the logical viewpoint, engineers who concern themselves with the physical viewpoint, and experts on domain-specific viewpoints.

**Model levels**

In addition to viewpoints, building a system architecture requires levels of specification. As the architecture is developed, it evolves from a general, abstract specification to a more specific, detailed specification. Consistent with RUP guidelines, there are four architectural model levels in RUP SE, as described in Table 2.

### Table 2: RUP SE model levels

<table>
<thead>
<tr>
<th>Model Level</th>
<th>Expresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context</td>
<td>The system and its actors</td>
</tr>
<tr>
<td>Analysis</td>
<td>Initial system partitioning in each of the viewpoints to establish the conceptual approach</td>
</tr>
<tr>
<td>Design</td>
<td>Realization of the analysis level to hardware, software, and people</td>
</tr>
</tbody>
</table>
Implementation Realization of the design model into specific configurations

Through these levels, the design goes from the abstract to the physical. The context model level captures all of the external entities (actors) that interact with the system. These actors may be either external or internal to the enterprise that deploys the system. In either case, the actors may be human beings or other systems. At the analysis level, the partitions do not reflect choices of hardware, software, and people. Instead, they reflect design approaches for dividing up what the system needs to do and how the effort should be distributed. At the design level, decisions are made regarding the sorts of hardware and software components and worker roles that are needed. At the implementation level, specific choices of hardware and software technology are made to implement the design. For example, at the design level, a data server is specified. At the implementation level, the decision is made to use a specific platform running a specific database application.

It is important to maintain traceability among these levels. As the enterprise or mission changes, the context level views need to be amended, along with any affected lower-level views. As the underlying technology changes, the implementation level and possibly the design level can be affected. In brief, the impact of enterprise changes flows down, whereas the impact of technology changes flows up.

**System architecture views**

The next step is to capture the system architecture in a set of views that express the architecture from various viewpoints and model levels. Each of the cells in Table 3 provides a view of the system. Note that at the implementation level, a single diagram captures the realization of hardware and software components for each system configuration.

**Table 3: RUP SE model framework**

<table>
<thead>
<tr>
<th>Model levels</th>
<th>Model viewpoints</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Worker</strong></td>
<td><strong>Logical</strong></td>
</tr>
<tr>
<td>Context</td>
<td>UML organization view</td>
</tr>
<tr>
<td>Analysis</td>
<td>Generalized system worker view</td>
</tr>
</tbody>
</table>
### Design System

<table>
<thead>
<tr>
<th>Design</th>
<th>System worker view</th>
<th>Subsystem class views</th>
<th>System data schema</th>
<th>Descriptor node view</th>
<th>Detailed process view</th>
</tr>
</thead>
</table>

### Implementation

**Worker role specifications and instructions**

Configurations: deployment diagram with software and hardware system components

---

The relationships among model levels, viewpoints, and views can be seen in Figure 1 below.

![Figure 1: Model levels, viewpoints, and views](image)

The domain-specific viewpoints also should have artifacts in place for one or more of the levels. The set of project artifacts within this framework should be a part of the project development case. Let us briefly investigate each of the viewpoints.

**Note:** At this writing, UML 2.0 is being readied for adoption and OMG has released a Request for Proposal for a Systems Profile for UML. Once UML 2.0 and the Systems profile are adopted, the RUP SE view semantics will be updated to take full advantage of those standards.

**Worker viewpoint**

Workers are sufficiently unique to warrant their own viewpoint. Workers are both logical and physical entities. They are logical in that they can, when instructed, provide services and collaborate with other logical entities. They are physical in that they are limited in terms of performance, responsiveness, and capacity. Of course, workers are blackbox entities not subject to further subdivision in the model.
Reasoning about how workers interact with the automated portions of the system and each other is a system engineering specialty. The worker viewpoint provides the setting for this reasoning.

Also, note that system workers are not the same as business workers. System workers are human beings who are a part of the system. They are not system actors, as they are partially responsible for delivery of the system services. In the RUP SE framework, system workers are represented as stereotyped classes. They may be associated if they have dependencies on one another or some other relationship. In the generalized system view, generic system workers are expressed with low detail.

In some applications, it is useful to introduce an abstraction of the automated part of the system -- the machine, which differs from a general system in that its realization contains no workers. The generalized workers and the machine may be used in the flowdown workflow to determine worker specifications and to reason about automation decisions. Figure 2 shows an example of a worker diagram.

![Worker Diagram](image)

**Figure 2: A RUP SE worker diagram**

For example, if you were modeling a system for a ship, at the analysis model level you might represent a sailor as a general system worker. At the design level, however, you might define a multitude of specific sailor roles.

**Note:** You may want to include an additional stereotyped classifier in the worker diagram -- a machine -- to support automation decisions. In the joint realization method discussed below, the machine would perform whitebox logical steps to be supported through automation.

**Logical viewpoint**

The logical viewpoint is the most familiar to object analysts. It describes, at different levels of abstraction, the kind of objects that realize the system. The elements of the views in the logical viewpoint are classes and UML subsystems. In UML 1.4, systems and subsystems inherit from classifiers and packages; there is no UML syntax that captures both the classifier and package aspects of a subsystem. Normally in UML, subsystems are represented as packages with dependencies. In RUP and RUP SE, proxy classes are used to represent the classifier semantics. In RUP SE, we stereotype the proxies and the packages as systems or subsystems as
appropriate, and, as appropriate, add the system semantics described above to subsystems. Figure 3 shows a UML subsystem view for a click-and-mortar retail system using the common notation. One could choose to use subsystem classifiers in place of the packages in this figure.

![Figure 3: Click and mortar subsystem diagram](Click to enlarge)

**Physical viewpoint**

In systems engineering, the physical resources are a part or aspect of the system. It follows that semantics need to be provided to reason about the properties of the elements of the physical realization of the system. More specifically, the outcome of a systems engineering environment includes a detailed specification of the hardware to be built or acquired. Note that systems engineering does not include the hardware engineering disciplines (mechanical, electrical) but does include sufficient specification to be used as input to the hardware design team.

As shown in Table 3, RUP SE uses an analysis level, physical viewpoint diagram called *System locality view*. In the physical viewpoint, the system is decomposed into elements that host the logical subsystem services. Locality diagrams are the most abstract expression of this decomposition. They express where processing occurs without tying the processing locality to a specific geographic location, or even the realization of the processing capability to specific hardware. Locality refers to proximity of resources, not necessarily location, which is captured in the design model. For example, a locality view might show that the system enables processing on a space satellite and a ground station. The processing hosted at each locality is an important design consideration.

The locality diagrams show the initial partitioning, how the system's physical elements are distributed, and how they are connected. The term *locality* is used because locality of processing is often an issue when considering primarily nonfunctional requirements.
As shown in Figure 4, locality diagrams consist of two elements:

- **Localities**: groupings of physical resources that enable a conceptual, physical partitioning of the system. Their icon is a rounded cube.

- **Connections**: linkages between the localities that may be used to pass data, service requests, or I/O entities. Connections are represented in UML as stereotyped associations.

![Diagram](image)

**Figure 4: Locality diagram elements**

**Locality semantics**

Localities are used to realize the physical characteristics of the system class, and their semantics derive from those associated with the physical nature of the system. In particular, localities have class and instance attributes, and measures of effectiveness captured as tagged values. Localities have two default sets of tags:

- **Quality**: reliability, availability, performance, capacity, and so on
- **Management**: cost and technical risk

These locality characteristics form a nominal set. Each development team should determine the best set of characteristics for their project. This determination could be a development-case-specification activity.

Locality characteristics are set to meet their derived requirements. There is a subtle difference between characteristics and requirements. For example, for good engineering reasons, you might specify a locality that exceeds requirements.

In the section on **Localities, services, and interfaces** below, we will show that localities host subsystem services.

**Connection semantics**
Localities are joined by connections, which represent the physical linkages between localities. Connections are stereotyped associations with tagged values, again capturing characteristics. Nominal connection tags are:

- **Throughput**: transfer rate, supported protocols
- **Management**: cost, technical risk

Since localities host services, connections must pass service invocations. In fact, there are at least three types of flow we have to consider in systems:

- Control flow
- Data flow
- Material flow

Consider, for example, the throttle in an automobile. The throttle linkage is the control connection that transmits the service requests (open or close) to the throttle. The gas line is also a connection to the throttle. The gasoline itself is not a service request, but rather a raw material used by the throttle to perform its services. Finally, there may be a network data connection to the throttle containing an ongoing stream of environment and automobile status data that is used to adjust the response to the throttle.

**Localities and nodes**

Recall that UML nodes are classifiers that have processing ability and memory. Used in deployment diagrams, the UML node semantics support reasoning about the hosting processors for the software components. The implicit assumption is that the physical resources are outside the software under consideration. For example, in software engineering, the hardware is often seen as an enabling layer below the operating system.

The UML does provide design and implementation-level artifacts for deployment diagrams:

- **Descriptor diagrams** for the design level
- **Instance diagrams** for the implementation level

In particular, instance deployment diagrams are meant to capture configurations and actual choices of hardware and software, and to provide a basis for system analysis and design, serving as an implementation level in the physical viewpoint. The *UML Reference Manual* describes an instance version of a deployment diagram as "a diagram that shows the configuration of run-time processing nodes and component instances and objects that live in them."

In RUP SE, this intent is preserved. A node, then, is a special sort of locality that is used at the design and implementation model levels to specify physical resources that execute software. However, as a kind of locality, RUP SE nodes can be stereotyped to include all of the locality semantics. Note that these semantics differ from standard nodes in UML. Localities are not so much stereotyped nodes as nodes are stereotyped localities. UML 2.0 will provide better means for dealing with
Localities, services, and interfaces

A locality specifies the physical resources that provide logical services. In practice, each locality will provide a subset of the services of one or more of the logical subsystems. The determination of those services is an outcome of the joint realization workflow we will describe below.

The set of hosted subsystem services for a given locality can be captured in a couple of ways:

- Survey of hosted subsystem services document
- Associated subsystem interfaces

The first method is simpler, associating a requirements document with a locality. The second requires a more sophisticated use of the UML. Subsystems are classifiers, and their services are classifier operations. In addition, the UML allows operations, and therefore subsystem services, to be organized into interfaces. That is, an interface is a subset of subsystem services. In this second approach, one defines the needed interfaces for each of the subsystems and then assigns them to the appropriate localities. Generally, there will be more than one interface associated to a locality.

Design trades

"Design trades" is the name of a common system engineering technique: building a set of alternate design approaches; analyzing the cost, quality, and feasibility of the alternatives; and then choosing the best solution. The locality view supports design trades by containing more than one locality diagram, each representing a different conceptual approach to the physical decomposition of the system.

Figures 5 and 6 are locality diagrams that document different engineering approaches to a click-and-mortar enterprise with a number of retail stores, central warehouses, and a Web presence. The first solution (Figure 5) depicts processing capability in the stores. The second solution (Figure 6) shows all terminals connected directly to a central office processor. In each case, characteristics can be set for the localities that are required to realize the design. Today, most people would agree that Figure 5 represents a better design; however, the solution in Figure 6 may be considered superior in a few years.
Locality decomposition and realization

Like subsystems, localities can be decomposed hierarchically into further localities. It is tempting to use aggregation to associate the localities with the sublocalities. However, there is a critical difference between the whole-part relationship in a physical decomposition and the relationship normally expressed with class aggregation: In common usage, when a class object (whole) is aggregated from
other class objects (parts), the whole's attributes include the attributes of the parts. The attributes of the whole in a system locality are \textit{functions} of the attributes of the parts. A simple example is that the weight of the whole is the sum of the weight of the parts. Often the relationship between a whole attribute and a part attribute is much more complex, yet there are no current semantics in the UML to express functional relationships between attributes. A workaround for capturing the relationship can be inserted in the model, using private attributes and operations that carry out the functions.

When realizing localities as physical components, we suggest that the realization be hierarchical. That is, each component is part of the realization of no more than one locality. Otherwise, it is difficult to maintain the traceability of derived nonfunctional requirements between localities and the components. However, you need not follow this suggestion if you want to have reusable components across localities. In this case, of course, the components will have to meet the most stringent requirements, discovered through flowdown of the system requirements across localities.

In addition, in some cases it makes sense to create multiple realizations of the same locality, flowing down to multiple system implementations. For example, one might have a product line with different implementations to support a range of price/performance points.

\textbf{Information viewpoint}

The use of UML for both object and relational database modeling is a well-developed practice that RUP SE makes use of in the information viewpoint. Note that maintaining the database modeling in the system model permits overall system coherency by supporting associations between data and functional classes, and by assigning database components to localities.

\textbf{Process viewpoint}

The process viewpoint is also represented using standard UML. Figure 7 shows an example of a system process view.
Moving between model levels

Moving down model levels adds specificity, not accuracy, to the models. At each level, you need to be as accurate as possible in specifying model elements, because accuracy at each level adds to the understanding of the system and discipline of the process. As you move down the levels, each view is a more specific decision, resulting in configuration items at the implementation level. It is important to note that the model elements at one level establish the requirements at the next level. Or, as indicated in Figure 8, we can say that each model level realizes requirements discovered at a higher level. For example,

- The analysis model level shows how requirements specified in the context model level are met.
- The design model level shows how requirements arising from the system analysis model level are met.
- The implementation model level meets design specifications.
Figure 8: Lower model levels realize requirements established at upper model levels

Click to enlarge

Figure 9 shows an example of how the physical viewpoint at the design level contains a descriptor node diagram, which shows a physical design that realizes each locality.

Because each model level establishes requirements or specifications to be realized at the next lower level, you can maintain traceability between levels by capturing how design elements meet those specifications.

In practice, as a team develops a model level, they may likely discover that the upper level should be revised because, for example, one or more elements it
specified cannot be realized. Hence, as development proceeds, no level is ever really "frozen"; each is maintained throughout development. However, as development proceeds, the focus of the effort typically moves down, level by level.

**Notes**


2 Ibid, p.252ff.

3 Ibid, p.455.

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

Copyright [Rational Software](#) 2003 | [Privacy/Legal Information](#)
Agile RUP for non-object-oriented projects

by Gary Evans
President
Evanetics, Inc.

The Unified Modeling Language (UML) and IBM Rational Unified Process,® or RUP,® are standard fixtures on most object-oriented projects today. UML is a notation for visually expressing object-oriented concepts; RUP is a process framework that uses the UML to express the content of its artifacts and tasks. But what do RUP and the UML offer that portion of our software community doing non-object oriented development? This article is my answer to that question. As we shall see, RUP and UML can be used quite effectively on software projects in COBOL, C, or even (the dreaded) assembler language.

RUP values and practices

RUP is not so much a defined process as it is a description of a process framework. Because it is a generic framework, it is large and complex. But putting aside its substantial footprint (more than 3,000 files in more than 200 folders), RUP incorporates a very simple vision based on six basic principles, or what IBM Rational calls software development best practices:

- Develop iteratively
- Manage requirements
- Use component architectures
- Model visually
- Continuously verify quality
Manage change

These core principles characterize the "personality" of RUP. They dictate that an effective process should be:

- Iterative -- Do the same activities in small pieces, over and over.
- Incremental -- Plan to gain a bit more understanding of the problem, and add a bit more of the solution, at each iteration, building on what you did previously.
- Risk-focused -- Address risk early and often. Focus on the most architecturally significant properties of the system, including the highest risk areas, before developing the easy, "low hanging fruit" of the system.
- Controlled -- Always know where you are, where you are going, and how far you have yet to go.
- Use-case (i.e., requirements)-driven -- Establish the goal of the system by capturing all its requirements; and capture the operational requirements in use cases.
- Architecture-centric -- Emphasize architectural stability over software design details.

For software groups doing procedural development within the traditional "waterfall" process, adopting the six RUP principles can seem a daunting challenge. Why should an organization give up all the techniques they know for new practices they don't know? This is a crucial question, and I will come back to it later to offer an answer.

Agile development values and practices

RUP has an additional characteristic that is implicit in the properties listed above: It is a prescriptive process definition. By prescriptive, I mean it prescribes (literally, it's all written down for teams ahead of time) the tasks, workers, and artifacts that a project using RUP should follow. Now, being prescriptive is not an evil thing. For many organizations used to developing software via the Indiana Jones process ("I'm making it up as I go, kid"!), the checklist and tour guide that RUP offers can dramatically reduce a team's anxiety in transitioning to an actual, defined process. However, I have also worked with way too many organizations that, in deciding to use RUP, thought they had to use all of RUP on every size project. Ouch! The sheer mass of content in RUP is enough to sink any project that tries to cover all of it.

