Team Retrospectives — for better iterative assessment

by Ellen Gottesdiener
"Want iterative? Use RUP®!" That, in a nutshell, is our value proposition regarding software development. In other words, if you are committed to an iterative, or spiral, model of software development -- as opposed to a sequential, or waterfall, model -- then we believe you need the guidance of others who have blazed an iterative trail or two, which is precisely what the Rational Unified Process® offers. Yet, even with RUP to serve as their map and compass, grumpy teams may tend to dash off on the next leg of the journey without assessing and correcting problems they recently encountered. Such teams never reach their full potential, even if they do manage to complete the project.

In this month's cover story, Ellen Gottesdiener presents a great antidote to this "never look back" tendency. She advocates team retrospectives to structure the periodic assessments RUP recommends. Retrospectives focus "not only [on] the condition of the code and technology, but also the team's performance and the quality of their teamwork." As Ellen suggests in her conclusion, taking time to reflect on a just-completed iteration can have the effect of speeding up the entire process.

Len Dimaggio offers a hefty batch of testing techniques for one of today's most frequently discussed modes of development: integration. In Part I of what he calls a step-by-step recipe, Len shows us where, when, and how to begin "testing at the boundaries" of software built from third-party components. Murray Cantor takes us on a quick, cautionary tour of the perils of using functional decomposition as a means to establish the system architecture and set subsystem requirements. See why this approach can lead to costly and inefficient software systems.

Do you prefer having software development philosophy grounded in details as specific as the source code itself? Then Michael Harrison's "Thoughts on the Craft of Programming: Abstraction, Refactoring, and How Changes
Introduce Bugs" is for you. In this unique piece, Michael shows how scientific reasoning can (and should) live alongside a developer's sense of aesthetics in code craft. Also for the more technical reader, an article by Rational quality engineer Susan McVey explains how her distributed team uses a "Virtual Status Portal" -- governed by IBM Rational® ClearCase®-- to keep quality testing on track.


And there's a confession! In fact, more than one, from former Rationalite Cindy Van Epps, who candidly reveals how RUP thinking will influence the way you understand parenthood, plumbing, food -- even software development. It's shocking...

Happy iterations,

Mike Perrow
Editor-in-Chief
Team Retrospectives — for better iterative assessment

by Ellen Gottesdiener
Principal
EBG Consulting

To build a positive project team it is essential to make self-reflection and continuous learning a part of your organization's culture. Perhaps the single most efficient way to do this is by incorporating ongoing team retrospectives into your project. You can think of these retrospectives as the "flesh" of the iteration and milestone assessments that the Rational Unified Process,® or RUP® recommends. The RUP provides only a skeleton for these assessments, leaving teams to structure the activities on their own. RUP suggests that project team members gather after each iteration and milestone to reflect on their performance and process. A team retrospective approach gives structure to these sessions. The purpose is to provide progressive learning, sustenance, and improvement. A hallmark of agile teams, retrospectives allow all team members to iteratively test and develop not only solid software but also effective teamwork.

Do Your Assessments Need More Structure?

Somewhere in a corner conference room, a team is holding a meeting that should be an iteration assessment, but they are instead focusing on what to do next. If we were magically to become privy to the thoughts behind their words, here's how the discussion would go...
**What they say:** | **What they are really thinking:**
---|---
**Tom (project lead):**
"Okay, now that we’re finished with the first iteration, we’d better get going and add these three use cases for iteration two."

"Yikes! If we don’t catch up, we’re going to really get behind!"

**Rita (requirements analyst):**
"I admire your optimism, Tom."

"How can we start the next iteration? We haven’t finished the prototype with the two additional scenarios the customers added on Tuesday. They blew their stack when they didn’t see those scenarios yesterday! What am I gonna tell them?"

**Tom:**
"Thanks, Rita. Joan, the test team needs to make those scripts available quicker. Can you do that?"

"Gotta cut a week and a half somewhere, or this project’s gonna be toast."

**Joan (test lead):**
"We’ll try, but we may be short-handed this week."

"Oh, give me a break! They didn’t define the requirements or scenarios until after they started to code! So now they expect my team to take up the slack. How can we do that?"

Seem familiar? If so, it may be comforting to know that your team is not unique. Unfortunately, even teams that have adopted many of the best practices in RUP often do not understand the purpose of an iteration assessment. They attend the meetings but focus only on what’s ahead instead of looking back over what they’ve just completed. They don’t share their conflicting concerns and needs or their differing perspectives. As a result, they move from iteration to iteration or project to project, repeating the same mistakes and failing to capitalize on their successes. The team never achieves its potential in terms of both project results and building a knowledge base for both present and future team members.

Fortunately, there is a simple and inexpensive way to turn things around -- to help team members learn how to stop repeating mistakes and recreating success by accident. It’s called structuring your assessments as a team retrospective.

**What Is a Team Retrospective?**

Unlike a classic meeting, which typically involves sharing information or status, a retrospective is a tool for learning that also generates new
information and action plans. It is a means of assessing not only the condition of the code and technology, but also the team’s performance and the quality of their teamwork. The team uses a skilled, neutral facilitator to guide its work, and includes everyone who was involved in creating the deliverables the group is reviewing or was otherwise involved in the project during the same time period.

Retrospectives can be incorporated into iteration and milestone assessments as well as post-project reviews. For a typical iteration or milestone assessment, a retrospective will take two to six hours; you will need more time if you are retrospecting a completed project.1

**Iterating Requires Continual Retrospection**

An iterative development lifecycle such as the one outlined in the RUP is composed of a series of cycles (iterations). The team plans work, does it, checks it, takes corrective action or ensures they repeat prior desirable behaviors, and moves on. They repeat this cycle -- essentially the "plan-do-check-act" process known in the quality improvement community -- across all phases (Inception, Elaboration, Construction, and Transition) until they deliver the entire product (see Figure 1). In an iterative development lifecycle, the project is treated as a live learning lab; project quality evolves through the ongoing, corrective feedback gained from each iteration via an iteration assessment, and from each phase via a milestone assessment, both of which can be structured as retrospectives. As we’ll discuss, retrospectives offer other benefits, but their main job is to give you a structure for exploiting the advantages of an iterative development cycle.

![Figure 1: Assessments/Retrospectives for Iterative Projects. Each project includes an assessment/retrospective following each iteration. The retrospective includes a series of steps: Readying, Past, Present, Future, and Retrospect the Retrospective. The team's job is to reflect, inquire, analyze, and use what it learned to plan the next iteration.](image)

**Learning from Retrospectives**

* A retrospective is a reality-based learning experience resulting in action
For learning to stay with you, it must have immediacy, relevance, and self-direction. Let's translate that into implications for your team retrospectives.

- **Immediacy** means that you have opportunities to apply what you learn very soon after the learning occurs (the thought process is, "I can remember it"). That's why it's important to time your retrospectives within days of the endpoint you are retrospecting.

- **Relevance** means that the learning is important to us, and that it applies to our current situation ("I care," "I need it," and "I can practice or test this -- real soon"). You should plan your retrospective around the team members' goals for the session, being sure to cover topics the team cares about.

- **Self-direction** involves taking ownership and control of our own learning and making necessary changes ("I own it," "I choose it," and "I will do it"). Always include steps in the retrospective for everyone to decide and plan what to change and how to change it.

### Planning and Running Successful Retrospectives

Successful retrospectives have a number of characteristics:

- They are planned.
- They are held multiple times throughout the project.
- They involve the project community.
- They are led by a neutral, skilled facilitator.
- They use data from the project.
- They acknowledge that feelings count.
- They follow a structure.
- They are the basis for change.

Let's look at each of these characteristics in detail.