Over the past four years, a grass-roots reaction to
Avoid them like the plague. Global variables are too often a result of bad design.

- **Public variables.** Non-static variables defined within a (class) file can be used to simulate class-scope variables. But don't overdo it!
- **Private variables and functions.** Declare these as static, making them accessible only to other functions in the same file (which is a class).
- **Public functions.** Declare normally without static. Use naming conventions to link the function to a specific class.
- **Global functions.** Limit them to zero, if possible.
- **Naming conventions.** Follow the conventions in this article: Name functions according to `<classname>$<functionname>(args)` to reflect the object-organized approach you are following.

Finally, **use DLL or shared library conventions** to hide access.

---

Prescriptive processes has swept our industry. These insurgent, counter-cultural processes are known collectively as **agile** development processes. Current agile contenders include:

- **Crystal** ([http://alistair.cockburn.us/](http://alistair.cockburn.us/))
- **SCRUM** ([http://www.controlchaos.com/](http://www.controlchaos.com/))
- **Feature-Driven Development** ([http://www.featuredrivendevelopment.com/](http://www.featuredrivendevelopment.com/))
- **Agile Modeling** ([http://www.agilemodeling.com/](http://www.agilemodeling.com/))

All these processes share the same strategic vision - to enable teams to develop high-quality software as quickly as possible -- but they differ in their tactics. What unites them under the rubric of agile development process is a philosophy that I characterize simply as "If you need to do something, then do it -- otherwise, don't."

Agile approaches openly rebel against the "tyranny of the defined." For example, just because the waterfall process says you should try to capture all your requirements up front, why is that the best way to capture them? Or, UML and RUP define the syntax and value of statecharts, but why should you have to do statecharts for every class, on every project? Of course, you don't. Yet the agile development approach questions everything about business-as-usual when it comes to software development. And the questioning is not like that of a rebellious teenager; rather, it comes from a perspective of humility, and it leads to the honest conclusion that there is no single, best way to develop software.

**The union of RUP and agility**

Recent releases of IBM Rational Unified Process, or RUP, have addressed this "tyranny of the defined" by offering "conceptual roadmaps" -- suggested profiles for applying RUP in small projects, e-business development, extreme programming, and so forth. The RUP Roadmaps are still somewhat prescriptive, but they can be immensely helpful as
a jumpstart for development teams working on projects that match these profiles.

With their inherent differences, is it conceivable that RUP and agile approaches could ever co-exist? Absolutely. IBM Rational has recently produced numerous papers on how to make RUP more agile. I also published a *Rational Edge* article about an agile approach to RUP that I use on all of my projects (see "A Simplified Approach to RUP" at http://www.therationaledge.com/content/jan_01/t_rup_ge.html).

Making RUP agile is not a violation of RUP philosophy; it simply requires that we make our own decisions about what we will use and do on our projects. If you approach RUP as a cafeteria menu rather than as a set of inscribed stone tablets, you will do fine.

**A new perspective for non-OO developers**

"Ok," you may say, "I have a cafeteria mentality, and I routinely question everything. How does this help me develop a COBOL application using RUP and UML? I don't have any objects in these languages." True enough, but you don't have customers, purchase orders, bills, or price quotes in your COBOL or C program, either. Yet these are concepts that everyone uses when talking about programs and business models. And your design thinking shows an *intent* to capture these concepts.

At this point in our discussion we must make some very important distinctions. We must discipline ourselves to think from qualitatively different *perspectives*, as Martin Fowler has expressed: ¹

- **Conceptual Perspective**: Represents the concepts in the domain of interest.
- **Specification Perspective**: Captures the interfaces of the software, not the implementation.
- **Implementation Perspective**: Captures the embodiment of the software, including internal implementation.

These distinct perspectives are crucial in our attempt to apply object-oriented techniques to non-object oriented development projects. When we are exploring our system-to-be from either a *conceptual* or *specification* perspective, the implementation language, algorithmic details, and platform constraints just don’t matter yet. It is easy to look at UML as a notation system and subtly conclude that it is tied to an implementation language. But it isn't -- in fact, the power of UML is a direct consequence of its ability to help you assume a conceptual and a specification perspective on your system-to-be.

These facts are always true:

- The development language we choose is not the focus of our business.
When we implement, we are always implementing some kind of conceptual or specification model.

Developers think in terms of these conceptual models, regardless of what development language they ultimately use.

UML lets us bring this conceptualization to the forefront of our thinking.

It should be evident that the pre-coding tasks in which requirements capture, analysis, and design occur all produce conceptual models as outputs, regardless of the implementation language. In the procedural world, these conceptual models would include data flow diagrams, flowcharts, logical data models, control flow diagrams, context diagrams, and so forth.

But conceptually, we can regard a COBOL billing system or a Call-record Tracking System as objects or components that:

- Have defined functionality.
- Are accessed according to specific interfaces.
- Should hide their implementation details.
- Have defined structure (the data they own and operate on).

**The object-organized approach**

Even the notational gap between the procedural/structured world and the UML world is not really as large as we might first think. Table 1 shows how the traditional modeling elements of structured analysis/structured design map to the UML diagrams.

**Table 1: Mapping structured artifacts to UML**

<table>
<thead>
<tr>
<th>Structured artifact</th>
<th>Description</th>
<th>Maps to</th>
<th>UML artifact</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context Diagram</td>
<td>Shows external data sources and sinks, and the data exchanged. (Highest level DFD)</td>
<td>➡️</td>
<td>(no map)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Diagram Type</th>
<th>Description</th>
<th>Related Diagrams</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Flow diagram</td>
<td>Shows flow of data through the system, and transformations of that data by system processes.</td>
<td></td>
<td>(no map)</td>
</tr>
<tr>
<td>Event Diagram</td>
<td>Shows events exchanged with processes outside the system.</td>
<td>Use-Case Diagram</td>
<td>Describes users of system (actors) and services the system provides to these actors.</td>
</tr>
<tr>
<td>Control Flows</td>
<td>Shows sequence of function activation within a system.</td>
<td>Sequence Diagram; Collaboration Diagram; Activity Diagram</td>
<td>Capture the dynamic (behavioral) properties of the system.</td>
</tr>
<tr>
<td>Functional Specification</td>
<td>Lists major functions of system and how the system interacts with users.</td>
<td>Use-Case Diagram; Use Cases</td>
<td>Describes sequence of actions between system and actors to carry out work of observable value to the actor.</td>
</tr>
<tr>
<td>State Transition Diagram</td>
<td>Shows states for the system's processes.</td>
<td>Statechart Diagram</td>
<td>Describes the states (i.e., constraints on behavior) of classes and objects in the system.</td>
</tr>
<tr>
<td>Structure Chart</td>
<td>Shows modules and their static relationship (derived through functional decomposition) to one another and other system elements.</td>
<td>Class Diagram; Component Diagram</td>
<td>Describes static relationships among classes or implementation modules.</td>
</tr>
</tbody>
</table>
We might dispute the exact extent of the mapping shown, but both sets of artifacts attempt to present a multi-dimensional, qualitative perspective of our software system. And, if there is such a large intersection in the semantic content of these various diagrams, we can assume that it is possible to use the UML diagrams readily within an agile RUP process -- even when implementing the target system in COBOL. In other words, you can actually pursue an object-organized approach to development even when you cannot be truly object-oriented.

Let's look at a project in which I did exactly that.

A case study

Using an agile RUP approach, the development team and I pursued this project according to the outline in Table 2. But this outline was just our guideline, not a rigid prescription. If we felt the need to change a task for good reason, we did. Note that this outline is focused on the software definition and development tasks. I have omitted the tasks associated with project management, configuration management, and so forth. In the space of this article I cannot show every step we performed on the project, but I will cover the major steps.

**Table 2: Outline of agile RUP steps**

**Getting Started:**

a. Get a handle on the scope and goals of the system.

b. Identify the initial packages that are part of the system being modeled.

c. Identify the major functional and non-functional requirements for the system.

d. Develop the major use-case descriptions.
Identify the major classes/components that are part of the domain being modeled.

f. Define the responsibilities and relationships for each class in the domain.

g. Construct an initial analysis class diagram.

h. Identify the major risk factors and prioritize the most architecturally significant use cases and scenarios.

i. Partition the major use cases/scenarios across the planned iterations.

j. Develop an Iteration Plan describing each "mini-project" to be completed in an iteration.

**For each iteration:**

1. Construct analysis-level interaction diagrams (i.e., Sequence Diagrams or Collaboration Diagrams) for each scenario in the iteration. (Usually on a whiteboard.)

2. Test and challenge the analysis-level Sequence Diagrams.

3. Develop analysis-level Statechart Diagrams for each class with "significant" state.

4. Enhance Interaction Diagrams and Statechart Diagrams with design-level content (add in technology classes such as collection classes, persistence interfaces, etc.).

5. Challenge the design-level Sequence Diagrams and Statechart Diagrams on paper, discovering additional operations and data assigned to classes; update the Class Diagram with these operations and data.

6. Develop the code for the use cases/scenarios
in the current iteration from the current diagrams.

7. Test the code in the current iteration.

8. Conduct an iteration review: Did you achieve your goal? What went right? What went wrong? What needs to be changed?

Conduct the next iteration (i.e., go back to Step 1), adding in the next set of use cases/scenarios until the system is completely built.

**Getting a handle on system scope and goals**

This case study covers a new taxation subsystem developed for a major cellular telephone company's legacy billing system, which was written in COBOL. The myriad ways of determining taxes were elaborated via in-line code within the billing system code. This was a maintenance nightmare because of frequent changes in state, municipal, and federal tax regulations. Our goal was to isolate the computation of taxes and provide a component-level interface to requestors of tax-calculation services. In addition, the solution had to easily incorporate changes in how taxes are computed, and it had to enable the taxation capabilities to be extended without affecting the remainder of the billing system.

We learned that it wasn't only the billing system that would need the tax-calculation services. Customers often called the company's customer service group to ask how a change to their account (e.g., adding another phone line) would affect their monthly bill, including the taxes applied. So we had not one, but two, immediate requestors of tax-calculation services.

**Identifying the major packages in the domain being modeled**

The team first constructed the UML Package Diagram (Figure 1) to capture our goal: The in-line taxation code would be extracted and isolated into a component-level taxation subsystem. The dashed lines with stick arrow heads denote dependency in UML. For example, the diagram shows that the order system is dependent upon the billing system, meaning that changes to the billing system interface will force changes on the order system.
Identifying the major functional and non-functional requirements

A billing system has dozens or hundreds of functions that it must perform internally, but we focused on the external, public services that the billing system had to provide (its specification perspective) to the requestors of those services. At the service level, the billing system computes taxes of two types: bill taxes on the account itself, and roaming taxes for individual roaming calls made on that account.

We also knew that some administrative functions for maintaining tax data had to be provided as well. We produced a Use-Case Diagram that captured our known requestors (actors) of tax calculation services and the major use cases we had identified.

This was an initial -- and not very elegant -- diagram, but it served our needs. It helped us understand the basics of our development goal. You can see from the use-case diagram that certain architectural and infrastructure decisions are evident even at this early stage of the project:
We would enlist the services of a commercial, industry-standard, third-party tax calculation engine (business constraint), and we made the assumption that we would manage taxes by interfacing with to-be-determined providers of services to administer the tax rule base and the requisite tax tables.

We also had a short list (not shown) of non-functional requirements that described our goals for reliability, availability, scalability, and so forth.

Developing the major use-case descriptions

We chose to develop our first use-case description (see Figure 3) for the Compute Taxes use case, since this was the most important one in a production environment. Although this particular description has more "internal" perspective than a normal use case, actually writing it down for the first time, with input from the team experienced in tax calculations, was a great way to make this knowledge part of our common understanding.

<table>
<thead>
<tr>
<th>System:</th>
<th>Taxation Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Case:</td>
<td>Compute Taxes</td>
</tr>
<tr>
<td>Actors:</td>
<td>Tax Client, Tax Calculation Engine</td>
</tr>
</tbody>
</table>

Basic Course

Initialization:

1. This use case begins when the Tax Client requests calculation of tax amounts for either Bill taxes or Roaming Call taxes.

Process:

2. The system verifies that the Tax Client has supplied the required data for tax computation. If this request is for telephone Bill taxes, the system obtains any required information not supplied by the Tax Client. When all information has been obtained, the system verifies the correctness and consistency of the information.

3. The system requests the Tax Calculation Engine to compute the tax amounts, using the supplied tax information. If this request is for telephone Bill taxes, the system...
performs any post-processing on those tax amounts (e.g., resetting tax amounts for tax-exempt customers, or computing tax-on-tax amounts).

**Termination:**

4.
   The system returns the tax amounts to the Tax Client, and this use case ends.

**Exceptions**

   **Exception: Error Detected**

1.
   If an error is detected at any point in the tax preparation or calculation, the system will inform the Tax Client of the error. This use case ends.

---

**Figure 3: Use case description for Compute Taxes**

At this point, just a couple of days into the project, we had determined:

- Our major use cases (Compute Taxes and Manage Taxes)
- Two scenarios within the Compute Taxes use case (Compute Bill Taxes and Compute Roaming Taxes)
- A description of the system's operational personality for the Computing Taxes use case.

And this was about all the documentation we produced. We just didn't need any more at this point.

**Identifying the major classes/components of the domain being modeled**

Our architectural goal in this project was to produce a taxation component. In its simplest definition, a *component* is a software entity that

1. Provides a set of defined functionality as requestable services.
2. Exposes its services through defined public interfaces.
3. Exposes none of its internal implementation.

Figure 4 shows a simple UML diagram illustrating the component interfaces, `Calc_Bill_Tax( )` and `Calc_Roamer_Tax( )`, on the taxation component. As we explored the uses of the taxation component, we learned that the billing system would require access to both Bill and Roaming tax calculations, but the Customer Service System would need access only to the Bill Tax service. The diagram reflects this.

![Figure 4: Taxation component with interfaces](image)

Now our team was at a point where we had to look inside the component and define the classes that actually composed it. Figure 5 shows the initial model the team developed, which consisted of three classes:

- The public `TaxRequest` class that supplies the `Calc_Bill_Tax( )` and `Calc_Roamer_Tax( )` operations.
- The private `TaxGuard` class that assures the integrity of the tax data.
- The private `TaxEngineInterface` class that hides the vendor-specific interface to the Tax Calculation Engine (actor), which actually performs the tax calculations.

![Figure 5: Internal classes of taxation component](image)

**Defining responsibilities and relationships for each class in the domain**
Naming the classes is not sufficient. I adhere rigorously to Rebecca Wirfs-Brock's practice of *responsibility-driven* analysis. In this approach, classes are not characterized by their data or their methods; rather, each class has a defined set of responsibilities that are completely separate from every other class's responsibilities. The methods (operations) and data defined in the class must serve these responsibilities -- not the other way around.

Table 3 shows our classes and their responsibilities:

**Table 3: Classes and responsibilities for constructing an initial analysis class diagram.**

<table>
<thead>
<tr>
<th>Class name</th>
<th>Responsibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaxRequest</td>
<td>● Assures proper sequence of Tax Calculation process.</td>
</tr>
<tr>
<td></td>
<td>● Returns calculated tax amounts, or exceptions, to Tax Client.</td>
</tr>
<tr>
<td></td>
<td>● Performs any needed processing after Tax Calculation engine's calculations.</td>
</tr>
<tr>
<td>TaxGuard</td>
<td>● Assures completeness and integrity of tax-related data.</td>
</tr>
<tr>
<td></td>
<td>● Identifies link area data exceptions that prevent Tax Engine from being called.</td>
</tr>
<tr>
<td>TaxEngine</td>
<td>● Encapsulates the third party Tax Calculation engine.</td>
</tr>
<tr>
<td></td>
<td>● Informs TaxRequest of any errors or exceptions that have occurred during tax calculations.</td>
</tr>
</tbody>
</table>

*Constructing an initial analysis class diagram.*

Given these responsibilities, we quickly developed the UML class diagram shown in Figure 6.
The TaxRequest class has been given a UML stereotype of <<façade>>, indicating that it presents a simplified public interface to hide the complexity of the tax calculations inside the component. But the internal TaxGuard and TaxEngineIF classes have no behavior! Class behavior (i.e., the class operations) are derived from the class's responsibilities. How do we best discover the operations for these internal classes? By constructing dynamic models of our system.