### Plan for a Successful Retrospective

In advance of the session, the team, with the help of the retrospective facilitator, plans what will be covered, who will be there, and how long the retrospective will take. For example, a team retrospecting on the completion of its Inception phase might reflect on customer involvement, management support, and team communications. A team retrospecting on the first iteration of Construction might focus on development tool effectiveness, defect detection, and architectural design. Here are some questions the team might cover:

- How clear was our product vision when we began the iteration? How
clear is it now that we have finished it?

- Were customers involved appropriately in this iteration? How did they react to the work we did?
- Based on this iteration, does our project plan seem sound, or does it need changing?
- What risks did we reduce in this iteration? What risks do we still face, and do we have effective ways to manage them?
- How well did the team communicate during this iteration? If there were misunderstandings or failures, why did they happen, and how can we make improvements?
- Did we have the right people to cover essential roles and responsibilities? Were the lines of responsibility clear to all team members? Did our team structure and organization work effectively?
- What tools and technologies did we use in this iteration? How effective were they?
- How well did the decisionmaking process work in this iteration? If anything went wrong, why did it happen?
- Did management provide adequate support for our work? If not, what could they have done differently?
- Are we working with high-quality requirements? How volatile are the requirements, and why?
- Do we have an effective change control process to guard against scope creep? How well did we manage scope during this iteration?
- How effective was our test plan? What did the tests we conducted point out?
- Are we creating or working with a high-quality architecture?
- Are we generating end-user documentation and training materials as we go and testing them on end users?
- Are our customers getting ready to work effectively with the new product? Did they make organizational or culture changes during this iteration to prepare for implementation?

Align the content of your retrospective with the goals for the iteration you've just completed. For example, suppose you have finished an iteration of the Elaboration phase with the goals of verifying the breadth of requirements, testing the validity of the overall project scope, and finding missing requirements. Your post-iteration retrospective should focus on requirements quality and volatility.

Table 1 provides a list of example questions your team might address in retrospectives for each phase of the RUP. (Also see an expanded list under Retrospective Questions by RUP Project Phase in the Appendix.)

Table 1: Example Retrospective Questions by Project Phase
<table>
<thead>
<tr>
<th>Project Phase</th>
<th>Potential Retrospective Questions</th>
</tr>
</thead>
</table>
| Inception     | ● How well did we delineate and communicate the product vision?  
               | ● How clear is our scope? How might we make it more clear, if necessary?  
               | ● How complete and accurate was our business case?  
               | ● Are we using risks to guide the project? Why? Why not? |
| Elaboration   | ● Did we select the right strategy for creating our architectural prototype? If so, how did we know? If not, how did we know?  
               | ● How much did our requirements change, and why?  
               | ● How did we involve customers? In what ways did that involvement work to our benefit or work against it?  
               | ● Was our architectural prototype sufficient to reduce risk and verify customer expectations? Is so, why? If not, why not? |
| Construction  | ● What was the quality of our code? How do we know?  
               | ● How well did we plan and execute our testing?  
               | ● How did our development, testing, and configuration management tools and processes work for us? What one thing would you change?  
               | ● How did the testers and developers interact? |
| Transition    | ● How well did we make use of beta testing to stabilize our product?  
               | ● How smooth was our plan and execution of database conversions?  
               | ● Was the customer ready for the product? How do we know?  
<pre><code>           | ● How effective is the product's support documentation? |
</code></pre>
<table>
<thead>
<tr>
<th>End-of-Project</th>
<th>All Phases</th>
</tr>
</thead>
<tbody>
<tr>
<td>- What are you most proud of in this project?</td>
<td>- What happened that surprised us?</td>
</tr>
<tr>
<td>- What one moment stands out the most for you?</td>
<td>- How well are we communicating with each other?</td>
</tr>
<tr>
<td>- What did we learn about iterative development from this project?</td>
<td>- Are we collecting the right metrics?</td>
</tr>
<tr>
<td>- What one recommendation would you make to other teams about their use of RUP?</td>
<td>- What one thing do you want to remember to do again in the next iteration?</td>
</tr>
</tbody>
</table>

### Retrospect Multiple Times Throughout the Project

Structure every RUP review and assessment as a retrospective as you progress through the project. Perform a final assessment when you close out the project -- even if the project was canceled or postponed. Use feedback from each retrospective immediately to improve and plan your next set of project work.

It's a good idea to calibrate the timing and length of your retrospective to the length and importance of the iteration. If you've just completed a short, two-week iteration that involves a build using a new technology, your retrospective can be short and highly focused on the goals of the iteration. Remember that retrospectives can be as short as one or two hours, especially if the team is in a groove with the process and observes healthy team norms. (For more information, see Defining Ground Rules for a Retrospective in the Appendix.)

Schedule iteration assessments/retrospectives for a few days after the end of the iteration. If you wait too long, memories of events will fade, or, even worse, team members will move on to the next iteration, and you'll miss out on planning for necessary changes and adjustments. Conduct your session in a comfortable space with blank walls for posting the work you'll do in the retrospective rituals. Make the team's collective knowledge visible, and you'll find it easier to examine the team's retrospective work. (For information about using walls for group work, see my article, "Specifying Requirements with the Wall of Wonder," in the November 2001 issue of The Rational Edge.) In addition, be sure to provide food or snacks for everyone during the session. If budgets are tight, ask different team members to make or bring food to each retrospective.

### Involve the Project Community
Everyone who played a role in producing artifacts during the target period is part of your project community and should participate in the retrospective. Include analysts, testers, the project manager, and customers if appropriate. Because roles change, specific attendees may vary.

In one project I worked on, during Elaboration we did a retrospective for our third and final requirements iteration. The business analyst, customer representatives, the requirements analyst, lead architects, the project manager, and the test lead attended. Six weeks later, when we did an iteration assessment for a Construction phase iteration, the retrospective included the developers, the test team, the project manager, the architects, and customer representatives. Remember to include customers; they are integral to the software development process.

For a large project, you may ask only a few representatives from each project role to attend the iteration assessment/retrospective to keep the size manageable (you will need more than one facilitator for more than fifteen participants). These representatives must be willing to prepare by interviewing people in the community ahead of time to gather data concerning key questions you want to address in the retrospective. They must act as ambassadors for the larger role community they represent. They are also responsible for sharing outcomes with their representative community immediately after the meeting. The retrospective facilitator (see below) should also plan other ways to give those not present a voice.

**Use a Neutral, Skilled Facilitator**

Someone who is substantially neutral and viewed as such by the participants should facilitate your session. This person must have good facilitation skills, including the ability to manage the group when things get emotional, plan the retrospective, and select appropriate activities for the time allotted. The facilitator must be comfortable with helping the group address not only its performance but also its process. This might include addressing previously taboo issues, directly addressing conflict, and helping group members explore their inferences and assumptions about each other.

A team member with solid skills can facilitate as long as she believes she can be neutral and the entire group agrees. If your organization is big enough, it's a good idea to grow facilitation skills across various development and engineering groups and then swap them around for various group sessions, such as retrospectives and requirements workshops.

On one project I served as both a team member and the interim retrospective facilitator; when we set up session ground rules, I made team members promise to inform me if I wasn't managing to remain neutral. We were able to proceed with that understanding. I also moved to one specific place in the room when I wanted to make a content contribution to the discussion, and then moved back to the center of the room to resume my facilitator role.
Use Project Data

The team should collect and bring data to the iteration assessment to perform the retrospective for the prior iteration and to plan the next one.

Choose a small set of metrics that are useful for both purposes. One useful technique is the goal-question-metric (GQM) approach:

- Define your goals.
- Ask relevant questions to check whether you achieved the goals.
- Derive metrics from the responses.

For example, if goals for an Elaboration iteration include saving time by using an optimum number of scenarios and increasing customer satisfaction by using prototypes effectively, you might ask questions such as:

- How many use cases were included?
- How many scenarios were generated and tested in the prototype?
- How many prototypes were developed and reviewed by the customer?
- How many days were needed to rework the prototype after a customer review?