**Constructing analysis-level interaction diagrams**

We next developed two Sequence Diagrams for the two scenarios in our Compute Taxes use case: a) calculate bill taxes, and b) calculate roamer taxes. The Sequence Diagram (SQD, pronounced *squid*) for Calculate Bill Taxes is shown in Figure 7 below:

Notice that a new actor, the Exemption Repository, has been added. As we explored the behavior of the system in calculating bill taxes, we learned that an account may have exemptions from certain taxes, as well as business rules that override the computation of certain taxes. We needed an actor that would provide the service of allowing these overrides, so we added the Exemption Repository actor. This discovery activity is a key...
benefit of developing UML dynamic diagrams such as Sequence Diagrams, Collaboration Diagrams, and Activity Diagrams. Groups that only do static Class Diagrams are severely handicapped in their ability to fully characterize the system under discussion.

**Challenging the interaction diagrams, updating the Class Diagrams**

When we had challenged and accepted this Sequence Diagram, we updated the Class Diagram to add the operations corresponding to the messages in the SQD. To do this, we promoted each message into an operation within the class of the object receiving that message. For example, on the SQD, an instance of the class TaxGuard receives a message named `Examine&Fill()`; therefore, we added an operation to the class TaxGuard to represent a method `ExamineAndFill()`. The updated Class Diagram is shown in Figure 8.

![Updated class diagram for Taxation component](image)

**Figure 8: Updated class diagram for Taxation component**

In the TaxGuard class we have specified two public (+) operations, and a private (-) `SetExemptions()` operation. The business rules of the system made it clear that `SetExemptions()` is called only within the `ExamineAndFill()` operation, and should not be called in any other context.

**Developing the implementation model**

The last step in our process is unique within our method of applying agile RUP and UML to a COBOL or C project. We have to transform our object-oriented thinking into a procedural or structured representation for the target language. Developing the implementation model requires that we have a clear understanding of the scoping semantics of the target language. Since procedural languages do not have native concepts of public or private access, we must use the language features to emulate these concepts.

For example, in COBOL a subprogram is a separately compiled program. The `CALL` verb transfers control to a subprogram linked into the system executable image. How do you hide access to a given subprogram to emulate private access, or prevent access to a class that is internal to a component? The COBOL language allows the programmer to define either internal or external subprograms. Some degree of information hiding can
be emulated through the COBOL rules determining who can call a subprogram, with the `IS GLOBAL` clause on data fields, and by passing parameters `BY REFERENCE` (the default) or `BY CONTENT` -- what C and C++ programmers know as `by value`.

Another topic of some interest is the level of traceability you may want to capture between your object models and your code. In our project, we chose two-way traceability: We wanted to be able to start with the Class Diagrams and map those classes to their implementation in the COBOL programs, or to start with the code and be able to trace the COBOL code back to the classes on the Class Diagrams and interaction diagrams.

Figure 9 shows a schematic version of our implementation model on this project. Note that both the Billing program and Customer Service program `CALL a subprogram .TAXREQUEST' (which is the class name), invoking a COBOL paragraph named `.TXRQ$CALC_BILL_TAX', or `.TXRQ$CALC_ROAMER_TAX'. What's with these weird names? Here we freely borrowed the idea of name decoration (also known as name-mangling) from C++. The C++ (and C# and Java) compilers rename every method in a class to a form that is roughly:

```
<classname>D<methodname>D<parameter-list>
```

where `D` represents a compiler-specific delimiter. For example, if you have a class called `Account` that has a method called `open( float amount)`, the programmer would invoke `Account.open( float amount )`, but the compiler will produce a "decorated" function call to something like: `Account$open%1F(amount)`. This format is guaranteed to produce a unique name for any method in a class.

As shown in Figure 9, we did a straightforward application of name decoration as a `naming convention` for all subprogram entry points in our COBOL system. Unlike a compiler-enforced language feature, our convention required error-prone human compliance, so the team agreed to the convention and to perform code reviews to ensure compliance.
Now we had a convention that supported the two-way traceability we wanted, as shown in Figure 10.

<table>
<thead>
<tr>
<th>Class name &amp; Operation names</th>
<th>Subprogram name &amp; Paragraph names</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaxRequest</td>
<td>TAXREQUEST</td>
</tr>
<tr>
<td>+CALC_BILL_TAX()</td>
<td>TXRQ$CALC_BILL_TAX</td>
</tr>
<tr>
<td>+CALC_ROAMER_TAX()</td>
<td>TXRQ$CALC_ROAMER_TAX()</td>
</tr>
</tbody>
</table>

**Figure 10: Traceability between Class and COBOL implementation**

It’s a simple trick, and it works very effectively. Each procedural language has its own scoping rules, but name-decorating is required in all of these languages.

We’ve come to the end of the case study. With the level of modeling and translation convention I have described, the programmers on the team were ready to start rewriting and refactoring the existing taxation code.
into our new Taxation component. As they began working with the first iteration, our "leader" team began modeling the other Calculate Roamer Taxes scenario in our Compute Taxes use case. We moved fast, and light, and followed the "core" practices of RUP in a very agile fashion.

The implications for non-OO project teams

I promised to come back to a question I posed early in this article: Why should an organization give up all the techniques they know for new practices they don't know?

Let's address this first from the perspective of process change. Does my case study suggest that your organization should change from a waterfall process and move to the unknowns of an "iterative" process like RUP? The answer is no, you don't have to, but you probably should. Quality improvement must occur along multiple dimensions. Using UML to capture and organize your concepts and document them in a standardized fashion is only one of these dimensions. Others dimensions are:

- Business process
- Staffing and training
- Standardization for reuse
- Compensation plans
- Management accountability reform, and so forth

The major reason to move to an iterative process such as agile RUP is to achieve maximum visibility in project management -- knowing whether or not your project is on schedule and on target in meeting the system's requirements. Another big payoff is being able to see, and show your customer, actual working code every few weeks. It's a win-win proposition.

Now, let's also consider the question about moving to an unfamiliar process from a project and cultural perspective. Do the techniques I have described in the case study mean your organization never has to move to an OO language? Certainly not, but without direct object-orientation support you are, at best, using a handsaw when a power saw is what you need. However, these techniques will allow you to move incrementally toward further quality improvements in project management, code quality, documentation, internal and external communication, and more.

Do these techniques mean you have to train your whole team in RUP and UML? Yes, but to varying degrees. System analysts and those producing the models must understand UML in some depth. Quality assurance, project management, and developers need to be able to read UML artifacts, but not necessarily produce them. Everyone has to understand their role and responsibilities in the development process. You will have to either grow your own object modelers to produce the models and lead the agile process definition, or you will need to hire people with these skills.
One last question: What about using CASE tools to capture our models? I am a believer in CASE tools until they become an end in themselves. A CASE tool that supports UML modeling and is flexible enough to be tailored to an agile process approach can be a great aid. But beware of updating every model just because an operation name has changed. In fact, while I invariably use a CASE tool on my consulting projects, I do not enter the models into the tool until the models become stable. If you let the tool be your master, you will spend all your time serving the demands of the tool rather than developing working software. And proper use of a modern OO CASE tool has an additional benefit: It allows the team members to upgrade their skills with additional software development techniques, and the employer to acquire better skills within its work force.

**Final suggestions**

I have described the important aspects of using an agile object-organized approach on a pure COBOL project. I hope I have conveyed that UML and RUP are tools that can be tailored in rather unlikely ways to produce business value for the many procedural software shops still generating new applications.

Because this approach is only "object-organized," and not object-oriented, there are certainly pitfalls you must be aware of. For example, you can model inheritance or polymorphism in UML, but you will have a very difficult time implementing those characteristics of OO programming in C or COBOL. Similarly, aggregation and composition are concepts not supported directly by any procedural language, and to implement these you will have to rely on convention and team communication.

Keep in mind that you can appropriate as much, or as little, of the object paradigm as you want. Use cases themselves are not object-oriented at all. Structural models such as class diagrams establish the main business concepts and their relationships. Behavioral models, such as sequence diagrams, let you "run" the system on paper, but they require a cohesive, object-oriented mindset. Evolve your use of both approaches and train key people on an as-needed basis. You don't have to throw away anything you currently know. My bottom-line recommendation is: Use the object-oriented paradigm to supplement your structured approach. If we can do this in COBOL, we can do it for any language!

---

**Notes**

3. Some of these ideas were shared by Umar Janjua.
For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
**Book review**

*Fundamental Concepts for the Software Quality Engineer*
Edited by Taz Daughtrey

American Society for Quality -- Quality Press, 2002  
Cover price: US$50.00  
Hardcover, 288 Pages

*Fundamental Concepts for the Software Quality Engineer* is a collection of twenty excellent articles, reprinted from *Software Quality Professional* magazine and the proceedings of recent conferences. The range of topics is wide, so readers with a variety of interests will find relevant and useful articles. Overall, the book is a good way for someone with a basic knowledge of software quality engineering methods to gain a deeper understanding of various techniques and processes, and to learn practical lessons from case studies. Many of the articles will be as interesting to software project managers and test managers as they are to quality engineers.

The structure of the book reflects that of the American Society for Quality's Software Quality Engineering Body of Knowledge, a list of topics that engineers must master in order to pass the ASQ Certified Software Quality Engineer (CSQE)\(^1\) test. However, this is not intended as an exam preparation textbook. Rather, it is a selection of the best two or three recent articles on each topic, so that the book as a whole presents a broad picture of current developments in the field. The eight sections of the book are:

- Standards, Principles, and Ethics  
- Quality Management  
- Software Processes  
- Project Management  
- Measurement  
- Inspection and Testing  
- Audits
I'll briefly describe each of the chapters in each section so that readers can pick and choose those that align with their interests.

**Standards, principles, and ethics**

In this section, Watts Humphrey's article "The Software Quality Profile" describes how deploying the Personal Software Process and the Team Software Process can improve software quality and team productivity. "Choice and Application of a Software Quality Model" by Dave Miller summarizes several well-known methods of measuring software quality -- Boehm's model, Cavano and McCall's model, FURPS+, Dimensions of Quality, ISO 9126, and the SEI model -- and compares them, with an eye to helping readers choose the model that best fits their application. "Risk Management: Supporting Quality Management of Software Acquisition Projects" by Gerard Getto discusses ways to identify and manage risk when buying software and identifies key risks and how to address them.

**Quality management**

This section includes John Elliott's "Achieving Customer Satisfaction Using Evolutionary Processes," which presents the results of an experiment using the Dynamic Systems Development Method to ensure that a software project would meet customer requirements and achieve a high level of customer satisfaction. "People Management and Development Process" by Giovanni Evangelisti, Emilia Peciola, and Cosimo Zotti is a case study of how a company improved its management practices, which led to happier employees and lower turnover. The lessons apply not just to software, but to any sort of management situation.

**Software processes**

This section begins with "Risk Identification Techniques for Defect Reduction and Quality Improvement" by Jeff Tian. He presents a collection of risk identification techniques ranging from traditional -- such as correlation analysis -- to cutting-edge, such as artificial neural networks. In "Cost of Software Quality: Justifying Software Process Improvement to Managers," Dan Houston supplies ammunition for quality engineers trying to convince their managers to invest in process improvements. Applying the Cost of Quality technique developed decades ago by J. M. Juran for industrial manufacturing, Houston derives a method for calculating the financial impact of implementing software development process improvements -- or of failing to implement them. This is followed by Craig Smith's "Defect Prevention: The Road Less Traveled," a useful case study of one company's successful attempt to improve product quality by creating a culture of defect prevention.

**Project management**

In the introduction to this section, Editor Taz Daughtrey proposes that
next to the Statue of Liberty should stand a companion Statue of Responsibility, as a reminder of the balance between freedom and accountability. In the software industry, as in other fields, he maintains, there is now general agreement about what constitutes acceptable practice, and therefore also about what represents deviation from acceptable practice, or malpractice. The idea of software malpractice claims is sobering, and reminds us of how far we are from a state of perfection.

"Initial Experiences in Software Process Modeling" by Ray Madachy and Denton Tarbet presents examples of how a project manager used metrics and mathematical models to make decisions about the optimal size for his project team, whether to share people among different tasks, how much code to reuse, and so on. On the flip side, Alan Weimer and R. Jack Munyan remind us how important it is to pay attention to "people issues" on a software project in "Recipe for a Successful System: Human Elements in System Development." Different though these two articles are, both have much practical advice to offer managers, and this is one of the most valuable sections in the book.

**Measurement**

William Florac and Anita Carleton's "Using Statistical Process Control to Measure Software Processes" begins this section by applying W. E. Deming's statistical quality control methods to software development in order to ensure software stability and capability. For example, Shewart control charts that are commonly used to record and analyze metrics about the physical characteristics of manufactured objects can be used equally well to represent defect discovery rates and resolution time in a software development project. "Managing with Metrics: Theory into Practice" by Denis Meredith is a case study of a large data-scanning project that used metrics to ensure a successful result, despite such challenges as a short schedule and a ban on maintenance releases once the software was deployed. Particularly useful are specific examples of metrics and charts the project manager used. Next, "Experiences Implementing a Software Project Measurement Methodology" by Beth Layman and Sharon Rohde describes the US Department of Defense's Practical Software Measurement program, explaining how it helps identify critical issues on a project and collects metrics to see how aspects of the project affect those issues. It includes examples of how the information was used in decision making and discusses lessons learned.

**Inspection and testing**

This section begins with an interesting but somewhat inaccessible article: Tom Gilb's "Planning to Get the Most Out of Inspection." He assumes that the reader has already read his book (I had not) in discussing document inspection as a way to prevent defects in a product. In other words, by discovering defects in project documents -- requirements specifications, design documents, and so forth -- you can prevent people from implementing these problems in the software. The next article, "A Testing Maturity Model for Software Test Process Assessment and Improvement," is on my short list of favorites because it adds so much to our collective
body of practical engineering knowledge. Most software engineers are familiar with the Software Capability Maturity Model (CMM), but quality specialists recognize that this model does not address some of the processes and practices specific to testing. A complement to the CMM called the Testing Maturity Model (TMM) is proposed here by Ilene Burnstein, Ariya Homyen, Taratip Suwanassart, Gary Saxena, and Rob Grom. The TMM defines testing-related maturity goals, activities, tasks, and responsibilities that correspond to each of the CMM's five levels. Burnstein et al. also propose a TMM Assessment Model to measure the TMM level of one's own project. The final article in this section, "Choosing a Tool to Automate Software Testing" by Mark Fewster and Dorothy Graham, looks at ways to decide what software to buy. Anyone charged with acquiring a testing tool will find this a very good resource for learning what questions to ask, what to test and look for in software tools, how to predict how well a tool will work in-house, and how to work with vendors, get the most out of demos and trials, and make a final decision.

Audits

In this section, Daughtrey points out that, although many books cover the subject of auditing, most of them are not specific to software audits. "Quality Evaluation of Software Products" by Jørgen Bøegh picks up where the previous article left off: instead of evaluating a tool for purchase, Bøegh's task is to systematically evaluate the quality of a product to determine whether or not it fulfills specified quality requirements. As Bøegh points out, software is increasingly used for such life-critical systems as traffic control, robotics, and medical systems, in which software defects can have extreme consequences. He lists the relevant ISO and IEC standards for software product evaluation and testing, goes on to present several methods for objectively evaluating software quality, and concludes with a real-world example of evaluating software for a fire alarm system. The second article on auditing, "Practical Quality Assurance for Embedded Software" by Erik P. W. M. van Veenendaal, is a case study of how inspections successfully detected and prevented defects in software for a television. By supplementing his company's ISO-9001 procedures with nine person-weeks of inspection time, his team found more than 1,400 defects and was then able to ship a high-quality product. The chapter includes detailed, clear instructions on how to conduct and apply the results of such inspections.

Configuration management

This final section of the book begins with "Software Configuration Management for Project Leaders" by Tim Kasse and Patricia A. McQuaid. They take a broad view of configuration management (CM), defining it not only as source code version control but also as a methodology for controlling change in all software project artifacts -- from design specifications and data files to test procedures and user documentation. CM, they contend, is a critically important process for producing and delivering software in a controlled manner, and they provide many specific details about CM implementation. The final article in the book, "Applying Quantitative Methods to Software Maintenance" by Ed Weller, explains
how to use measurement results to make good decisions, and also holds up some commonly-held beliefs for quantitative scrutiny. On one three-year project, for example, the engineers on his team believed that if they were to push bug fixes out faster, they would introduce more new bugs in the process. But when Weller compared the recidivism rate (percent of new bugs) with time spent on fixing bugs, he found no correlation. Throughout the project he improved his group's processes by conducting similar quantitative analyses, and this chapter presents some good lessons for both project managers and process engineers.

Daughtrey ends the book with a reminder that there are triumphs as well as problems in software development projects: Everyone hears about a few well-publicized failures, but many successes go unnoticed. For those of us who spend our days looking for hidden problems in every apparently-functioning bit of software, this is a good reminder. The goal of all our efforts, after all, is to produce something that works.

On the whole, this book presented many useful ideas and techniques; rarely did I find myself muttering "No way, not in my project!" as I read. The collection of articles also accurately reflects the state of the art of software quality engineering. Note that I am not using state-of-the-art in the colloquial sense, as a descriptor for bleeding-edge innovation, but rather to refer to the true state of our current practices, warts and all. Software quality engineering is a discipline that is still in the process of maturing. And I recommend this book for anyone who wants to see a compact but detailed snapshot of what is going on in that particular corner of the software development world.

-Susan McVey
Software Quality Engineer
Rational Software
IBM Software Group

Notes
1 A copy of the CSQE Body of Knowledge is included as an Appendix to the book. Readers who are interested in software quality engineering certification can find out more at http://www.asq.org.

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

Cisco Internet Architecture Essentials Self-Study Guide: Cisco Internet Solutions Specialist

Infrastructure for e-business services (CCIE No. 8849)

by Matthew Recore, Jeremy Laurenson, and Scott Herrmann

Cisco Press, 2003
Cover price: US$50.00
Hardcover, 288 Pages

This book is a storehouse of good information for anyone concerned with setting up a network infrastructure to support Internet applications. It packs an amazing amount of material into just 288 pages. Although intended as study material for network engineers who may want to take the Cisco Solutions Architecture Fundamentals exam, it can easily serve as a useful guide for others charged with supporting Internet-aware applications and business solutions. The authors describe in detail the issues involved in setting up a sound network infrastructure and then offer a variety of feasible solutions.

Although this book does educate readers about Cisco Internet solutions technology, it also goes beyond that. Rather than simply enumerating Cisco solutions and their implementation, the authors focus first on explaining concepts and technologies related to an Internet solution architecture. This approach adds a lot of value to the book, which uses figures and sample network topologies where appropriate to explain these concepts. A section on network management also includes a brief subsection on non-Cisco technologies, such as IBM's Tivoli/Netview, HP's Openview, and Computer Associates' Unicenter TNG.

This book is divided into eight chapters, each dealing with a different aspect of an overall Internet solution architecture. Each has a consistent, textbook-like format, beginning with a list of objectives, and following with the main contents, a summary, and then review questions at the end. After introducing a network technology or concept, each chapter proceeds with a discussion that helps readers understand the technology and the business need for it, a list of possible solutions, a cost and benefit analysis if applicable, and finally, an explanation of how Cisco devices/processes
can help meet the business need.

The early chapters introduce the fundamentals of an Internet system architecture, drawing on case studies from real-world applications. The emphasis is on e-business services: customer resource management, e-commerce, supply chain management, e-learning, workforce optimization, and e-publishing.

These chapters also introduce the various components that may comprise such an architecture, including informative discussions on switches, firewalls, content engines, content routers, content caching, intrusion detection systems and so on. The authors include details about these components' implementation, configuration, placement in the network, scaling, and benefits, so this is useful review material for IT support staff. Later chapters cover various concerns related to Internet system architecture: how to achieve high availability, design issues, cost/benefit analyses, and total cost of ownership estimates.

Throughout this section, a big plus is that the authors not only discuss business concerns from a budgetary point of view, but also provide enough technical detail for a solution architect/technical staff member to make informed purchasing decisions and implementation plans. For example, they tell you exactly why using layer-three switches is preferable to using layer-two switches for planned redundancy in a network.

A chapter on network security discusses this issue in depth, including types of attacks and how to mitigate them by using various devices/policies. Like most sections in the book, the one summarizing public key encryption packs a lot of information into a small amount of space. But despite this density (and to the authors' credit) the information is always easy to follow. Although their second example regarding public-key encryption with a digital signature seemed redundant, I found their discussions of smart intrusion detection, content caching, and filtering interesting reading.

The chapter on quality of service explains what it is and how to deploy it, and describes available tools. The chapter on network management does a good job of explaining most aspects of the discipline, including performance management, fault management, and troubleshooting. There is also a detailed and easy-to-follow summary of SNMP (Simple Network Management Protocol) and descriptions of solutions that provide ease of administration, user interface via a Web browser, and automated configuration. The last chapter describes Service Level Agreements and Cisco solutions for evaluating, managing, and validating SLA objectives from different vendors.

In summary, this book delivers a thorough introduction to Internet architecture, as promised. It is excellent review material for IT support personnel and for anyone who wishes to learn how to establish and support a sound Internet framework.

-Kaveesh Mishra
Software Engineer
For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
Risk reduction with the RUP Phase Plan

by Mark Aked
Managing Consultant
Lamri Ltd.

During the Inception and Elaboration phases of a project based on IBM Rational Unified Process,® or RUP,® the project manager, ideally in collaboration with the architect, develops an artifact called the Phase Plan. A key component of the overall Software Development Plan, the Phase Plan provides a high-level, coarse-grained view of the project, showing the total number of planned iterations across the four RUP phases as well as key milestone dates for each of these iterations.

For practitioners new to RUP, a common mistake during phase planning is to decide what functionality to add during each iteration based on the sequence of the use cases. This approach overlooks the risk-driven element of RUP; it means the practitioner does not understand how to combine risk-assessment and functionality considerations in selecting content for iterations.

This article presents a disciplined approach for building the RUP Phase Plan that does consider technical risks -- those that the project team can mitigate during the Elaboration phase by implementing functionality that will manifest these risks. It also highlights ways to avoid common mistakes managers make in deriving the content of Elaboration iterations.

We will focus on the allocation of use cases for the Elaboration phase but also address some concerns related to the Construction phase. We will not cover the allocation of activities or resources to the iterations.

This article is aimed primarily at project managers and/or software architects, as these are the two roles most deeply involved in planning the project and developing the Phase Plan. It assumes some knowledge of RUP and use cases. Although we will briefly describe RUP fundamentals,
readers not familiar with the process should consult the References section below for sources.

Definitions

Let’s begin by defining terms we will use throughout the article. These are captured in Table 1.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUP or IBM Rational Unified Process</td>
<td>A comprehensive, flexible software project development framework that embodies an iterative approach and other best practices.</td>
</tr>
<tr>
<td>Use case</td>
<td>A visual description of system behavior that shows sequences of actions. A use case contains all alternate flows of events related to producing an &quot;observable result of value.&quot;</td>
</tr>
<tr>
<td>Risk</td>
<td>An ongoing or upcoming concern that has a significant probability of adversely affecting the completion of major milestones and product quality.</td>
</tr>
<tr>
<td>Issue</td>
<td>A risk that has become a reality.</td>
</tr>
<tr>
<td>Software Development Plan</td>
<td>A comprehensive, composite artifact that gathers all information required to manage the project. It encloses a number of artifacts developed during the Inception phase and is maintained throughout the project.</td>
</tr>
<tr>
<td>Risk Management Plan</td>
<td>Details how to manage the risks associated with a project. Specifies risk management tasks that will be carried out, assigns responsibilities, and identifies additional resources required for the risk management activity. For smaller projects, this plan may be embedded within the Software Development Plan.</td>
</tr>
<tr>
<td>Phase Plan</td>
<td>A high-level, coarse-grained view of the project, developed during the Inception phase. It shows the total number of planned iterations across the four RUP phases, and key milestone dates for each of these iterations.</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Iteration Plan</td>
<td>A time-sequenced set of activities and tasks that includes assigned resources and task dependencies for the iteration; a fine-grained plan.</td>
</tr>
<tr>
<td>Risk List</td>
<td>A list of known and open risks to the project, sorted in decreasing order of importance and associated with specific mitigation or contingency actions.</td>
</tr>
<tr>
<td>Use-case flows</td>
<td>Generic term for the use-case flow of events that encompasses the normal flow of events and the alternate flow of events.</td>
</tr>
<tr>
<td>Normal flow of events</td>
<td>The basic flow of events that covers what &quot;normally&quot; happens when the use case is performed.</td>
</tr>
<tr>
<td>Alternate flow of events</td>
<td>Covers optional or exceptional behavior as well as variations of the normal behavior (&quot;detours&quot; from the basic flow of events).</td>
</tr>
</tbody>
</table>

**RUP in brief**

IBM Rational Unified Process, or RUP, is a process framework focusing on software development. Two tenets of the process are that it is *iterative* and *risk-driven*.

A RUP project consists of four phases: Inception, Elaboration, Construction, and Transition. Figure 1 illustrates the structure of a RUP project, showing these phases on the horizontal axis. The phases include *iterations* that proceed through each of the disciplines shown on the vertical axis. The different phases place different emphases on each of these disciplines as the project progresses through iterations. The end of an iteration (shown horizontally at the bottom of Figure 1) delivers an executable product -- either an internal or external release.
Through years of experience with software development projects, the authors of RUP identified a tendency for teams to address the easiest or best-known aspects of the project first, in order to get the project going on a sound footing, get individuals to operate as a coherent team, and build confidence. Although these are valid objectives, this approach pushes out all the unknown, risky elements toward the end of the project, so that they can easily develop into issues very close to the delivery date. Typically, this approach forces out the delivery date, adding costly, unplanned iterations to the project.

In contrast, project teams who are risk-driven don't accept this approach. Instead, they attempt to identify potential risks and tackle them early in the project lifecycle. Once they understand or even mitigate these risks, they deal with the more comfortable aspects of the project later in the lifecycle, so the project team has a far higher probability of delivering the product on time and within budget.

**Risk-driven planning**

As we saw in Table 1, the Phase Plan, a key component of the Software Development Plan, is developed during Inception. This high-level, coarse-grained view of the project shows the total number of planned iterations across the four phases, and key milestone dates for each of these iterations. Finer-grained details are captured in an Iteration Plan for each iteration.

One reason RUP is divided into four phases is to distinguish among the different focuses of activities at different times in the project lifecycle. Table 2 shows the questions we are trying to answer and the area of focus for each RUP phase.
Table 2: The focus of RUP phases

<table>
<thead>
<tr>
<th>Phase</th>
<th>Inception</th>
<th>Elaboration</th>
<th>Construction</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Key question</strong></td>
<td>Should we build it?</td>
<td>Can we build it?</td>
<td>Are we building it?</td>
<td>Have we delivered it?</td>
</tr>
<tr>
<td><strong>Focus</strong></td>
<td>Scope</td>
<td>Risk</td>
<td>Functionality</td>
<td>Delivery</td>
</tr>
</tbody>
</table>
justification for determining the content for iterations. Although this approach may not be appropriate for every project (i.e., the project manager may be able to do this activity informally, in his or her head!), it is useful when documented justification is required, to encourage planning focused on addressing risk, or simply to provide an impetus for recovery when the planning activity has stalled.

Prerequisites

This approach requires a number of prerequisite artifacts. (In RUP, you produce these artifacts before doing the Phase Plan.)

- **Risk List.** Produced during the activity to *identify and assess risks* within the Project Management discipline; provides the current risks identified for the project.

- **Use-Case Survey.** Produced during the *find actors and use cases* activity within the requirements discipline, identifying the normal and alternate flows.

- **Development Case.** Produced during the *Develop development case* activity within the environment discipline, identifying activities to be undertaken and the particular iteration strategy for the project.

This approach also uses a particular style to specify the use cases, which is presented in **Appendix A.**

Phase Plan development

Allocating content to each of the iterations within the Elaboration phase requires the following steps:

- Prepare the Risk List and Use-Case Survey.

- For each iteration within the plan:
  - Identify candidate use cases for Elaboration.
  - Identify candidate use-case flows for Elaboration.
  - Assign use-case flows to the iteration.

- Complete the Phase Plan.

Prepare the Risk List and Use-Case Survey

The first step is to ensure that the primary inputs for the Phase Plan are of suitable quality and contain the information required.

As we have noted, the Risk List is created in the RUP activity *Identify and assess risks* (this activity is performed again during each iteration) to identify, analyze, and prioritize the risks, and assign an appropriate risk management strategy (avoidance, mitigation, or contingency) to each risk.
At this stage we are interested only in the technical risks and those risks we have decided can be mitigated through implementing use cases. We should calculate the risk exposure for each technical risk we extract from the Risk List, and then prioritize the risks. Those risks with the highest probability of occurring and that will have the most impact on project success should have highest priority.

A risk is anything that may adversely affect the outcome of the project, and they come in many forms, from technical to political. Although the project will be able to address some of these risks, others may well be outside the project's circle of influence. We might be able to manage some of the latter risks, but the project should focus on the former. However, keep in mind that just by working in an iterative manner, the team may either flush out or mitigate risks that they thought were outside the project's influence.

Figure 2 shows an example of technical risks managed within IBM Rational RequisitePro. The values used for the Probability and Impact should be documented in the project's Risk Management Plan. This example uses a scale of 1 to 5, where 1 indicates low probability and low impact, and 5 represents high probability and high impact. Multiplying the probability value by the impact value for each risk provides a Risk Exposure value we can apply for prioritizing the risks.
process, as it provides us with additional criteria for deciding what to do first.

The simplest approach is to list all of the use cases and set a *priority* value for each one. Again, you can use a scale of 1 to 5, with 1 being low priority and 5 being high priority, or essential. Conduct this activity with the system's end users or customers to ensure that you are capturing their priorities.

**Identify candidate use cases for Elaboration**

We are now ready to produce a risk-to-use-case matrix. You can do this easily with IBM Rational RequisitePro by creating a traceability matrix; set the row requirement type to Risk and the column requirement type to Use Case (see Figure 3). This matrix could also be created on paper or in a spreadsheet application, but if you have captured the textual components of use cases in IBM Rational RequisitePro, it makes sense to maintain this information in a single repository.

Using this matrix, we can examine each risk in turn by looking across the use cases and asking, "If we implement this use case, will it transform this risk into an issue"? If the answer is "Yes," then we insert an arrow (or a trace if using IBM Rational RequisitePro) at the intersection between the risk and the use case in the matrix. If the answer is "No," we leave the intersection blank.
Having carried out the risk-to-use-case mapping for all of the technical risks, we can now remove use cases without arrows (e.g., UC4) as candidates for development during Elaboration; we can safely postpone these risks until the Construction phase.

Now we can look along the row for the first risk and identify all the intersecting use cases; implementing any of those use cases will attack that risk. If we continue this exercise for each risk, it soon becomes apparent that we can attack a number of high risks by implementing just a couple of use cases -- UC1 and UC2 -- during Elaboration.

Figure 4 shows how we can use IBM Rational RequisitePro to highlight candidate use cases for Elaboration. UC1 is the prime candidate because it enables us to address four of the five highest risks. By including UC2, we can address all five of those risks.

At this point it is tempting to allocate these two use cases to the first iteration in Elaboration, communicate to the team that they are to be developed, and start detailing the use-case specification for every flow.

However, remember that earlier we said the goal of Elaboration is to implement the smallest amount of functionality that will confirm whether the high-priority risks we identified either are -- or might become -- issues. Implementing the whole use case contradicts this principle, because many of the use-case flows will not address these risks.
To give our plan discipline and to avoid this situation, we can take the use case down to the next level of detail and carry out exactly the same process we used for identifying the use cases for Elaboration, but this time on the use-case flow of events.

**Identify candidate use-case flows for Elaboration**

Using the use cases we have identified as candidates for development during Elaboration, we can now produce a second matrix, this time focusing upon the use-case flow of events (or *use-case flows*, as we'll now refer to them).

If we are using IBM Rational RequisitePro, we can use the parent-child relationship feature to expand the use cases, reveal the use-case flows, and produce a risk-to-use-case flow traceability matrix (Figure 5). Then, using this matrix, we attend to each risk in turn and, looking across the use-case flows, we ask the question: "If we implement this use-case flow, will it transform this risk into an issue"? If the answer is "yes," then we insert an arrow at the intersection between the risk and the use-case flow in the matrix. If the answer is "no," then we leave the intersection blank. If, as in Figure 5, we are able to identify the mitigation of each risk with an actual step within a use-case flow, then we should place the arrow at this intersection; this may help us in selecting which use-case flows to address during Elaboration.