(For more discussion on this topic, see http://www.processimpact.com/articles/metrics_primer.pdf.) This data should be captured so you can review it for patterns and trends and leverage it in planning future iterations and projects.

Project data can be useful to inform your inquiry into both team performance and process. You can ask "what if" questions to explore how your outcomes might have changed. For example, in one retrospective for an iteration assessment, we discussed the number of scenarios we generated as well as the ratio of scenarios to use cases, the number of business rules associated with each use case, and the number of workshops and customer reviews that we had used in that iteration. The data revealed a pattern of how many scenarios ("happy paths" as well as "unhappy paths," or exceptions) we used to elicit each use case and its associated business rules and to build a prototype. We speculated about how the iteration might have gone if we had used fewer use cases or more scenarios, fewer happy path scenarios, or more scenarios and fewer use cases. We also toyed with various ways to involve different customers to speed up the process. As a result of this inquiry, we adjusted our plans for defining and prototyping in our next iteration, and we delivered on our iteration goals with no major changes to the iteration prototype and very satisfied customers.

Acknowledge That Feelings Count
Because a retrospective goes beyond simply checking the outcomes or performance of a team, you should openly acknowledge the feelings of team members. Feelings may range from regret, sorrow, anger, and frustration, to pride, joy, and appreciation. When feelings are running strong, it can get in the way of getting the job done or interacting effectively, because we may selectively filter out information. A retrospective is not a therapy session, but it should not neglect feelings. A skilled facilitator knows that helping a team express feelings promotes healthy team collaboration. They do this by probing into the thinking and evidence that led to the feelings.

For example, during a retrospective for a Construction phase iteration, one developer replied curtly to comments that a tester was making. As facilitator, I noticed that the rest of the team immediately clammed up, and some people pursed their lips as if to say, "There he goes again!"

I restated what the developer had said and pointed out that his tone sounded angry; then I asked him whether I had interpreted his tone correctly. He sighed deeply and then admitted he was tired from working long hours and frustrated that his modules were the source of most of the outstanding defects. His anger turned to embarrassment and even shame; his anger was really directed at himself. Knowing this, the other participants were able to get past his "negative attitude" and see the real problem. The team began to explore what led to that situation and how they might help. This brief exchange helped improve the team's dynamics throughout the rest of the project.

A retrospective shouldn't focus solely on difficult, negative issues, although sometimes it is hard to elicit positive, complimentary feedback. Many organizational cultures value problem analysis and problem solving. But focusing on that to the exclusion of examining the half-full glass -- what is working -- involves a serious risk: that you won't be able to repeat your successes.

For that reason, retrospectives should incorporate appreciative inquiry -- the practice of seeking the positive core of the project's results and experience. You can include an appreciation ritual as the first step of a "Temperature Reading" (see Example Retrospective Rituals in the Appendix and Norm Kerth, Project Retrospectives: A Handbook for Team Reviews. Dorset House, 2001).

You can also use positive questions to explore topics that the team needs to address. For example, if the topic is communications, your facilitator can ask participants to recall a moment in the iteration when communication allowed them and another person to really connect and work exceptionally well together. You might consider:

- What were the circumstances?
- What made that communication compelling?

Next, the facilitator would ask participants to examine the present team and consider the various ways they communicate. Questions might
include:

- Which ways are most effective?
- Which foster a sense of connection and alignment with our project goals?
- Which enable us to work together in ways that are mutually satisfying?

Finally, the facilitator would pose "change" questions to the team to set a future direction. He or she might ask you to imagine that you arrive at work tomorrow and discover that a miracle has happened: Compelling communication has become a way of life on our project! Then, you might explore these questions:

- What is different?
- How does it feel?
- What did we do to get here?

(For more examples of positive questions you might ask in a retrospective, see Positive Questions for an Appreciative Inquiry in the Appendix.)

**Follow a Structure**

A retrospective should follow this overall structure:

1. Get ready.
2. Explore the past.
3. Understand the present.
4. Decide the future.
5. Retrospect the retrospective.

During the *readying*, the facilitator reviews the agenda and the group agrees on ground rules, or guidelines for participation. When you anticipate that emotions might be high, the facilitator must be sensitive to that and spend additional time establishing a "safe" environment, using activities that test for safety. At this time, groups may also review their guidelines for participation. One of the most critical is what Kerth calls the "Prime Directive":

Regardless of what we discover, we must understand and truly believe that everyone did the best job he or she could, given what was known at the time, his or her skills and abilities, the resources available, and the situation at hand.

The readying step is also the time for team members to make a personal commitment to being open to the outcomes of the retrospective. (For questions a facilitator and team would explore to establish retrospective
In the past step, the group examines data from the iteration or project they just completed, reviews significant events, appreciates what worked, and analyzes why things happened. During this phase, the facilitator will ask such questions as, What happened? How did you react? Why did it happen? For example, team members might look at the data around their architectural prototype and the portion of the requirements they have selected to prototype. They may recall one or more prototype review sessions with the customer and bring in e-mails that have provided feedback from customers off site. They examine their personal feelings and what happened in the team, how communications transpired, and so on.

During the present step, the group interprets what happened by asking, "What did we learn?" and "What do we do well?" They may play out scenarios to discover other ways they might have tackled the iteration, speculating on whether and how the outcomes might have been different. For example, one team speculated about how their iteration outcomes might have improved if they had obtained scenarios for their use cases from a broader customer group, rather than relying on a single customer representative. The representative readily agreed, and they discussed why she used a solo strategy. The team adjusted its approach for the next iteration, to promote greater customer buy-in and save development time. Another project team speculated on how reorganizing the team roles and involving testing earlier might have prevented some of their beta test problems.

The present step of the retrospective prepares the team to transition to future thinking and to explore the question, "Now what?" The group examines its strengths to leverage them and plans how to overcome its weaknesses for the next iteration. For example, one early iteration retrospective resulted in decisions to add more glossary terms to help customers delineate business rules, add more rigor to the nonfunctional requirements, have customers use a more rigorous decision rule process for prioritizing their use cases, and insist that management not add new people to the team in the next iteration.

The retrospective ends with, well, a retrospective of the retrospective itself. The team takes five or ten minutes to reflect on the value, productivity, and quality of their interaction in the retrospective session itself, including giving feedback to the facilitator. Incorporating a ritual of this nature into every group gathering is essential to healthy teamwork.

(For a summary list of questions to ask and answer, see Questions for Each Step of a Retrospective in the Appendix.)

Base Changes on What You Learn

Retrospectives provide a structure for developing teamwork, and they're the best way to implement sustainable team change. But unless you put what you learn from them into action, your investment in retrospectives
will not be realized. In other words, don't waste your time with retrospectives unless everyone -- including management -- is committed to taking action.

When they leave, all participants in the retrospective should have a clear understanding of what specific actions or behaviors need to be sustained or changed. That means your retrospective should deliver an action plan in the "decide the future" step. For example, during its iteration assessment/retrospective for its second Construction iteration, one project team created an action plan for correcting critical problems with system tests, reestablished how and when team communications would occur, and redefined several key roles, including that of the project sponsor.

Get management support up front for the team to make such changes. Ask managers to participate in each retrospective, to explicitly empower the team to make changes as it chooses, or by creating a set of "Recommendations to Management" for their immediate review and ratification. On one project, we had senior managers come to the last ten minutes of our iteration assessments to see the outcomes of our retrospective recommendations. This gave us immediate sponsorship for both the retrospective process and the specific actions we needed to take.

Making the Case for Retrospectives

To realistically position retrospectives as the structure for your RUP iteration assessments, it's a good idea to share their benefits and barriers with your team and management. Table 2 can help you get started.