![Figure 5: Candidate use-case flows for Elaboration](Click to enlarge)

In the example shown in Figure 5, we now have a number of choices as to
the use-case flows from UC1 we may wish to implement in the first Elaboration iteration in order to address four high-priority risks.

The first step in the normal flow of events (UC1.1) addresses one of the four risks (RISK6). The second step in the normal flow of events (UC1.2) addresses four of the highest priority risks (RISK1, RISK3, RISK4, and RISK6). However, we should also consider one of the alternate flows (UC1.2.2), because it addresses the same four risks. Both the normal flow (UC1.5) and alternate flow (UC1.5.1), at the fifth step in the use-case, address RISK6, so again we have a choice.²

Table 4 summarizes the candidate use-case flows for Elaboration.

<table>
<thead>
<tr>
<th>Candidate use-case flow 1</th>
<th>Candidate use-case flow 2</th>
<th>Candidate use-case flow 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>UC1.1</td>
<td>UC1.1</td>
<td>UC1.1</td>
</tr>
<tr>
<td>UC1.2</td>
<td>UC1.2.2</td>
<td>UC1.2</td>
</tr>
<tr>
<td>UC1.3</td>
<td></td>
<td>UC1.3</td>
</tr>
<tr>
<td>UC1.4</td>
<td></td>
<td>UC1.4</td>
</tr>
<tr>
<td>UC1.5</td>
<td></td>
<td>UC1.5.1</td>
</tr>
</tbody>
</table>

Assign use-case flows to the iteration

The particular use-case flow we choose to allocate to the iteration depends upon any additional criteria that may pertain to our project. For example, early on in the project we may wish to increase the confidence of some stakeholders by showing them the normal flow of events, or determine the strategy by which we will handle exceptions. This may help us select which candidate use-case flow to implement during the first iteration of Elaboration.

Once we have selected a use-case flow, the final step of the process is to capture that decision by updating the Risks being mitigated and Use-case flows to implement sections in the Phase Plan for the first iteration, as shown in Table 5.

| Table 5: Phase Plan with first Elaboration iteration content allocated |
Complete the Phase Plan

Having produced the content for the first iteration of Elaboration, we will now follow the same process (it is iterative!) for subsequent iterations in the remaining project phases. If we have addressed all of the high-priority risks, the planning can concentrate on the content for the Construction phase, in which the focus is to deliver functionality. We can consider the priorities we assigned earlier to the use cases when assigning use-case flows to initial Construction iterations. If any high-priority risks remain (or new risks have been discovered), then we can address them during the last Elaboration iteration.

Table 6 shows the completed Phase Plan for our example project.

Table 6: Completed Phase Plan
<table>
<thead>
<tr>
<th>Phase</th>
<th>Elaboration</th>
<th>Construction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Iteration number</strong></td>
<td>Iteration 1</td>
<td>Iteration 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Iteration 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Iteration 4</td>
</tr>
<tr>
<td><strong>Objective</strong></td>
<td>Risk</td>
<td>Risk</td>
</tr>
<tr>
<td></td>
<td>Functionality</td>
<td>Functionality</td>
</tr>
<tr>
<td><strong>Risks being mitigated</strong></td>
<td>RISK1 RISK3 RISK4 RISK6</td>
<td>RISK5</td>
</tr>
<tr>
<td><strong>Use-case flows to implement</strong></td>
<td>UC1.1 UC1.2 UC1.3 UC1.4 UC1.5</td>
<td>UC2.1 UC2.2 UC2.3</td>
</tr>
<tr>
<td><strong>Milestone date</strong></td>
<td>To be completed</td>
<td>To be completed</td>
</tr>
</tbody>
</table>

Figure 6 demonstrates how we can represent this Phase Plan in IBM Rational RequisitePro by defining an *assigned to iteration* attribute against the use-case requirement type.³
Common mistakes

Practitioners adopting RUP for the first time who use this risk-driven approach to planning are often concerned about the amount of content in the Elaboration iterations. They find it disconcerting that only one or two use-case flows are addressed early in the project and fear that progress will be too slow.

Attempting iterative development for the first time does require a different approach: Contrary to intuition, constraining the content of these iterations means that we can rapidly move through all of the disciplines and deliver the iteration in a short period of time. This approach is essential for helping the development team to:

- Adapt to and accept this new iterative style of developing software.
- Gain buy-in to the new approach through a successful, early delivery of a tangible product.
- Obtain essential feedback for the process engineer that enables him or her to assess the Development Case and add or remove activities for the next iteration.
- Create a natural "heartbeat" for the project, so that a working product is regularly delivered, and assessment of the product and process (i.e., progress) become commonplace.
It is important to educate stakeholders about the objectives of the phases, especially Elaboration, in which the aim is to attack risk and not necessarily to provide functionality. Progress during Elaboration should therefore be measured by the number of risks addressed rather than by the number of use-case flows implemented. Although the latter metric is essential, it is used to measure progress during Construction, when the focus is on implementing functionality.

If, after producing the content for an Elaboration iteration, the remaining development is insufficient to keep everyone busy, we recommend that you introduce additional use-case flows addressing the same risks for that particular Iteration. This "belt and braces" approach absolutely ensures that a risk has been addressed by testing it in more than one place.

Keep in mind that the example presented in this article is a simplification of a real-world scenario; in some cases, implementing a particular use-case flow to address many risks may be more expensive at this early stage than selecting a number of flows from separate use cases to address particular risks. If this happens, and if you can implement the separate use-case flows within the time allocated for the Elaboration iteration, then go with it. You should consider these additional factors as you plan the iteration content, but keep the focus on addressing particular risks.

If you find that you must (e.g., for political reasons) add use-case flows to an Elaboration iteration that do not address high-priority risks, then simply increase the scope of the risks you address and use this as the vehicle for introducing additional use-case flows. Don't be tempted to just add another use-case flow to provide additional functionality. During Elaboration, it is important to maintain the project team's focus on addressing risks.

**Next steps**

After completing all the steps we have described, the next steps are to allocate resources to the iterations, draw up a schedule, and then fill in the milestone date section of the Phase Plan.

**Summary**

This article outlines an approach for identifying and planning the content of iterations during the Elaboration phase of a RUP project. To ensure that your planning is risk-driven during this crucial phase, observe the following practices.

- Keep in mind that the objective of Elaboration iterations is to address and attempt to drive out technical risks.

- Use the technical risks to identify the use cases that will address these risks, and then identify the flows for these use cases that address the technical risks.
● Don't waste effort implementing the whole use case: Implement as little as possible to mitigate the risk, and then move on.

● Don't be too ambitious: Keep the content of an iteration at a manageable size.

● Maintain the team's focus on completing the current iteration's goal.

● Use requirements management tools such as IBM Rational RequisitePro to capture planning decisions (e.g., "We are implementing this use case to address these particular risks") and to capture the allocation of use cases to particular iterations.

Resources

Use Cases

Various books are devoted to the topic of writing use cases. Two good ones are:

● *Writing Effective Use Cases* by Alistair Cockburn (Addison-Wesley, 2000).

● *Use Case Modeling* by Kurt Bittner and Ian Spence (Addison-Wesley, 2002).

RUP

For more information on IBM Rational Unified Process, see


You'll find additional information on RUP in IBM Rational whitepapers (www.rational.com/products/whitepapers/index_all.jsp) and archived Webinars (www.rational.com/events/webinars/index.jsp) as well as the archives of The Rational Edge (www.therationaledge.com).

Notes

¹We refer to this as the "while I'm in there" syndrome, as it is easy to think, "While I have
the code on the screen, I might as well do a little bit extra. Team members should avoid this
tendency, as it not only distracts them from the goal of the iteration, but also it can introduce
errors that prolong the iteration and the Elaboration phase.

Note that we will always implement a flow in its entirety. A fragment of a flow is not
testable in isolation, so by implementing it we cannot objectively assure that we have indeed
mitigated the target risks.

The assigned to iteration attribute is a multiple value list type with the values of 1, 2, 3, and
so forth, corresponding to the iteration number. A multiple list is used to enable the iteration
numbers to be represented at the use case level.

Appendix A: An effective use-case specification style

You can use various styles for documenting a use case, and we are not
here to recommend a particular style. However, to be able to allocate a
certain use-case flow of events to a risk, you should use a style that
enables you to identify each of these flows.

The style used in this article follows a simple outline numbering scheme
that exploits the parent-child relationship within IBM Rational RequisitePro
by allocating a unique identifier to each relevant component of the use-
case specification -- for example, the use-case name, normal (or basic)
flow of events, and alternate flow of events.

As Figure A-1 shows, at the top level is the use-case name, which is X.
Each step in the basic flow is then named as a subset of X -- X.1 is step
1; X.2 is step 2, and so forth. Each alternate flow is then numbered at the
next level as a child of the basic flow step from which it branches.

X <Use Case name>
X.1 <1st step of basic flow of events>
X.2 <2nd step of the basic flow of events>
...
X.n <nth step of the basic flow of events>

X.1.1 <1st alternate flow from 1st step of the basic flow of events>
X.1.2 <2nd alternate flow from 1st step of the basic flow of events>
X.n.1 <1st alternate flow from nth step of the basic flow of events>

Figure A-1: Numbering scheme for use-case flows

Figure A-2 shows how the numbering scheme works for a use case called
Borrow Item.
Figure A-2: Example of a numbering scheme for a specific use case

An advantage of this style is that the numbering allows the reader to immediately identify the particular component of the use-case specification. For example, 1.2.1 is an alternate flow from step 1.2 of the normal flow in use case 1.

For more information on the products or services discussed in this article, please click [here](#) and follow the instructions provided. Thank you!
Principles and techniques for analyzing and improving IBM Rational ClearCase performance

Part II: Improving performance

by Tom Milligan
Software Configuration Management (SCM) Specialist
Rational Software
IBM Software Group

with Jack Wilber
IBM Rational Consultant

Part I of this series, published in the July issue of The Rational Edge, provided an overview of the principles of performance assessment in an IBM Rational® ClearCase® environment. I outlined an approach that I have found useful in diagnosing performance issues. This approach is based on proceeding methodically through the performance stack -- starting at the operating system/hardware level, then checking and tuning Rational ClearCase parameters, and finally examining and optimizing the application layer. Now, in Part II, I will discuss how to use specific tools and practices at each layer of the performance stack to assess and improve the performance of IBM Rational ClearCase on both Windows and UNIX platforms.

The OS/Hardware level of the performance stack

As I discussed in Part I, I tackle performance issues by starting at the bottom of the performance stack -- at the OS/Hardware level (see Figure 1).
At this level I check for:

- Memory shortfalls
- Disk input and output (I/O) problems
- Network bandwidth and latency issues.

I'll discuss these issues in detail below.

## Checking for memory shortfalls

IBM Rational ClearCase uses a number of caches to improve performance. Memory is used to cache data on view servers and VOB servers. In addition, there are MVFS caches, and a cache for each view in a view server process. All of these caches rely on having enough memory to work effectively, and that is why memory is so critical to Rational ClearCase performance.

Memory is also related to processor performance and disk input/output. A system that is short on memory may also have high CPU utilization, because the processor must frequently swap memory pages to disk.

On UNIX systems, I use UNIX utilities such as `vmstat`, `sar`, and `glance` to check memory utilization and processor loading. However, not all of these utilities are found on all versions of UNIX. Linux and Solaris implementations are likely to have `vmstat`, and HP-UX will have `glance`.

Figure 2 shows `vmstat` output from a system with a severe memory shortage; each row in the `vmstat` output represents a snapshot of the system at a specified time interval.
Figure 2: Output from the `vmstat` utility shows virtual memory statistics

The scan rate is a key indicator of a memory shortfall. When the system has determined that it needs space in memory, it scans for pages of memory that it can swap to disk. The scan rate reflects how many pages of memory per second it is looking at. Before the introduction of the Solaris 8 operating system, a sustained scan rate of more than 200 pages per second indicated a memory shortfall situation. For Solaris 8 and beyond, any non-zero value in the scan rate column means the system is short of memory.

The run queue column is also important because it shows how many processes are in the run queue awaiting service from the processor. The run queue should not be more than the number of CPUs in the system minus 1, but I look for any non-zero value in the run queue.

The metrics in Figure 2 indicate a memory shortage that seems to point to a need for more processor power. The CPU idle time, shown in the last column, is zero. The run queue is also stacked up, with 15 processes waiting in the final row. If you were to look only at these two columns, you might think that more CPU power would solve the problem; but in fact the processor is doing so much memory scanning that it is overwhelmed with system overhead tasks, and has little time to actually run other processes.

Note that the paging rate values are not always as important as the scan rate. In some operating systems, input/output is mapped to memory pages, so a high paging rate may simply reflect heavy I/O activity. This is especially true on machines that are NFS servers.

If `vmstat` is not available on your system, you can get much of the same...
information from `sar`, although it is not always as simple to read. Figure 3a shows `sar` output on a system that has enough memory, and Figure 3b shows `sar` output on a system that is low on memory.

```
[root@linux root]# sar -B 5
Linux 2.4.7-10 (linux) 04/17/2003
11:20:54 AM 0000/gpgin/s 0000/gpgout/s 0000/activepg 0000/inactypg 0000/inacnpq 0000/inactarpg
11:20:59 AM 0000/gpgin/s 0000/gpgout/s 0000/activepg 0000/inactypg 0000/inacnpq 0000/inactarpg
11:21:04 AM 0000/gpgin/s 0000/gpgout/s 0000/activepg 0000/inactypg 0000/inacnpq 0000/inactarpg
11:21:09 AM 0000/gpgin/s 0000/gpgout/s 0000/activepg 0000/inactypg 0000/inacnpq 0000/inactarpg
```

Figure 3a: Output from the `sar` utility for a normal system

```
[root@linux root]# sar -r 5
Linux 2.4.7-10 (linux) 04/17/2003
11:23:38 AM 0000/kmemfree 0000/kmemused 0000/kmemused 0000/kmemused 0000/kmemused 0000/kmemused 0000/kmemused
11:23:43 AM 0000/kmemfree 0000/kmemused 0000/kmemused 0000/kmemused 0000/kmemused 0000/kmemused 0000/kmemused
11:23:48 AM 0000/kmemfree 0000/kmemused 0000/kmemused 0000/kmemused 0000/kmemused 0000/kmemused 0000/kmemused
```

Figure 3b: Output from the `sar` utility for a system with memory shortfall

You can see the percentage of memory used is only 77 percent in the normal system but 94 percent in the troubled system; the normal paging rate is zero, but the troubled system's rate is hundreds of pages per second. For the latter system -- and for the system shown in Figure 2 -- adding more memory is a good first step toward improving performance.

Alternatively, you can view other processes running on the system using `ps` or `top`, to see whether some memory-intensive processes can be offloaded to a different system.

On Windows platforms, the Windows Task Manager (`taskmgr.exe`) is an easy way to check memory and CPU usage. Performance Monitor (`perfmon.exe`) is another good tool for analyzing Windows systems. In addition, you may find open source or third-party Windows versions of `vmstat`, `sar`, and other UNIX utilities for checking system resources.

### Checking for disk I/O problems

Just as the `vmstat` utility shows virtual memory statistics, the UNIX
The `iostat` utility shows I/O statistics. Figure 4 shows sample `iostat` output, which you can use to identify disk bottlenecks and load balancing issues.

```
Disk Device Activity || iostat -xn 1 10
```

```
extended device statistics
     r/s  w/s  kr/s  kw/s  wait  activ  wsyc_t  asyc_t  %w  %b  device
 0.0  0.0  0.0   0.0    0.0     0.0      0.0     0.0   0.0   0.0   fd0
...
  2.0  9.2  8.6  54.0   0.0    0.4    0.0    0.0   31.9  0  13  c0t0d0
  2.0  9.2  8.6  54.0   0.0    0.3    0.0    0.0   30.2  0  13  c0t1d0
  0.3  0.1  3.2  0.9   0.0    0.0    0.0    0.0   23.5  0  0  c0t2d0
  0.3  0.1  3.2  0.6   0.0    0.0    0.0    0.0   10.6  0  0  c0t3d0
  0.1  0.0  5.8  0.1   0.0    0.0    0.0    0.0    8.7  0  0  c1t1d0
  0.0  0.0  0.0  0.0   0.0    0.0    0.0    0.0    0.0  0  0  c1t2d0
  0.0  0.0  0.0  0.0   0.0    0.0    0.0    0.0    0.0  0  0  c1t3d0
  8.0  6.0 29.0 53.1   0.0    0.2    0.0    0.0   24.5  0  9  c1t4d0
  0.0  0.0  0.0  0.0   0.0    0.0    0.0    0.0    0.0  0  0  c1t5d0
  0.0  0.0  0.1  0.0   0.0    0.0    0.0    0.0    8.2  0  0  c1t6d0
...
```

**Figure 4: The `iostat` utility displays disk activity**

In `iostat` output, I look first at the `%w` column, which indicates the percentage of time that there are transactions waiting for service. Put another way, this is the percentage of time that the device queue is not empty. The number here should be zero. The remedy for a non-zero value in the `%w` column is to redistribute the disks so that busy disks are spread across multiple disk controllers. Alternatively, you can redistribute the data so that often-used data is spread across disks on different disk controllers, or install a faster disk controller.

I also look at the `%b` column, which shows the percent of time that a device is busy. This column can help identify load balancing issues. For example, you might notice that almost all of your devices have low `%b` values -- say less than 5% -- and that one disk is 30% busy. In this case, you have a disk that is really in demand, and you might want to spread the data across other disks.

For Windows environments, I wrote a small Visual Basic application (see Figure 5 and Appendix A) that helps identify both disk bottlenecks and memory shortfall situations.\(^1\)
Checking for network bandwidth and latency issues

If I do not detect any resource shortages after checking the memory, disk, and processor utilization on the IBM Rational ClearCase VOB host, view host, and relevant Rational ClearCase client machines, my next step is to look for network latency problems. From the client machine, I use the UNIX and Windows ping utility to measure the roundtrip time to the VOB and view hosts. As a rule of thumb, if I see roundtrip times of greater than 10 milliseconds, then network latency may be affecting IBM Rational ClearCase performance. Lost packets are also a cause for concern, because they have to be retransmitted, effectively doubling transmission time.

If ping indicates there may be a problem, I use the traceroute utility (on Windows, tracert.exe) to dig a little deeper by tracing the route a packet takes as it travels to a server. Ideally, I like to see just one hop (or even zero, but that's not likely in most networks) to the server. Each hop represents a delay, as the packet is routed through network hardware -- so the fewer hops the better. Output from traceroute can also help pinpoint exactly which hop is causing the most delay. Network administrators can use this information to help improve network performance, and, ultimately, IBM Rational ClearCase response times.

Two other tools that can be helpful in identifying network problems are strace and truss.² They are particularly useful in tracking down repeatable, command-centric problems, because they identify each system
call that is executed by a program. For example, if I notice that a snapshot view update is taking a long time, I can run `strace` during one of the updates. In analyzing the output, I look for lengthy pauses during any system calls. If I see a long delay on a call related to a network operation - for example, resolving a hostname or opening a file on a network server - I can use that information to isolate and resolve the cause of the slowdown.

If I suspect the network is contributing to poor performance but cannot determine what the problem is, I use a "packet sniffer" such as `snoop` or `ethereal`. Typically, I use these only as a last resort, to see exactly what is going on at the packet level. It is important to remember sensitive data often flows over a network in an unencrypted form and sniffer software makes this data visible to the user. For this reason many companies take a dim view of unauthorized sniffing of their internal networks, so it is advisable to get network administration staff directly involved if this kind of data collection is required.

Sometimes I find that it is instructive to just play around with the tools. For example, not long ago I was wondering what would happen if my registry server went down. I configured my network, fired up `ethereal`, and unplugged the network cable to the registry server. Then I tried to use `cleartool`, the IBM Rational ClearCase command line utility, to look at a VOB. I saw a packet go out, asking where the registry server was, and it hung there waiting for a response. At the command line, `cleartool` was also blocked and waiting for a response. As soon as I plugged the cable back in, the packets started flying again, and `cleartool` continued.

When I am tracking down performance problems, if a particular IBM Rational ClearCase command appears to be blocked, I can use `ethereal` to see what kind of packet activity occurs when the command finally wakes up. That often helps me identify what network resource is either not available or is slow in responding.

**The IBM Rational ClearCase level of the performance stack**

Once I resolve any problems I find at the bottom of the performance stack, I move up and take a look at the Rational ClearCase tunable parameters. At this level I look at three principal areas to improve performance, all of which involve dynamic views:

- **MVFS caches.** In IBM Rational ClearCase, MVFS (multiversion files system) supports dynamic views. Dynamic views use the MVFS to present a selected combination of local and remote files as if they were stored in the native file system. Essentially, MVFS enables users to view a VOB as a collection of files and folders. MVFS maintains several caches to maximize performance.

- **Individual view caches.** In addition to MVFS caches, each individual view has its own cache. Dynamic views require frequent
checks for updates, making them fairly "chatty" in terms of the number of RPCs (remote procedure calls) they use. The view server maintains several caches, consisting mostly of VOB data, to respond faster to RPCs from clients.

- **Express builds.** In my experience, relatively few people are aware of express builds. Technically, these are not really tunable parameters, but they can be used to significantly improve build performance when using dynamic views.

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

**MVFS caches**

I use two IBM Rational ClearCase tools to assess MVFS cache performance: cleartool and mfsstat. Using cleartool getcache -mvfs, I can see how full the various MFVS caches are.

As Figure 6 shows, cleartool getcache -mvfs also offers recommendations for adjusting MVFS cache parameters. The IBM Rational ClearCase Administrator’s Guide provides more information on what each of the caches does.

```
> cleartool getcache -mvfs
Mmode: (active/max) 155/8192 (19.3068)
Mmode freelist: 1652/1000 (32.9449)
Cfile freelist: 69/1000 (6.9303)

DNC: Files: 1597/1600 (99.0512)
      Directories: 390/400 (97.5000)
      EROOT: 1000/1000 (100.0000)

RPC handles: 10/10 (100.0000)
```

These recommendations assume that you have performed typical ClearCase activities recently on this machine. To adjust view cache size, use "cleartool setcache -view". To adjust NFTS parameters temporarily, use "cleartool setcache -nfts" with the options given below to adjust them permanently, see the documentation for administering ClearCase.

Mmode freelist may be too small—increase mfs_vobfreemax with -vobfree <cnt>

Simultaneous number of RPC handles may be too low—

increase mfs_client_cache_size with -rpchandles <cnt>

DNC directory cache may be too small—increase mfs_dncdirmax with -dirdnc <cnt>

DNC regular file cache may be too small—increase mfs_regdirmax with -regdir <cnt>

```
Attribute cache miss summary (for tuning suggestions, see the
documentation for administering ClearCase):
Attribute cache total misses: 7710000 (100.0000)
Close-to-open (view open) misses: 3575956 (46.1446)
Cache timeout misses: 21250 (0.9246)
Cache fill (new file) misses: 970087 (12.5084)
Event time (vob/view mod) misses: 3138149 (40.7747)
```

**Figure 6: Sample output from cleartool getcache -mvfs**

Having a mostly full cache is not necessarily a problem. In fact, it often reflects a smoothly running system. However, if you have a full cache with a low hit rate, then that cache is likely undersized. The hit rate indicates how often IBM Rational ClearCase finds the information it is looking for in the cache.

To determine hit rates for the MVFS caches, I use mfsstat -cl, as in
Figure 7.

```
/usr/atria/ste/mvfsstat -cl
Directory Name Cache: 596590551 calls:
  556427560 (99.2%) hit:
  0950681 current directory
  24463244/280546055 directories: (90.64)
  4102600/157590667 regular files: (80.79)
  4053000/44002766 name not found (90.39)
  908420 (0.78) miss
  1760553 event misses
  42200339 (7.14) add:
  3993291 directories
  23906084 regular files
  4350564 name not found
Attribute cache: 714000676 calls:
  710000176 (99.44) hit: 3294086 lvm-generated
  7700000 (1.14) miss:
  2540254 close-to-open
  0 build generation mismatch
  73498 timed out
  970007 new
  310079 vob/view event;
  197550 lvm also missed
67164088 updates:
  3973457 unexpected modifications
  1940544 requested modifications
  34415 VOB/view cache modifications
Readir cache:
  52 calls:
    0 (0.0%) hit
    51 (100.0%) miss
```

Figure 7: Sample output from the IBM Rational ClearCase utility mvfsstat

I look for the pattern of full caches and low hit rates to identify a cache that might benefit from an increased size. In general, I like to see a hit rate above 90 percent. If I notice a hit rate below 80 percent, and the cache is approaching full, then I increase the cache size. One more note: the *enonct name not found* cache is an important one, because a cache miss there is particularly time consuming. If you find that the cache is full with a low hit rate, increasing its size should improve performance substantially. On multi-processor machines, you also need to be careful not to make the cache too large. Oversized caches can actually cause some performance degradation on systems that have multiple processors, because of the way IBM Rational ClearCase processes manage locks on the cache memory and the extra time it takes to search through a larger cache.

You can also measure the effectiveness of MVFS caches with respect to a single command using `mvfstime -cl <command>`, where `<command>` could be `clearmake` or another IBM Rational ClearCase tool. This can be useful to assess how the caches are performing during a specific activity such as a build. After the command completes, `mvfstime` will display the cache hit rates during the execution of that command.

Adjusting the size of MVFS caches is fairly straightforward. On Windows, use the IBM Rational ClearCase applet in the control panel (see Figure 8). A scaling factor is used to bump up the size of all the caches, which is the preferred approach in most cases. Alternatively, you can set the sizes of the individual caches by clicking the "override" check box. However, changing individual cache sizes may negatively impact performance. Normally, I just gradually increase the scaling factor, and then check
cache performance again, repeating this until the cache hit rate rises to an acceptable level.

On UNIX, I use `cleartool setcache -mvfs -persistent -scalefactor` to adjust cache sizes. The `-persistent` option maintains the change across MVFS restarts. Without it, the cache sizes would return to their original values the next time MVFS started. It is beyond the scope of this article to go into full detail, but the section on **Examining and Adjusting MVFS Cache Size** in the IBM Rational ClearCase Administrator's Guide provides excellent guidance.

![Figure 8: Adjusting MVFS cache sizes with the IBM Rational ClearCase applet](image)

**Individual view caches**

The steps needed to analyze and adjust individual view caches are similar to those for MVFS caches. To check cache effectiveness, use `cleartool getcache -view viewname` on Windows and UNIX systems.

As with the MVFS caches, I look for full caches with low hit rates. The example in Figure 9 shows a lookup cache is only six percent full, with a hit rate of 59 percent. If the lookup cache was close to 100 percent, I would make an adjustment, but in this case there is nothing to worry about. It is also important to check cache statistics more than once to ensure you are seeing valid results, and not transient values that may be the result of a recently restarted server.
Figure 9: Checking individual view cache effectiveness

Figure 10 shows how to adjust individual view cache sizes using cleartool setcache -view. In the example, the -cview option specifies that I am setting the current view. The -cachessize 1M option specifies the total amount of memory to use for all caches. The default values for total cache size are 512 KB on 32-bit platforms and 1 MB on 64-bit platforms. Adjusting the total cache size does not change the ratio allotted to the subcaches; if I increase the total cache size, each subcache gets proportionately bigger. I can also set a site-wide default on cache sizes, using the cleartool setsite command.

```
cleartool setcache -view -cachessize 1M -cview
The new view server cache limits are:
  Lookup cache:  84816 bytes
  Readir cache:  419430 bytes
  Fstat cache:    247380 bytes
  Object cache:  296856 bytes
  Total cache size limit:  1048576 bytes
```

Figure 10: Setting individual view cache sizes

Express builds

You may also want to consider using express builds to improve build performance. If you are unfamiliar with express builds, here is a brief overview.

During a normal IBM Rational ClearCase audited build in a dynamic view, the fact that a derived object (for example foo.o) has been built, along with other information such as the version of the source files and the commands used to generate the derived object, is registered with the VOB. This is done to facilitate the IBM Rational ClearCase build avoidance (or binary sharing) mechanism. Consider a project in which multiple people are working together, and some are building the same software over and over. When one person builds foo.o using clearmake in a dynamic view, clearmake "goes shopping." That is, it asks the VOB if anyone else built foo.o using the same version of foo.c and all other source files, using the same command line. If someone did, then there is no need to rebuild foo.o, so clearmake simply shares the original foo.o, which it moves into a special pool within the VOB called the derived object pool. When anyone builds using the same parameters, they will use the
from the derived object pool. This operation is known as "winkin", and it typically happens faster than a straight compilation of foo.c.

What that means from a performance perspective is that every time I use clearmake or omake to do a build in a dynamic view, these tools tell the VOB what I am building. This generates a lot of RPCs and write transactions to the VOB, which slows performance.

During an express build, however, these tools do not tell the VOB about anything that is built. I can still access the VOB's derived object pool and share existing objects, but I don't tell the VOB what I am doing -- so no one else can share the results of my build.

Now, this may seem selfish, but it speeds up the build process in two ways. First, during an express build, derived objects are "registered" with the VOB. This means the VOB database is not blocked for write access; so other users can get quicker access to the VOB during my builds, and the VOB host can support more users on existing hardware. Second, my builds are faster because those same time-consuming VOB writes are eliminated. I have found that individual express builds are up to 20 percent faster than normal builds, depending on the project.

In Figure 11, you can see that response time increases along with the number of background users doing builds in non-express build views. For express builds, the line has a much shallower slope. You can support many more users with the same hardware because express builds reduce the amount of traffic going to the VOB and the amount of work the VOB server has to do.

Figure 11: Express builds keep response times down as the number of users increases
Adopting express builds as a team practice usually requires a change in the team’s user interaction model. If I use express builds, then I have to explicitly share my build results or build avoidance will not work well for the rest of the team. I can do that manually using the winkin command. Also, I can use cleartool ls -long to determine if an object is shared or not. For example,

```
cleartool ls -long foo
  derived object (non-shareable) foo@@05-
  Mar.10:31.2147483694
```

indicates that the object foo is not being shared with the rest of the team. It is important to remember that a shareable derived object cannot have non-shareable parts. If I build an entire application, for example, I can’t just share the application; I have to share all of the objects that compose it.

Typically, when a team adopts this model, they will set a site-wide default to create views that use express builds. They will also set up one normal view that is a not an express build view. Instead of relying on all team members to explicitly share their derived objects, they use this normal view to do a nightly build -- or even an hourly build if desired. The purpose is to populate the VOB with derived objects that everyone on the team can winkin. That way, everyone can still enjoy the advantages of binary sharing, but without slowing down to tell the VOB every time they build something.

To create a new view that uses express builds, simply use the -nshareable_dos option in cleartool when making the view. For example, you can use this command: cleartool mkview -nshareable_dos <rest of command line>. To convert an existing view to express builds, use the same option with chview. For example, use cleartool chview -nshareable_dos <rest of command line>. After that, just use clearmake or omake to build as usual.

In a Unified Change Management, or UCM, development stream, express build views are created by default. In non-UCM projects they are not. However, you can use cleartool setsite to set a site-wide default for creating views that use express builds. For more information on express builds, see Building Software with Rational ClearCase.

Whether you decide to use express builds or not, if your builds take advantage of IBM Rational ClearCase's build avoidance and binary sharing capabilities, then Rational ClearCase administrators can help improve build performance by ensuring that obsolete views are deleted when they are no longer needed. The process of "shopping" for appropriate derived objects to winkin can affect performance when Rational ClearCase must examine several potential configuration records for a possible match. This suggests that sites that depend on winkin need to keep the VOB cleansed of old derived objects. This, in turn, implies removing views that are no longer being used. The scrubber utility, whose job it is to remove unused
derived objects, will not remove a derived object so long as a single view
still holds a reference to it. Over time, the number of references
diminishes until only the view that created it holds a reference, which is
sufficient for the derived object to linger in the VOB while the view
persists. By deleting unused views, the Administrator enables the scrubber
to remove old derived objects from the pool, which accelerates the
process of "shopping" during dynamic-view builds.

**The application level of the performance stack**

The application level is the uppermost layer of the performance stack. It
consists of process scripts and triggers that perform activities related to
IBM Rational ClearCase functions. It also consists of clearmake -- for
building applications -- and the makefiles that clearmake depends on. As I
mentioned in Part I of this series, the application level is sometimes
difficult to address, as it is often more complex, and frequently defended
by the owners or creators of the scripts, triggers, or makefiles that you
may want to change. However, the potential for performance gains in this
area is great.