Table 2: Analyzing Retrospectives as a Project Management Tool

<table>
<thead>
<tr>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>● Develops a team and project culture that values open and honest feedback.</td>
</tr>
<tr>
<td>● Fosters agility: The team learns to take corrective actions sooner and continually, thereby increasing the likelihood of project success.</td>
</tr>
<tr>
<td>● Exploits what we know about adult learning -- provides immediacy, relevance, and self-direction.</td>
</tr>
<tr>
<td>● Can clarify goals, roles, and communication needs on the project.</td>
</tr>
<tr>
<td>● Establishes repeatable, successful behaviors.</td>
</tr>
<tr>
<td>● Seeds an organization with people who bring healthy learning experiences from earlier projects.</td>
</tr>
<tr>
<td>● Provides opportunity to change the stories, or myths, of the organization, project, or team culture.</td>
</tr>
<tr>
<td>● Builds collective ownership of project outcomes and processes.</td>
</tr>
<tr>
<td>● Allows you to make process improvements by fixing known problems.</td>
</tr>
</tbody>
</table>
Barriers

- People will be negative or seek to blame; some will become emotionally upset.
- People will want to rush through an abbreviated iteration assessment instead of doing a full retrospective if they are in a time crunch. Retrospectives should be built into the work plan.
- Managers may fear that if the retrospective is not done well, it will breed cynicism.
- Managers and team members may have fears about airing dirty laundry, especially if that runs counter to the company culture.
- Some team members may feel retrospectives focus only on what is wrong, without sustaining what is working.
- There's a risk of losing team members who may not be willing to look inward or act as team players.
- Change is difficult to implement, and some team members may resist it.
- Reveals team, individual, and management accountability, which may be threatening if it's not part of the company culture.
- If you don't follow up on process improvement resolutions you formulated during the retrospective, then managers and team leaders will lose credibility with the team.
- Takes effort to quantify the return on investment.

Perhaps the easiest route to convincing management that retrospectives are valuable is to talk about your most recent iteration deliverable. You know the one I mean. It's the one that says if you had known then what you know now, you would have saved days or weeks. Calculate the cost of gathering the team for some three hours against the time you could have saved, and the business case becomes obvious.

End Well to Begin Well

Ironically, pausing to do in-depth retrospectives permits a project community to adapt and effectively speed up. A key intent of iteration assessments in RUP is to provide learning and self-correcting feedback. Structuring iteration assessments as retrospectives gives team members a specific way to review, play back, and think reflectively about not only how the technology is working, but also how the group process is working (or not working), and this kind of learning is essential to ongoing success. Retrospectives help the group to become more self-sufficient and productive more quickly. They also promote internal and public commitment to corrective action and build not only better software but also a healthy project community.
Appendix: Conducting a Retrospective

References


http://www.retrospectives.com

Notes


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

To accompany "Team Retrospectives — for better iterative assessment" by Ellen Gottesdiener

- Defining Ground Rules for a Retrospective
- Questions for Each Step of a Retrospective
- Retrospective Questions by RUP Project Phase
- Positive Questions for an Appreciative Inquiry
- Example Retrospective Rituals

Defining Ground Rules for a Retrospective

People in groups take on norms of behavior. Norms are standards of interaction and codes of conduct that team members adhere to when they work together. To learn, people need to adopt healthy norms. Healthy norms are the basis for conducting open and honest debriefings and are useful for any team gathering -- team meetings, requirements workshops, peer and customer reviews, and interim and end-of-project retrospectives.

Under ideal circumstances, healthy norms can emerge spontaneously during a project. Under other circumstances, though, unhealthy norms waste time and drain mental energy. Unhealthy norms include failing to share information, being late to meetings, changing baselined deliverables without notification, having secret meetings after the meeting, or coming unprepared to reviews or group meetings. These behaviors jeopardize team and customer relationships, hamper teamwork and collaboration, and can ultimately destroy your project.

One way to establish healthy norms is to explicitly define them as ground rules, or guidelines for participation. Ground rules communicate the team's commitment to work together. Without them, team members may make false assumptions about the words and behavior of others. This often results in miscommunication, poor group process, project delays, and low-
quality team deliverables. Table A-1 lists some ground rules teams have found useful.

Table A-1: Example Ground Rules for Retrospectives

<table>
<thead>
<tr>
<th>Ground Rule</th>
<th>Usage/Circumstance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tell the truth without blame or judgment.</td>
<td>Emotions are high, and people tend to regress into blaming.</td>
</tr>
<tr>
<td>Keep what's shared here confidential -- and agree unanimously on what will be discussed outside.</td>
<td>Team members fear retribution by management or are afraid their comments will be repeated out of context.</td>
</tr>
<tr>
<td>Listen, and then ask clarifying questions.</td>
<td>Team members engage in overtalk, meaning that they interrupt each other and don't fully hear other points of view.</td>
</tr>
<tr>
<td>Be on time and be prepared.</td>
<td>Team members arrive late, or do not bring information, materials, and data needed for the session.</td>
</tr>
<tr>
<td>Test inferences and assumptions.</td>
<td>Team members are too busy or focused on their own work to see the whole story or know others' experiences, leading them to erroneously interpret others' behaviors.</td>
</tr>
</tbody>
</table>

Here are some questions that can help you and your team define ground rules either before a retrospective session or at its opening, during the readying step:

- How would you describe the working atmosphere of our team?
- What are our strengths? In what ways are we particularly effective?
- To what extent are we honest and open?
- Do we ever stop and evaluate how we're doing?
- What's the best thing about this team? The worst thing?
- Are there any topics that are off-limits, or hidden agendas that you are aware of?
- What interactions are wasting our team's time right now?
- What has been tried that failed?
- How do we typically make decisions?
- Are there any questions about ground rules that I should be asking but have not yet asked?

Explicitly discussing team behavior in the context of establishing ground rules has healthy and useful consequences for the team because it
reinforces mutual responsibility for making the team work well together. Honestly evaluate how personal interactions are working. This will help team members begin to relax and have more fun together -- and that's when work really gets done.

References


Questions for Each Step of a Retrospective

Here are the phases of a retrospective (adapted from Kerth, 2001):

1. Get ready.
2. Explore the past.
3. Understand the present.
4. Decide the future.
5. Retrospect the retrospective.

Table A-2 shows questions to ask and answer for each of these steps.

Table A-2: Questions by Retrospective Step

<table>
<thead>
<tr>
<th>Phase</th>
<th>Metaquestion</th>
<th>Subquestions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Readying</td>
<td>What do you want from this session?</td>
<td>What must happen for this session to be a success? What do you want to focus on? What will you contribute to achieve your desired outcome? How safe do you feel?</td>
</tr>
<tr>
<td>Past</td>
<td>What happened?</td>
<td>What did you observe during the iteration we're retrospecting? What events do you recall? What did you see and hear? What were the significant occurrences? What surprised you? What stands out? How did you feel? How did you react?</td>
</tr>
</tbody>
</table>
Retrospective Questions by RUP Project Phase

The four phases of the Rational Unified Process, or RUP, are as follows:

1. Inception
2. Elaboration
3. Construction
4. Transition

Table A-3 shows questions to ask and answer for each of these phases.

**Table A-3: Questions by RUP Phase**

<table>
<thead>
<tr>
<th>Present</th>
<th>So what?</th>
<th>What did you learn during the iteration we’re retrospecting? Why is it important? How does it inform the work we are about to start? What puzzles you? Why? What ifý?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Future</td>
<td>Now what?</td>
<td>How do the things you’ve learned inform our continuing work? What do we want to continue? How can we leverage our strengths? What will you do differently? What will the team change? What support do we need? What will you commit to do? How will we check our progress?</td>
</tr>
<tr>
<td>Debrief</td>
<td>How did we do?</td>
<td>Was this session a valuable experience? What did we do well? What should we do more of? Less of? How did our facilitator help us? When were we interacting well? What should we remember to do in our next debrief? What should we change in our next debrief?</td>
</tr>
<tr>
<td><strong>Project Phase</strong></td>
<td><strong>Potential Retrospective Questions</strong></td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------------------</td>
<td></td>
</tr>
</tbody>
</table>
| Inception        | • How well did we delineate and communicate the product vision?  
|                  | • How did we involve stakeholders in creating our vision?  
|                  | • Could we have done it more effectively?  
|                  | • Did we involve stakeholders in a timely and appropriate manner?  
|                  | • Did we have any issues or surprises with the product vision? Why or why not?  
|                  | • Do all stakeholders understand the product vision?  
|                  | • How clear is our scope? How might we make it more clear, if necessary?  
|                  | • How complete and accurate was our business case?  
|                  | • What worked well in our vision workshops?  
|                  | • What could have been better?  
|                  | • What successes and difficulties did we have in identifying the key use cases for the product?  
|                  | • How complete was our analysis of the buy-build-reuse decision?  
|                  | • What obstacles and smooth spots did we experience in defining the candidate architecture?  
|                  | • How complete was our risk analysis?  
|                  | • Are we using risks to guide the project? Why or why not?  
| Elaboration      | • Did we select the right strategy for creating our architectural prototype? If so, how did we know? If not, how did we know?  
|                  | • What surprises and jolts did we experience during gathering requirements?  
|                  | • Do we believe our requirements are complete enough? Why or why not? Were there missing or erroneous requirements? How do we know?  
|                  | • What if we had prevented these (if any) gaps in our requirements? What would have happened?  
|                  | • How much did our requirements change, and why?  
|                  | • Which nonfunctional requirements did we
capture, and which did we not? Did we make the right choices? How do we know?

- How well did we specify "doneness" or fitness criteria for our requirements?
- How did we involve customers? In what ways did that involvement work to our benefit or work against us?
- What worked well with our requirements workshops that we should continue to do? What could we do better?
- Did we prioritize the use cases appropriately? How do we know?
- Did they provide a basis for testing?
- Was our architectural prototype sufficient to reduce risk and verify customer expectations? Is so, why? If not, why not?
- Did we select the most architecturally significant scenarios to test in our prototypes? How do we know?
- What if anything changed about our vision after we reviewed our prototypes?
- Did our development case include an appropriate number of iterations? In retrospect, would we have changed that? How?
- Did we select the best set of use cases for each iteration?
- Did we manage our requirements well? Why or why not?
- What is the quality of our architectural design? How do we know?
- How did our automated tools facilitate delivery of our Elaboration artifacts?

### Construction

- What was the quality of our code? How do we know?
- How did we do with code inspections and reviews?
- How smooth was the transition from Elaboration to Construction?
- Did we have sufficient analysis and design knowledge to begin coding?
- How well did we plan and execute our testing?
- Did we follow our test plan? If not, why not?
- How much did we use our use cases as the basis for test cases and scripts?
- What was missing and what was useful in our requirements and design artifacts for creating testing artifacts?
- Did we test in a timely manner?
- How well did we plan and manage our alpha or beta releases?
- What surprises, if any, did we have about requirements during this phase? How could those have been prevented?
- What are most of the defects about?
- What is the source of our defects?
- How did our development, testing, and configuration management tools and processes work for us? What one thing would you change?
- Did we handle configuration management well?
- Did we involve customers appropriately during Construction?
- How did the testers and developers interact?
- When did the tools get in the way? When did they help us?

<table>
<thead>
<tr>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>- How well did we make use of beta testing to stabilize our product?</td>
</tr>
<tr>
<td>- How smooth was our plan and execution of database conversions?</td>
</tr>
<tr>
<td>- Did we define and use appropriate release criteria?</td>
</tr>
<tr>
<td>- Was the customer ready for the product? How do we know?</td>
</tr>
<tr>
<td>- Were marketing and sales positioned for the roll-out?</td>
</tr>
<tr>
<td>- How did we engage them in preparing for Transition?</td>
</tr>
<tr>
<td>- What was the quality of the training?</td>
</tr>
<tr>
<td>- How effective is the product's support documentation?</td>
</tr>
<tr>
<td>- How did we involve our customers in this phase?</td>
</tr>
</tbody>
</table>
### Positive Questions for an Appreciative Inquiry

Table A-4 shows positive questions you might ask during retrospectives as part of an Appreciative Inquiry.

**Table A-4: Sample Positive Questions**

<table>
<thead>
<tr>
<th>Category</th>
<th>Questions</th>
</tr>
</thead>
</table>
| **End-of-Project**| • How realistic was our development case?  
• What are you most proud of in this project?  
• What are you most regretful or sorry about?  
• What one moment stands out the most for you?  
• Did we use our team's skills effectively?  
• How was management support?  
• How well did we manage risk throughout the project?  
• What did we learn about iterative development from this project?  
• What one recommendation would you make to other teams about their use of RUP?  
• How did we manage the trade-off between time, cost, quality, and functionality? |
| **All Phases**    | • What happened that surprised us?  
• What puzzles us?  
• How well are we communicating with each other?  
• Are we involving the right people?  
• Are our customers involved and engaged?  
• Are we collecting the right metrics?  
• What one thing do you want to remember to do again in the next iteration? |
<table>
<thead>
<tr>
<th>Topic</th>
<th>Positive Questions -- Past</th>
<th>Examination Questions -- Present</th>
<th>Change Questions -- Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compelling</td>
<td>Recall a moment in this iteration when communication allowed you and another person to really connect and work exceptionally well together. What were the circumstances? What made that communication compelling?</td>
<td>Consider the various ways we communicate on this project. Which are most effective? Which foster a sense of connection and alignment with our project goals? Which enable us to work together in ways that are mutually satisfying?</td>
<td>Imagine you've arrived at work tomorrow and we've had a miracle -- compelling communication is a way of life on our project! What is different? How does it feel? What did we do to get here?</td>
</tr>
<tr>
<td>Communications</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Getting it done</td>
<td>What has been a high point in this iteration -- in other words, when were you able to be very focused and disciplined to get a deliverable done? What good things came from that deliverable? What were you doing to maintain your discipline and drive?</td>
<td>Consider the moments when we truly &quot;kept our eyes on the prize&quot; in this project, when we adopted a &quot;just do it&quot; attitude and achieved a lot. What are we doing to foster that discipline? How are we promoting that ethic?</td>
<td>If we waved a magic wand and changed the project so that we were consistently working in a &quot;get it done&quot; mode, what would have happened? How would things have changed?</td>
</tr>
<tr>
<td>Quality moments</td>
<td>Recall a time in this iteration when you were on the receiving end of a quality moment --when a deliverable or service provided by someone on the team delighted you, as a customer. What was it like?</td>
<td>Describe how we address quality on this project. What allows us to deliver a moment like you described? How do we recognize and foster quality? What are customers' and</td>
<td>What can we learn from these quality moments? How can we apply it to the next iteration? How can we take what you know about creating quality moments and ensure we will sustain</td>
</tr>
<tr>
<td></td>
<td></td>
<td>customers' and</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Temperature Reading</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>----------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Usage</td>
<td>Any retrospective</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Procedure</td>
<td>Provide the following list as the structure for discussion, allowing team members to participate in each phase as they wish. The facilitator describes the purpose of each phase of discussion and encourages team members to participate during each phase.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. **Appreciations**: Acknowledge, support, and compliment a team member for something she did that had an impact on you or the team.

2. **Complaints with Recommendations**: No complaint can be offered without a recommendation, allowing the team to solve its own problems and to avoid whining while addressing genuine deficiencies.

3. **New Information**: Share information you may have that others don't.