**Process scripts and triggers**

Many organizations use process scripts written in an interpreted language
such as Perl or a shell scripting language to perform special processing.
For example, instead of using a simple cleartool checkout, they will
create a script that first logs the checkout, and then issues one or more
cleartool commands. Other organizations create scripts to resemble
legacy version management systems. In either case, the scripts tend to
issue multiple cleartool commands, one after another.

Triggers, which are configured to run automatically during a particular IBM
Rational ClearCase operation, can also invoke cleartool from an
interpreted language. Unlike process scripts, triggers are often invisible to
users, who might not know that any additional processing is happening as
part of their check-out operations, for example.

If you are using Perl to implement process scripts or triggers, there are a
few steps that you can take to improve performance. In fact, if you are
not using Perl, you may want to consider re-implementing your application
level scripts in Perl to take advantage of some of these techniques.

As I mentioned in an earlier Rational Edge article, "Using Perl with Rational
ClearCase Automation Library (CAL)," when a program invokes
cleartool, the operating system must create a new process, and there is
a significant amount of overhead associated with starting that new process
-- allocating memory for it, creating a new entry in the process table, and
so on. This can be very time consuming, and if a script calls cleartool
repeatedly, the delays accumulate and can become very noticeable. You
can avoid the delays associated with multiple cleartool commands by
using the Perl ClearCase package or the ClearCase Automation Library,
which provide entry points directly into IBM Rational ClearCase. In my previous article on CAL, I noted a simple experiment that I conducted to gather basic performance measurements. I timed a task that I completed with cleartool and then timed the same task using a CAL implementation instead. For this specific operation, the CAL implementation was about 30 percent faster.

Another technique that can help improve performance is to precompile your Perl scripts into self-contained applications, using PerlApp from ActiveState's Perl Development Kit, for example. In Part I of this series, the organization highlighted in the case study took another approach. When they re-evaluated the functionality they had implemented using scripts and triggers, they noticed that Unified Change Management could handle much of it. So they decided to adopt UCM. As a byproduct of that change, much of their application-level processing was moved down into native Rational ClearCase operations, which are much faster.

Before UCM was available, a number of organizations had written wrappers around check-out and check-in operations to build up change sets associated with activities. The wrapper scripts would prompt users to specify what activity they were working on and then track the activity, using a database or some other mechanism. Now, all of those operations are native UCM Rational ClearCase operations. By using UCM to perform them instead of homegrown scripts, you can achieve significant performance gains.

**Simplifying makefiles and accelerating builds**

At many customer sites that I visit, I often find makefiles that are arcane and convoluted.

Organizations that have used make or clearmake for lengthy projects grow afraid to touch their makefiles for fear they will break the build. The result is that these files grow over time with no controlling plan. I compare this phenomenon to the Winchester Mystery House, near San Jose, California, which has stairways and corridors that go nowhere, and doors that open into empty spaces or walls.

Like this house, many makefiles have been built on over the years with no blueprints for guidance. And people are afraid to do anything to them, other than to add what is absolutely required to make new builds. Typically, these makefiles are not optimized or pruned, and no one knows exactly how they work. Yet organizations rely on them to build their systems.

I always encourage customers to periodically review/rewrite project makefiles to avoid the Winchester Mystery House syndrome. Plus, it is always useful to have someone around who actually knows how to build the software. If you don't know exactly what your makefile is doing but you can build your software, then you can use the IBM Rational ClearCase configuration record to create a basis for a new makefile. Simply build your application with an audited build using clearmake. When the build is
done, run cleartool catcr -makefile myapplication. This will output the configuration record in makefile format. This makefile may not be optimized, but it can be more understandable and provide a basis for optimizations.

Of course, using express builds can also accelerate the build process. And you may also want to consider performing parallel distributed builds in UNIX environments or parallel builds in multi-processor Windows environments. See the IBM Rational ClearCase manual, *Building Software with Rational ClearCase* for more information on setting up a parallel build.

A final note on build performance and MVFS caches: be careful of using many -I directives in compiler search paths. The -I <dir> compiler directive is used to designate <dir> as one of the directories in which to search for header files used in source compilation. For each header included in a source file, the compiler will search each directory in turn until it finds a match, then it moves on to the next. The MVFS caches file information in the dirent and enoent caches, the latter caching the pathnames at which each file sought was not found. These caches provide a faster means of lookup, both for finding correct and avoiding incorrect pathnames. If you use many -I directives for many directories, and each of those directories has many files of which only very few are actually needed by the compiler, the effect is to fill the enoent cache with every directory entry that was not a match for every file that was not found. With sufficiently large numbers of file entries, the enoent cache becomes full and searching it becomes less efficient. If you think this may be affecting the performance of your builds, consider linking all headers and other compile-time included files to a single directory, and include this one directory in a -I directive.

**Shared network resources and IBM Rational ClearCase Doctor**

Shared network resources -- such as the domain controller, name servers, and so on -- also affect IBM Rational ClearCase performance. On Windows systems, analysis in this area is comparatively easy; just let the IBM Rational ClearCase Doctor do the work.

Figure 12 shows results from the standard static analysis that Rational ClearCase Doctor performs when it starts up. It tests host name resolution and reports how long it took to contact the domain controller, to acquire a license, and to contact the registry server.
Most customers I talk with know about these reports, but few know that IBM Rational ClearCase Doctor also does dynamic analysis. From the Analysis menu, you can select Server Accessibility Analysis... to access a variety of dynamic analysis capabilities, as shown in Figure 13a.

Figure 12: Rational ClearCase Doctor shows response times for shared network resources

Figure 13a: IBM Rational ClearCase Doctor provides server accessibility analysis capabilities
In Figure 13a, I've selected the radio button to check a specific path relative to a view and a VOB. After I click "Start Analysis," I see the results in Figure 13b.

![Figure 13b: The results of a server accessibility analysis](Click to enlarge)

The result shows me the response times to access the file I specified in the given view and VOB. In this case the response times were very fast, because everything was self-contained on my machine. However, if I notice that it is taking too long to check out or check in a particular file, I can use this dynamic analysis in Rational ClearCase Doctor to get a better picture of what is happening.

**Other performance factors**

There are a few other ways to prevent slowdowns of IBM Rational ClearCase.

In general, don't store thousands of files in the VOB root, which is not cached in the same way as other directories; instead, put them in subdirectories. In fact, I follow this practice for any file system that I use.

Also, if you are using a version of UCM from 2002 or earlier, try to keep the number of modifiable components in your project down to ten or fewer and avoid long-lived streams with large numbers of baselines. Either of these situations can adversely affect performance. Enhancements to UCM for IBM Rational ClearCase version 2003 have greatly improved these areas, so these limits no longer apply.

**Take it step by step**
Analyzing and improving IBM Rational ClearCase performance can be a complex undertaking. But like any complex task, it becomes manageable when you break it down into smaller pieces. In my experience, the performance stack provides an excellent framework for partitioning the job. As I move through the stack -- from the OS/Hardware layer to the IBM Rational ClearCase parameters and application layers, I keep in mind that even small performance gains can have a large impact on the productivity and efficiency of an organization in the long term. Spending a little time improving performance now can save a lot of time down the road.

---

**Appendix A**

The Visual Basic application shown in Figure 5 is available for [download here](#).

Note: This application is for instructional purposes only, and may not be suitable for production use.

---

**Notes**

1. My application is based on two books by Curt Aubley: *Tuning & Sizing NT Server* and *Tuning and Sizing Windows 2000 for Maximum Performance*, both published by Prentice Hall. These books provide more details on using native Windows tools, such as Task Manager and Performance Monitor, to pinpoint Windows performance bottlenecks. More information is available online at [http://www.tuningandsizingnt.com](http://www.tuningandsizingnt.com).

2. `truss` is specifically a Solaris utility (AIX has now emulated it in OS 5.x, but this version is not as sophisticated as that on Solaris 5.7 and later). MKS offers a Windows version, `truss.exe`. There are (at least) two types of `strace` for Windows, the better of them being that which is installed with Cygwin. Cygwin `strace` does provide time-deltas, whereas the downloadable standalone does not. Cygwin is careful to point out that `strace` is primarily designed for debugging the Cygwin dll. The analogous tool on HP 11.x is called `tusc`, and it provides output and functionality commensurate with that of `truss` on later versions of Solaris. AIX has a facility called 'trace' which, while more difficult to use, does provide a comprehensive system call-tracing facility.

3. `snoop` is primarily a Solaris tool. Microsoft's `Netmon` is available for Windows platforms; `ethereal` is available for multiple operating systems. In addition, `tcpdump` provides similar functionality for UNIX versions other than Solaris (including Linux).

4. This manual ships with the IBM Rational ClearCase product.

5. Unified Change Management is IBM Rational's "best practices" process for managing change, from requirements to release. Enabled by IBM Rational ClearCase and IBM Rational ClearQuest, UCM defines a consistent, activity-based process for managing change that teams can apply to their development projects right away.

6. This manual ships with the IBM Rational ClearCase product.

7. IBM Rational ClearCase uses a hashing strategy to quickly eliminate most derived objects,
but large development efforts can still have several potential matches that must be examined in more detail.

8 The Perl ClearCase package is available from CPAN at http://www.cpan.org. The package ClearCase::CtCmd, provided by IBM Rational, is a compilable interface into the cleartool subcommand process table. It bypasses the startup cost of a cleartool session by linking directly with the IBM Rational ClearCase libraries and using compiled entry points into the Rational ClearCase shared objects. This can significantly improve performance in scripted use of Rational ClearCase within Perl.

9 See my Rational Edge article "Using Perl with Rational ClearCase Automation Library (CAL)," for more information on this, and additional details on using CAL.

10 Sarah Winchester, an heir to the Winchester rifle company fortune, had constant work done on the house for 38 years. See http://www.winchestermysteryhouse.com/

References


For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
"Component and Deployment Diagrams"
(Chapter 5*)

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.

In last month's issue we featured Chapter 4, "Use-Case Diagrams." This month we offer Chapter 5, which shows how to model a system's implementation and environment, respectively. With UML-based component modeling, a special type of structural modeling, you can represent the system implementation and determine the elements of the system on which the implementation will focus. A deployment model, in contrast, represents the external resources that these components require and helps teams determine how deployment activities will make the system available to users. Both types of modeling usually begin when the system design is fairly complete. Covering basics for both types of models, this chapter includes a discussion of nodes and various relationships with respect to components and nodes.

*Chapter 5 is posted in its entirety by permission from O'Reilly (www.oreilly.com).

Chapter 5 pdf file (748 K)
For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!

Copyright Rational Software 2003 | Privacy/Legal Information
Learning UML

Sinan Si Alhir
This chapter focuses on component and deployment diagrams, which depict the implementation and environment of a system, respectively. First, I introduce component and deployment diagrams and how they are used. Next, I discuss components and nodes, which are elements depicted on those diagrams. Finally, I discuss various relationships relating to components and nodes. Many details of our project management system that were not fleshed out in Chapter 2 are more fully elaborated here, and throughout the chapter, I include suggestions relating to component and deployment diagrams.

Component modeling is a specialized type of structural modeling concerned with modeling the implementation of a system. Using the UML, you can communicate the implementation of a system using component diagrams. You usually apply component modeling during design activities to determine how implementation activities will build the system; that is, to determine the elements of the system on which implementation activities will focus. Component modeling typically starts after the design of the system is fairly complete, as determined by your system development process.

Deployment modeling is a specialized type of structural modeling concerned with modeling the implementation environment of a system. In contrast to modeling the components of a system, a deployment model shows you the external resources that those components require. You typically apply deployment modeling during design activities to determine how deployment activities will make the system available to its users; that is, to determine the elements of the system on which deployment activities will focus. Like component modeling, deployment modeling usually starts after the design of the system is fairly complete, as determined by your system development process.

Components

As mentioned in Chapter 2, a component is a part of the system that exists when the system is executing. For example, the project management system may be decomposed into the following components:
A **user interface component**  
Responsible for providing a user interface through which users may interact with the system

A **business-processing component**  
Responsible for implementing business functionality, including all the project management functionality provided by the project management system

A **data component**  
For implementing data storage functionality

A **security component**  
Provides various forms of security functionality to the business-processing and data components, including user authentication and verifying user privileges when accessing data

Components 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 components as well as specific components of a class. When speaking of a class of components, it’s customary to use the terms component or **component class**. Thus, while you might think of a component as a specific thing, in the UML, a component really represents a class of things. When speaking of a specific component of a class, use the term **component instance**.

A component exists during execution time and requires a resource on which to execute, which I talk about in the next section, “Nodes.” In the UML, a component is shown as a rectangle with two small rectangles protruding from its side. The rectangle is labeled with the name of the component class.

Figure 5-1 shows various components associated with the project management system, including user interface, business-processing, data, and security components.

![Figure 5-1. Components of the project management system](image)

A component instance is a specific component. For example, specific components of the project management system include:

A **web user interface component instance**  
Allows users to access the project management system via the Web

A **client/server user interface component instance**  
Allows users to access the project management system in a client/server environment
A local data component instance
Stores project management data for a specific user or group of users

An enterprise data component instance
Stores project management data for a complete organization

A component instance is shown similar to a component class, but is labeled with the component instance name followed by a colon followed by the component class name, with all parts of the name fully underlined. Both names are optional, and the colon is present only if the component class name is specified.

Figure 5-2 shows various component instances of the component classes in Figure 5-1, including two user interface component instances, named Web and Client Server, two data component instances, named Local Data and Enterprise Data, a nameless business processing component instance, and a nameless security component instance.

Nodes

A node is a resource that is available during execution time. (Nodes were mentioned in Chapter 2.) Traditionally, nodes refer to computers on a network, but in the UML a node may be a computer, printer, server, Internet, or any other kind of resource available to components. For example, the project management system may be deployed on the following nodes:

A desktop client
On which the user interface component executes

A printer
Which the project management system uses to print reports

A business-processing server
On which the business-processing component executes

A database server
On which the data component executes and where project-related information is stored
Nodes 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 nodes, as well as specific nodes of a class. When speaking of a class of nodes, it’s customary to use the terms node or node class. Thus, while you might think of a node as a specific thing, in the UML, a node really represents a class of nodes. When speaking of a specific component of a class, use the term node instance.

A node is available during execution time and is a resource on which components may execute. In the UML, a node is shown as a three-dimensional rectangle labeled with the node’s name.

Figure 5-3 shows various nodes associated with the project management system, including a desktop client, business-processing server, database server, and printer node.

![Nodes used by the project management system](image)

A node instance is a specific node. For example, specific nodes used by the project management system include:

- **A desktop client node instance**
  - Used by Jonathan to access the project management system

- **A desktop client node instance**
  - Used by Andy to access the project management system

- **A group business-processing server node instance**
  - Used by a group of users to manage projects

- **An enterprise business-processing server node instance**
  - Used by a complete organization to manage projects

A node instance is shown similarly to a node class but labeled with the node instance name followed by a colon followed by the node class name, all fully underlined. Both names are optional, and the colon is present only if the node class name is specified.

Figure 5-4 shows various node instances of the node classes in Figure 5-3, including two desktop client node instances, named Jonathan’s Computer and Andy’s Computer, two business-processing node instances, named Group Server and Enterprise Server, a printer node instance, named Group Printer, and a database server node instance.
Dependencies

Figure 5-1 shows components associated with the project management system, and Figure 5-3 shows nodes associated with the project management system, but how are components related to undifferentiated and differentiated classes, packages, subsystems, and to other components and nodes? Specialized types of dependencies—called reside, use, and deploy dependencies—address these questions. The next few sections in this chapter discuss these specialized types of dependencies. Dependencies in general are discussed in Chapter 3.

Reside Dependencies

A reside dependency from a component to any UML element indicates that the component is a client of the element, which is itself considered a supplier, and that the element resides in the component. The element may be an undifferentiated or differentiated class, package, or subsystem. An element may reside in any number of components, and a component may have any number of elements that reside in it.

A reside dependency is shown as a dashed arrow from a client component to a supplier element marked with the reside keyword. Figure 5-5 shows that the User Interface and Utility packages reside in the User Interface component. Because the User Interface package depends on the Utility package, the User Interface and Utility packages must reside in the same component; otherwise, the User Interface package would not be able to use the Utility package.

Figure 5-6 shows that the Business Processing subsystem and Utility package reside in the Business Processing component. Because the Business Processing subsystem provides the IBusiness Processing interface, the Business Processing component also provides the interface. Again, because the Business Processing subsystem depends on the Utility package, the Business Processing subsystem and Utility
package must reside in the same component; otherwise, the Business Processing subsystem would not be able to use the Utility package. Remember, it’s perfectly fine for an element to reside in more than one component. For example, the Utility package resides in both the User Interface and Business Processing components, and, as you will soon see, in the Data component.