4. **Puzzles**: Share things you are confused about, that don't make sense; don't try to solve them, just surface them.

5. **Hopes and Wishes**: Focusing on the future, share personal desires for yourself or the project, perhaps allowing people to understand their common needs and ending the ritual in a positive way.

References


Example Retrospective Rituals

Here are four retrospective rituals that facilitators can use at various points in the retrospective. A skilled facilitator will select rituals that match your team's specific needs and the allotted time of the retrospective.
<table>
<thead>
<tr>
<th>Time</th>
<th>Half an hour to one-and-a-half hours, depending on group size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Materials</td>
<td>Markers, flip chart paper, or index cards and a &quot;sticky wall&quot; for writing down recommendations that require further action</td>
</tr>
</tbody>
</table>
| Variations | 1. During the Appreciations phase, ask the person offering the appreciation and the person being appreciated to stand. Use the words, "Jim, I appreciate you for ý"  
2. I like to add a second step: Apologies. This is the time to allow anyone who wishes, to offer an apology or to request an apology.  
3. You can vary the sequence or eliminate one of the phases, but always start with Appreciations and end with Hopes and Wishes. For example, another possible sequence is Appreciations, Apologies, Puzzles, Complaints with Recommendations, Hope and Wishes. |

**References**  

<table>
<thead>
<tr>
<th>Name</th>
<th>Draw the Iteration [or Phase, Milestone, End-of-Project]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usage</td>
<td>Any retrospective</td>
</tr>
<tr>
<td>Procedure</td>
<td>Provide a piece of poster paper to each participant. Ask them to divide the paper into thirds by creating a horizontal across the middle, then a vertical line in the center of the top section, as shown below.</td>
</tr>
</tbody>
</table>

![Diagram of poster paper divided into thirds]

1. **Draw**: Ask everyone to draw (on top right) a visual
picture or image-- not text -- that depicts their most important learning for the iteration. Ask everyone to remain silent while they are working on their posters.

2. **Create a slogan** for the picture and write it in the top left section of the poster.

3. **List** on the bottom of the poster three to five key bullets that crystallize the key ideas in the image.

4. **Share** each poster. Ask each person (or small team that worked on a poster together) to share the three portions of the poster.

5. **Discuss** the common themes as well as differences as a group. Decide what action, if any, you want to take for the next iteration.

<table>
<thead>
<tr>
<th>Time</th>
<th>Twenty minutes to one hour, depending on group size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Materials</td>
<td>Markers, flip chart paper</td>
</tr>
</tbody>
</table>
| Variations    | 1. Have an affinity group (two or four people who want to pair up or who worked in similar roles) create a single poster.  
2. Ask the affinity groups to create posters that answer specific questions such as, "What do we want to celebrate and appreciate about the just concluded iteration?" "What was the most significant event that happened during the iteration?" "What scene would you like to depict that you wish had happened, but never did?"

<table>
<thead>
<tr>
<th>Name</th>
<th>Start, Stop, Continue, More of, Less of</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usage</td>
<td>Any retrospective; also can be used to retrospect the retrospective.</td>
</tr>
<tr>
<td>Procedure</td>
<td>1. Draw a circle on a poster, divide it into 5 sections and write the words &quot;start,&quot; &quot;stop,&quot; &quot;continue,&quot; &quot;more of,&quot; and &quot;less of&quot; around each section (see below).</td>
</tr>
</tbody>
</table>
2. Ask: "What should we start doing that perhaps we haven't done yet? What should we stop doing, that's not contributing or is getting in our way? What is working that we want to continue to do? What should we do more of? Less of?"

3. Encourage comments in any category. Clarify what category the comment belongs in, and record the comment (or summary of it) on the poster.

<table>
<thead>
<tr>
<th>Time</th>
<th>5 to 25 minutes, depending on group size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Materials</td>
<td>Markers, flip chart paper</td>
</tr>
<tr>
<td>Variation</td>
<td>For large groups, divide into subgroups to focus on one aspect of the iteration or project such as testing, requirements, customer involvement, and so forth. Encourage participants to form cross-functional subgroups (so different roles are represented in each subgroup). Each subgroup creates a poster, as depicted above, in eight minutes. Next, ask everyone to move around the room, reading the various posters silently. Finally, discuss key points and decide specific actions to take for the next iteration.</td>
</tr>
<tr>
<td>Reference</td>
<td>Developed by Ellen Gottesdiener, EBG Consulting, Inc., <a href="http://www.ebgconsulting.com">www.ebgconsulting.com</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Timeline/Key Events Storyboard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usage</td>
<td>Full project retrospective</td>
</tr>
<tr>
<td>Procedure</td>
<td>Draw a timeline on the wall, with tick marks indicating key time points in the project (e.g., months, or milestones with dates). Ask participants to write, one per card, the key events, or things that stand out in their mind, during the whole project. After each person has a stack of five to fifteen cards, have everyone simultaneously place their cards on the wall, aligning each event with its timeframe. Allow participants to review the wall and then comment on what surprised them in each other's cards, what implications the wall has for them, and what they will do differently in their next project.</td>
</tr>
<tr>
<td>Time</td>
<td>One to two hours, depending on the number of participants</td>
</tr>
<tr>
<td>Materials</td>
<td>Wall prepared with poster or butcher paper roll, sticky notes or cards, dark markers</td>
</tr>
</tbody>
</table>
Variations

1. Ask participants to form affinity groups and create their cards as a group.

2. Ask participants to write or draw their key events using color-coded cards -- for example, green (pleasurable or fun event), blue (very challenging or worrying), red (frustrating or infuriating), pink (shocking or puzzling), yellow (funny)

3. Rather than write, have participants draw a picture on the card that describes the event visually. After each card is posted on the timeline, allow the individual or team to explain which event the card is depicting.

4. After the team members silently read each others' cards on the wall, ask them to use a marker and draw a "seismograph." Have them move the marker up and down horizontally (use an open area below the cards) to indicate how they were feeling about the project as time progressed.

Reference


Notes


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
Thoughts on Functional Decomposition

by Murray Cantor
Principal Consultant
Rational Strategic Services
Organization
IBM Software Group

It is common knowledge in the Rational field organization that functional decomposition is something to avoid, and a great deal of practical experience reinforces our belief that it leads to poorly designed, low-quality systems. However, some organizations find this practice useful and seem unwilling to part from it. We have worked to understand exactly what the term functional decomposition means to different organizations and how they translate it into practice. In this article I will share what we have learned and what Rational recommends.

What Is Functional Decomposition?

I am not aware of any official definition of functional decomposition. Like many system engineering terms, it is used to describe several activities, including the following three, which are not entirely disjointed. Depending on the context, functional decomposition can refer to:

1. Adding more detail to a general requirement. "In order to do A, the system must do X, Y, and Z." For example, if your requirement is "A simple application that prints out student report cards," you might add, "In order to print out student report cards (A), the system must be able to access student records (Z)." This explains the requirement more fully.

2. Organizing requirements, particularly use cases, into packages. This is often done on larger systems. For example, a project to build a comprehensive system for a ship may have so many requirements that it makes sense to sort them into packages: navigation capabilities, performance requirements, endurance under extreme conditions, and so on. This type of organization helps the
team deal with the multitude of requirements.

3. **Determining subsystem requirements.** In some instances the team defines subsystems by the system requirements they enable. For example, a bank account management system may have one subsystem dedicated to online customer transactions and another subsystem for in-branch transactions. These subsystems maintain their separate databases.

There is little harm in the first two uses. They are simply ways to better understand and manage requirements. But the third use is a problem. *Using functional decomposition methods to derive the system architecture and set subsystem requirements puts the system at risk.*

**Architecture Should Come First**

Teams use functional decomposition to define architecture in two different ways:

- **Requirement allocation.** This involves adding details to requirements (as noted in item 1 in the above list), and then assigning requirements to subsystems (as noted in item 3 in the above list).
- **Requirement derivation.** This involves determining subsystems first, and then deriving their requirements.

Let's explore how organizations pursue each of these approaches.

**Requirement Allocation**

Often organizations perform activities 1 and then 3 from our list above: *They define subsystems by grouping decomposed requirements.* For example, if subsystem 1 may be specified by meeting requirements X, Y, and Z, the customer considers requirements X, Y, and Z as "allocated" to subsystem 1.\(^1\) This approach to finding subsystems typically leads to poor architectures. *It provides no mechanism for determining whether the subsystems can provide common, underlying services that could be reused across a range of requirements.* Typically, such common services (e.g., business-rule parsers, event handlers, etc.) do not emerge when you add detail to system requirements, but rather when you perform some flavor of object analysis. *The outcome of using functional requirement allocation to create an architecture -- rather than starting with a defined architecture -- is subsystems with duplicate code. This leads to unnecessarily large, complex systems that are costly to develop and operate, and difficult to maintain.*

Here is an actual example: One image satellite ground support system that is currently being fielded was built with a functional decomposition architecture. The system requirements included the ability to plan missions, control the satellites, and process the collected data for analysis. Accordingly, the developer built three subsystems: mission planning, command and control, and data processing. Each of these subsystems was
given to an independent team for development. During the project, each team independently discovered the need for a database with the satellite's orbital history (the satellites can, to some extent, be steered to different orbits as needed). So each team built its own separate database, using separate formats. But the information needs to be consistent for the overall system to operate correctly, and now, the effort required to maintain these three databases is excessive and could easily have been avoided had the team done some kind of object analysis, including a study of the enterprise data architecture.

Requirement Derivation

In my work, I have seen organizations perform what they refer to as "functional decomposition" by determining their subsystems first and then deriving requirements. Usually, the team decides up front that they will use a certain type of architecture (e.g., standard three-tier database application). Sometimes they are also carrying a logical decomposition around in their heads but not recording it in a shared document. Quite literally, they "have a design in mind." In any case, these teams decompose the requirements to align with their instincts about what the system architecture should be; then, they allocate those decomposed requirements to the intended architectural elements. Sometimes they follow this with a synthesis step to combine similar requirements.

Although teams often refer to this practice as "functional decomposition," it is really a form of requirement derivation, and may not result in the same problems as functional allocation. The main difficulty I have seen with this approach is that the team does not document the architecture explicitly. Instead, they proceed with their work based on the architecture that is implied by the way requirements are allocated. But unless the architecture is very simple and the team very small, this lack of documentation for the architecture leads to poor understanding of the system and therefore to many missteps along the development path. There is great value in explicitly documenting the logical architecture in UML.

An explicit, well-documented architecture also makes requirement derivation a repeatable practice by introducing limits and discipline to the process. For example, consider an automobile differential, which allows the driving wheels to run at different speeds when turning, so the automobile can maintain traction while in a curve and thereby hold the road. The differential's requirements are based on (a) supporting the basic requirements for an automobile, such as the need to maintain traction on curves, to be of a certain overall weight and volume, to be easy to maintain, and so on; (b) an architectural decision to design the automobile with a drive train consisting of a single engine connected to a drive shaft, which is connected to a differential that is connected to two rear axles, which in turn are connected to the rear wheel hubs. Since there are other possible architectural solutions to the basic set of requirements for an automobile -- solutions that do not require a differential -- the differential's requirements must be derived from the chosen architecture and cannot be discovered merely by doing a functional allocation.
Some Final Thoughts

Decades of experience have shown that good system architectures -- those that are maintainable, extendable, cost effective, and so on -- result from finding subsystems with internal services that:

- Are reusable.
- Enable those subsystems to collaborate in order to meet system requirements.

Generally, each subsystem can play a role in meeting more than one system requirement. This is the essence of logical decomposition. You can find a workflow for deriving requirements for subsystems in Unified Modeling Language–based design models in the whitepaper "Rational Unified Process ® for Systems Engineering, TP 165" and in the RUP plug-in for RUP SE. As in the use-case flowdown method, these requirements are "derived" from studying collaborations in the design models. That means the subsystem requirements will have a many-to-many relationship with the system requirements.

To help Rational customers understand this concept, I explain that it is optimal for subsystems to provide services in an "m-n relationship." That is, each subsystem requirement may be derived from many system requirements, and each system requirement can result in many subsystem requirements. And subsystem requirements are related to architectural decisions, as we saw in the automobile example above. In fact, they can even think about this approach to subsystems as a form of functional decomposition -- with derived subsystem requirements. To quote Nelson Mandela, "If you want to make peace with your enemy, you have to work with your enemy. Then he becomes your partner."

Notes

1 A more detailed description of this process and its shortfalls can be found in Practical Software Requirements by Benjamin Kovitz (Manning, 1999).

2 Available at http://www.rational.net/ (authorization required).

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
Progressive Acquisition and the RUP
Part V: Contracting Activities

by R. Max Wideman
Project Management Consultant
AEW Services

In Part I of this series we discussed bridging the gap between the expectations of traditional procurement specialists and the realistic needs of software developers. In Part II we showed how traditional contracting could be tailored to meet these needs. Part III and Part IV covered contracting basics and the various considerations that govern contract formulation.

Now, in Part V, the final installment in this series, we will take a look at the activities involved in contracting and how they fit into the Rational Unified Process,® or RUP,® from the acquirer's perspective. In the Appendix, we will provide a helpful glossary for preparing and administering progressive software acquisition contracts.

Progressive Acquisition Workflow

As you will recall, Part II described a two-level contracting approach consisting of a Head Contract and a series of Contract Work Orders (CWOs). Now we will look at some aspects of the primary contracting activities involved in developing the Head Contract and pursuing the CWOs. For readers who are familiar with RUP terminology, we can express the workflow, activities, and artifacts for the Head Contract and first CWO as shown in Figure 1.
For subsequent CWOs and the remaining project life span, we can express these things in a similar view, but we would modify the figure to look like the one in Figure 2. Note that most of the work associated with the Head Contract has been eliminated, and this workflow focuses specifically on the next CWO.
However, our real goal is to help technical professionals talk to procurement personnel, so let us now discuss these aspects of contracting in the traditional terms of the contracting industry rather than in RUP terms. Indeed, some terms are used quite differently in these different contexts.

Throughout our discussion, we will view these aspects of contracting from the acquirer's perspective.

**Specifying the Work**

In general contracting practice, the information that defines or specifies the work of the contract can take one of several forms:

- Functional Specification
- Performance Specification
- Technical Specification
- Some combination of all three types of specifications -- provided that the instructions are not in conflict.

Please note that we are using these terms as they are used in the contracting industry; the terms *functional*, *performance*, and *technical* have a different meaning within the context of the Rational Unified Process. To avoid confusion, and to understand the general application of these three terms, here is a brief explanation of each.

**Functional Specification.** This is a document that describes how users will employ the product and the functional capabilities it must have. It stimulates suppliers to propose creative responses, possibly at lower cost. The supplier has the risk of performance, but as an acquirer, you may find it difficult to establish or enforce acceptance criteria.

**Performance Specification.** This document specifies measurable capabilities the product must achieve in terms of operation (sometimes referred to as "form, fit, and function"). As an acquirer, you can spell out acceptance criteria as objective tests the product must satisfy; otherwise you will reject the product. The supplier carries the risk of performance.

**Technical Specification.** This document describes in detail the elements and commercial products contained within the final product for delivery -- in other words, the programming language, platform, configuration or architecture, methodology if applicable, and so forth. It should include charts, diagrams, and tables that specify how to construct the product and prescribe the tests and observations that you, as the acquirer, intend to conduct to verify or ensure compliance. In this case, you -- as the acquirer -- take the risk of performance. However, if you demand a fixed price contract, then the supplier assumes the cost risk.