Alternatively, an element that resides inside a component may be shown nested inside the component. Figure 5-7 shows that the Data subsystem and Utility package reside in the Data component. The Data subsystem is drawn inside the Data component, while the reside dependency to Utility is still drawn in the same manner as in Figures 5-5 and 5-6.

Notice that the Utility package resides in all the components in Figures 5-5, 5-6, and 5-7, because each component described in those figures has a package that uses the Utility package. Details of the Utility package are discussed in Chapter 3.

Use Dependencies

A use dependency from a client component to a supplier component indicates that the client component uses or depends on the supplier component. A use dependency
from a client component to a supplier component’s interface indicates that the client component uses or depends on the interface provided by the supplier component. A use dependency is shown as a dashed arrow from a client component to a supplier component or a supplier component’s interface. The dependency may be marked with the use keyword; however, the keyword is often omitted because this is the default, and the meaning is evident from how the dependency is used.

Figure 5-8 shows how the various components of the project management system are related:

*The User Interface component*
Uses the Security component and the IBusiness Processing interface provided by the Business Processing component

*The Business Processing component*
Uses the Security component and the IProducible and IConsumable interfaces provided by the Data component

*The Data component*
Uses the Security component

**Deploy Dependencies**

A deploy dependency from a client component to a supplier node indicates that the client component is deployed on the supplier node.

A deploy dependency is shown as a dashed arrow from a client component to a supplier node marked with the deploy keyword. Figure 5-9 shows that the User Interface component is deployed on the Desktop Client node.

Figure 5-10 shows that the Business Processing component is deployed on the Business-Processing Server node.
Alternatively, a component that is deployed on a node may be shown nested inside the node. Figure 5-11 shows that the Data component is deployed on the Database Server node.
Communication Associations

Figure 5-3 shows nodes associated with the project management system, but how are those nodes related? A specialized type of association, called a communication association, addresses the question of how nodes are related. (Associations are discussed in Chapter 3.)

A communication association between nodes indicates a communication path between the nodes that allows components on the nodes to communicate with one another. A communication association is shown as a solid line between nodes. Figure 5-12 shows that the Business-Processing Server has a communication association with the Desktop Client, Printer, and Database Server nodes.

Figure 5-13 combines Figure 5-8 and Figure 5-12 to show how components are related to nodes. Notice that if two components are related and reside on different nodes, the nodes must have a communication association between them to allow the components to communicate; otherwise, the components are not able to communicate and be related to one another. For example, if the communication association between the Desktop Client and Business-Processing Server nodes was removed, the User Interface component could not be related to the IBusiness Processing interface and Security component. If the communication association between the Business-Processing Server and Database Server nodes was removed, the Data component could not be related to the Security component, and the Business Processing component could not be related to the IProducible and IConsumable interfaces.
Exercises

1. Describe Figure 5-14: identify components and nodes, and describe the relationships among components and nodes.

2. Describe Figure 5-15: identify the various elements and their relationships.
   Update the diagram stepwise to show the following details. After each step, check your answers against the solutions shown in Appendix B:
   a. The User Interface package uses the IView and IPrint interfaces provided by the Reporting subsystem.
   b. The User Interface and Utility packages resides in a User Interface component.
   c. The Reporting subsystem and Utility package reside in a Reporting component.
   d. The User Interface component is deployed on a Desktop Client node.
   e. The Reporting component is deployed on a Report Server node.
   f. The Desktop Client node is connected to the Report Server node, and the Report Server node is connected to a High-speed Printer node.
Figure 5-14. Components and nodes for the project management system

Figure 5-15. Packages and a subsystem
The illusion of simplicity

by Grady Booch
Chief Scientist
Rational Software
IBM Software Group

I recently met with the CTO of a large company who was charged with providing the technology infrastructure upon which his organization’s developers could craft their applications. I spent much of our meeting commiserating with him about the many challenges he faced in trying to weigh the tradeoffs between Microsoft’s .NET and IBM’s WebSphere while at the same time keeping a large legacy platform alive. One thing that he said stuck with me all day: Ultimately, he said, he was rewarded for making things simple.

Simplicity is an elusive thing. In a software-intensive system that might consist of hundreds of thousand of lines of custom code on top of several million lines of middleware code on top of several million lines of operating system code, there is an essential complexity that cannot be eliminated. From the perspective of its end users, simplicity manifests itself in terms of a user experience made up of a small set of concepts that can be manipulated predictably, logically, and consistently. From the perspective of those who deploy that system, simplicity manifests itself in terms of an installation process that addresses the most common path directly while at the same time making alternative paths accessible and intuitive. From the perspective of the developers who build that system, simplicity manifests itself in terms of an architecture that is shaped by a manageable set of patterns that act upon a self-consistent, regular, and logical model of the domain. From the perspective of the developers who maintain that system, simplicity manifests itself in the principle of least astonishment, namely, the ability to touch one part of the system without causing other distant parts to fall off.
Simplicity is most often expressed in terms of Occam's Razor\(^1\). William Occam, a 14th century logician and Franciscan friar stated, "Entities should not be multiplied unnecessarily." Isaac Newton projected Occam's work into physics by noting, "We are to admit no more causes of natural things than such as are both true and sufficient to explain their appearances." Put in contemporary terms, physicists often observe, "When you have two competing theories that make exactly the same predictions, the one that is simpler is the better." Finally, Albert Einstein declared that "Everything should be made as simple as possible, but not simpler."

As software developers, we cannot measure simplicity, but we do know it when we see it. As C.A. R. Hoare observed, "There are two ways of constructing a software design; one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

As such, you'll often hear programmers talk about "elegance" and "beauty," both of which are projections of simplicity. Don Knuth's work on literate programming -- wherein code reads like a well-written novel -- attempts to bring beauty to code. Dick Gabriel's work on the "quality with no name" which builds upon the architect Christopher Alexander's work, also seeks to bring beauty and elegance to systems. In fact, the very essence of the patterns movement, from the Gang of Four and beyond, encourages simplicity in the presence of overwhelming complexity by the application of common solutions to common problems.

Simplicity in software design is important because we as humans have a limited capacity for dealing with complexity. Indeed, I often know that my designs are getting simpler when the implementation gets smaller, primarily due to refactoring that extracts these common solutions. For example, as we were working on the original UML metamodel, we'd see the number of classes in the model grow (as we added semantics) and then shrink (as we discovered some underlying simplicity that was previously hidden). This regular breathing of our metamodel was a good measure as to the health of our design; too rapid a rhythm and it was clear we were churning, but a steady, slowing rhythm told us that we were on track.

The entire history of software engineering can perhaps be told by the languages, methods, and tools that help us raise the level of abstraction within our systems, for abstraction is the primary means whereby we can engineer the illusion of simplicity. At the level of our programming languages, we seek idioms that codify beautiful writing. At the level of our designs, we seek good classes, and in turn good design patterns, that yield a good separation of concerns and a balanced distribution of responsibilities. At the level of our systems, we seek architectural mechanisms that regulate societies of these classes and patterns.

Buckminster Fuller once said, "When I am working on a problem, I never think about beauty. But when I have finished, if the solution is not beautiful, I know it is wrong." In short, beauty in software is an important after-effect of simplicity in design.
Notes

1 In particular, see http://math.ucr.edu/home/baez/physics/occam.html

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!
A case study of using IBM WebSphere Studio Application Developer 5.0 with IBM Rational ClearCase LT

Part I: Installation, integration, and Unified Change Management

by Ali Manji and Ying Zhao

Editor's note: Each month, we feature one or two articles from IBM 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 IBM Rational Developer Network now!

This article, the first in a series, focuses on the installation and integration of IBM® Rational® ClearCase® products with IBM WebSphere® Studio V5. Other articles in this series will describe unified change management (UCM) principles using Rational ClearCase, and best practices for team development of J2EE applications.

IBM offers various application development environments for different audiences, all of which extend the open-source initiative known as Eclipse. These products include WebSphere Studio Site Developer, WebSphere Studio Application Developer, WebSphere Studio Application Developer Integration Edition, and WebSphere Studio Enterprise Developer. For more information on which product is most suitable for you, see References below. From here on, this article will use WebSphere Studio to refer to the full WebSphere Studio family of products. Regardless of which WebSphere Studio product or products you are using, completing a large software project requires a closely coordinated team effort. Therefore, using a software configuration management (SCM) product such as IBM Rational's ClearCase or ClearCase LT is essential for effective change management and successful project delivery.
For information on installation and use of IBM Rational ClearCase within WebSphere Studio V5.0, see the excellent article by Reginaldo W. Barosa, "Integrating Rational ClearCase LT and WebSphere Studio Application Developer V5." While that article goes into detail about installation and basic Rational ClearCase operations, this article focuses on additional installation issues and use of the tools in the UCM context.

**Improved team programming support in Eclipse 2.0**

The WebSphere Studio V5 family of products extends Eclipse V2, whereas the WebSphere Studio V4 products extended Eclipse V1. Extending or contributing to Eclipse involves providing one or more related plug-ins that together compose a feature.

Support for team development improved significantly in Eclipse V2, providing more stability and flexibility. By leveraging this new version of Eclipse, IBM WebSphere Studio V5 can now support multiple repository vendors concurrently. Therefore, you can have one project in a WebSphere Studio workspace managed by IBM Rational ClearCase, and another project managed by Concurrent Version System (CVS). Consequently, multiple repository client implementations (features and plug-ins) can be installed and can co-exist. In WebSphere Studio V4, you had to make a one-time decision during installation on which repository you wanted to use, and you were bound by that decision. A further improvement is that in WebSphere Studio V5, you can install repository clients from third-party vendors after the WebSphere Studio installation is complete. Therefore, organizations do not have to commit to a single team repository vendor when developing different applications with WebSphere Studio.

The improved flexibility of the team support in Eclipse V2 does not stop at multiple repository vendor support. It also supports multiple client implementations for the same repository. For example, a repository vendor may provide two different client implementations for the same repository -- one for novice users and one for advanced users. With ClearCase, IBM Rational provides one client implementation for both versions of its ClearCase servers: IBM Rational ClearCase LT, and the enterprise-scalable IBM Rational ClearCase. Furthermore, you can have two separate client implementations based on the repository in use. Another scenario for using multiple client implementations for the same repository derives from the open-source nature of Eclipse technology. That is, a poorly integrated client implementation for a particular vendor's repository could soon face competition from client implementations provided by ISVs. Therefore, not only do repository vendors face competition in providing the best repository server, but they may also face competition in providing a client implementation that integrates well with Eclipse-based products, most notably IBM WebSphere Studio.

**Installation issues**

Team programming generally requires at least two main components: the server (sometimes referred to as the repository) and a client. The client
resides on developers' machines and lets them carry out two key operations -- sending changes to the shared repository, and retrieving changes made by other developers into their private workspaces. These two operations are collectively referred to as synchronization.

IBM WebSphere Studio ships with both the client and server components for IBM Rational ClearCase LT. The client component has two logical subcomponents. The plug-ins and features that integrate into WebSphere Studio (hereafter referred to as the ClearCase client SCM Adapter), and a separate client program that performs the actual communication with the server. The client program shipped with WebSphere Studio is compatible with Rational ClearCase LT only. If you are using the full Rational ClearCase server, you must install the client program that IBM Rational ships with the full Rational ClearCase server. However, the SCM Adapter shipped with WebSphere Studio is compatible with the client program for both Rational ClearCase LT and the full Rational ClearCase. For information on the functional differences between Rational ClearCase LT and the full Rational ClearCase server, see References below.

The IBM Rational ClearCase LT Server, unlike the IBM Rational ClearCase client and SCM Adapter, can be installed on a single dedicated machine and does not need to be installed on every developer's workstation. A single Rational ClearCase server (LT or the full Server) can be used to manage the repositories of many different projects. Managing projects, project milestones, and SCM policies from the Rational ClearCase Server can become a full-time job within the team. Whoever fills this role should note that the Rational ClearCase LT server that ships with IBM WebSphere Studio obtains its licensing information automatically from the WebSphere Studio installation. Therefore, WebSphere Studio should be installed on the server machine as a Rational ClearCase LT server. Naturally, if you are using the full Rational ClearCase server, you will have a license key directly from Rational and will not need to install WebSphere Studio on the same machine as the full Rational ClearCase server.

The client program for both IBM Rational ClearCase LT and the SCM Adapter are installable from the IBM WebSphere Studio CDs. For the SCM Adapter to be successfully installed, you must explicitly select the Rational ClearCase Team Adapter during the installation of WebSphere Studio, as shown below:
If you've already installed WebSphere Studio without selecting this option, you can install the SCM Adapter from the CD.

The client program for IBM Rational ClearCase LT with which the Rational ClearCase SCM Adapter communicates can also be installed from the IBM WebSphere Studio CDs. The installation program is separate from WebSphere Studio, and is the same one used to install the Rational ClearCase LT Server. When the installation begins, you are prompted to choose the desired component to install the client or the server, as shown in Figures 2 and 3:
In Figure 3, despite the green checkmark, you may see a warning dialog after you click **Next**. You can ignore this warning. The client installation will also prompt you to provide the hostname for the server installation:
Both the IBM Rational ClearCase client program and the server can communicate successfully only when running under users that are part of a network domain and when both are installed on Windows®.

**IBM Rational ClearCase UCM vs. base IBM Rational ClearCase**

Rational's UCM methodology involves tools as well as processes. IBM Rational provides ClearQuest and ClearCase for those who want to take full advantage of UCM for project development. IBM Rational ClearQuest is not required, but is strongly recommended to fully exploit the UCM methodology. IBM Rational ClearCase is a general-purpose SCM tool that manages various types of applications in many different environments. Its strength lies in its rich functionality and flexibility. This general-purpose flexibility of Rational ClearCase is often referred to as *base ClearCase*. After extensive observations of customer usage, Rational observed that most users follow a common usage pattern, and UCM refers to this pattern with the addition of key "best practices" and tools support (*Rational ClearCase UCM*). For more information on the Rational UCM methodology, see [References](#) below.

Base ClearCase gives you with most if not all of the function that SCM tools provide for change management. IBM Rational ClearCase UCM recognizes that changes to artifacts normally occur in the context of certain project-related *activities*, where an activity is an object that records the set of files (the *change set*) that a developer creates or modifies to complete a development task such as a bug fix or the implementation of a feature. The completion of a project activity may affect one or more project *components*. Thus, Rational ClearCase UCM is essentially founded upon two basic concepts: *activity-based change*
management, and component management. In addition, the UCM process identifies key team roles, including project manager, developer, and integrator.

Figure 5: Project manager, developer, and integrator work flows

Figure 5 illustrates the key roles involved in the UCM process and the various tasks required by these roles. The project manager's role involves assigning components to teams, and assigning activities to various components. The other project manager responsibility is to create the actual Rational ClearCase projects for work to be delivered to and to set policies for these projects on how change is to be managed. The role of the developer is to complete and deliver work activities, which involves making changes to files and directories under source control, and unit testing those changes. The integrator's primary responsibility is to accept delivered activities from developers, create new baselines for the various components to which activities have been delivered, build components against the new baseline, assign testing of the new baseline, and then promote the new baseline once testing is complete.

In addition to IBM Rational ClearCase, the other main UCM tool provided by IBM Rational, ClearQuest, contributes to the UCM process through its ability to manage and track activity progress. However, it is not required for the developer role. For more information on IBM Rational ClearQuest and on the roles involved in the UCM process, see References below.

Conclusion

Among the key assets that IBM Rational ClearCase integration provides to IBM WebSphere Studio, particularly in V5, are the UCM methodology and tools. The Rational ClearCase client is divided into two pieces -- the actual
client that communicates directly with the server, and the client adapter that integrates into WebSphere Studio as an Eclipse feature. Part 2 will focus on the role of the developer in the UCM process.

References

- IBM WebSphere Studio Zone
- Eclipse home page
- IBM Rational ClearCase product page
- IBM Rational Whitepaper: "Integration between the Rational ClearCase and the IBM WebSphere Studio product family"
- "Integrating Rational ClearCase LT and WebSphere Studio Application Developer V5"

Acknowledgments

The authors wish to thank Dan England, Senior Technical Engineer, IBM Rational Software, for reviewing this article.

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

Copyright Rational Software 2003 | Privacy/Legal Information