On the face of it, this may seem like a winning solution and, indeed, in the
absence of contrary instructions it is the type most often chosen by procurement departments. Unfortunately, in the real world of software development, it is practically impossible to develop technical specifications with sufficient accuracy to enable a controversy-free basis for a fixed price contract. This is true even if you assume that the acquirers know exactly what they want, are technically knowledgeable, and are proficient in the RUP. For this reason, fixed-price contracts are also the type that lead most often to serious disagreements, unsettled claims, expensive legal actions -- and less than successful projects!

**Selecting or Pre-Qualifying Suppliers**

From the acquirer's perspective, it is always a good idea to seek more than one proposal because there are several benefits:

- Serious proposers will invest more effort into understanding your requirements and produce a better response.
- It is a useful way of collecting alternative ideas for solving the particular software problem (although this is frowned upon by the supplier community, which views it as "free" consulting).
- Competition sharpens both the mind and the pencil.

The criteria for selecting suppliers typically include:

- Technical capability relative to the required product.
- Management capability.
- Corporate financial strength.
- Previous positive experience with the supplier.

You should carefully avoid and strongly discourage less desirable selection criteria, especially biases based on proximity, nationality, ethnicity, or political persuasion.

Under certain conditions, you may decide to seek bids from one supplier only:

- There are few qualified suppliers for the work contemplated.
- Time pressures preclude a competitive proposal process.
- Quality rather than lowest cost is the overriding consideration.
- Your company has a long-standing and comfortable relationship with the chosen supplier.
- The work is highly confidential and can be assigned only to a trusted or legally bound supplier.

**Making the Solicitation**
The means by which an organization secures a bid for a software development project is typically a solicitation called a Request for Quotation, Request for Proposal, or Request for Tender (or Bid). Although the terms are often used interchangeably, the actual form and content for these solicitations varies widely across the contracting community. We can distinguish among them as follows.

**Request for Quotation (RFQ).** Use this to request a price or prices for standard products that can be purchased in varying quantities using a purchase order -- software licenses, for example. The purchase order issued to the supplier represents the offer, and it becomes the contract when the supplier accepts it.

**Request for Proposal (RFP).** Use this to solicit proposals for both price and technical approach, based on a performance specification included in the RFP. Then, base your selection of a supplier on pre-set evaluation criteria, and negotiate the work content, price, and terms. When you review the various suppliers' proposed technical solutions, it's important to ensure they have not introduced unwanted modifications to your intended technical approach. It is equally important to ensure that the technical approach they offer is acceptable for your particular working environment. Consequently, it may be necessary to negotiate the technical content and language in the final contract.

**Request for Tender (RFT), Request for Bid (RFB).** If you must insist on a firm price, rather than adopting the progressive acquisition approach we recommend, then make sure you have a healthy management reserve to cover legitimate changes! Request a firm-price tender from competitive bidders based on a technical specification. This form of solicitation demands the most thorough legal and technical specifications from both the acquirer and suppliers. Include the exact form of contract with the RFT or RFB; the offer (tender or bid) you accept will then become part of the contract. In evaluating responses, make sure that the respondents meet all the criteria you described in the request. Those that do not meet the published criteria should be rejected; otherwise, you will undermine the validity of your process and tarnish your reputation.

In the case of both RFPs and RFTs, you should plan on holding a "bidders' conference" to answer potential suppliers' questions and thus ensure everyone has a clear understanding of the requirements. If this raises issues that you overlooked in the original request, you can issue an "addendum" to the original requirements. In any case, you should formally document and distribute your answers so that they become part of the solicitation documents.

**Evaluating Submissions**

As the acquirer, your objective is to negotiate or select the solicitation response that best serves your requirements. So, in addition to criteria relative to the supplier's capability, you should consider applying the following criteria:

- Product technical performance relative to your functionality "musts"
Supplier's schedule for performing the work.

Cost, including both product lifecycle costs (i.e., potential in-service maintenance service costs), as well as the initial project acquisition costs.

Demonstration of ability to provide a quality product.

Naturally, suppliers will be more responsive if they perceive your evaluation process as fair; including your evaluation criteria as part of the solicitation documentation helps to convey this message. The key is to conduct an open, honest, and logical appraisal.

**Negotiating the Contract**

Once you receive and evaluate proposals in response to a solicitation, you select the highest-ranking supplier or suppliers to develop a contract that best fulfills your needs. This is not an easy process; and especially for large contracts, you must conduct this process with the utmost ethical integrity. The following measures can help ensure high standards:

- Plan and prepare carefully for negotiations; get internal consensus on what you want as an organization.
- Conduct meetings formally and stick to an agenda.
- Follow up promptly and translate terms of agreement into writing as soon as possible.
- Conduct an in-house, post-negotiation review to capture lessons learned.

At the end this process you should arrive at a binding agreement that obligates the supplier to produce and deliver the product and you, the acquirer, to pay for it.

**Administering the Contract and Controlling the Supplier's Work**

In all but the simplest of agreements, the acquirer has certain technical responsibilities with respect to the contract. How well you perform these responsibilities will have a significant effect on the supplier's performance. Your overriding aim should be to maximize the likelihood of meeting all the contract objectives on both sides. Therefore, you should pay close attention to fulfilling the following responsibilities.

- Provide technical clarifications quickly.
- Respond as soon as possible to supplier requests for information, or reviews and approvals, relating to schematics, architecture, interface configurations, use of subcontractors, and so forth.
- Coordinate, or ensure coordination, among suppliers if the project
Promptly exercise quality control acceptance, waiver, or rejection; if you reject something, quickly request correction of defects and certify progress when the correction is complete.

Resolve disputes and/or claims early by forewarning the supplier of potential difficulties, initiating fact-finding activities for potential or registered disputes, and initiating a change order process if appropriate.

Show interest by monitoring and tracking the supplier's progress and expediting roadblocks.

Process changes expeditiously.

Abide by the terms of the contract.

Above all, pay progress payments promptly. Nothing discourages a supplier more than leaning on his 30-, 60-, or 90-day line of credit!

Contract Control

As we noted earlier, the type of contract you select largely determines the degree of control you have over the supplier's work; the firmer the price, the lower the acquirer's level of control over contract performance. Nevertheless, even in fixed-price situations you still have levers, short of extreme actions such as termination and lawsuits. These levers are:

- You can tie payments to acceptance or rejection of results; in other words, you can agree to pay upon acceptance but hold back payment if there are performance deficiencies. Your position will be stronger if you specify staged progress payments in the contract.

- You can submit change orders to the supplier for adding or subtracting work. Preferably, you and the supplier should agree on the validity of each change order as well as on a corresponding adjustment to the cost.

- You can issue a stop work order (under unilateral and exceptional conditions) provided that you justify the cause.

Of course, you must make sure to write these control procedures into the contract. You cannot expect to unilaterally assume these privileges after the contract is signed -- at least not without risk of legal action by the supplier!

Common Problems and Issues

Knowing some of the common problems and issues that arise in contract situations can help you avoid them. Here are four things to watch out for.

- **Flawed technical requirements, specifications, or descriptions.** If the product you specify turns out to be a practical impossibility (although the supplier will have a difficult time proving this), all or part of the contract may be invalidated.
Formal registration of disputes and/or the application of extreme remedies. These typically cripple any work in progress. Therefore, make every effort to ensure that the work can continue while disagreements are being resolved.

Questions over warranty. Most contracts include an express warranty clause. However, if the supplier is aware of the acquirer's purpose and use for the product, then there is also an implied warranty. Disputes arise when the product does not meet the acquirer's needs, and the supplier refuses to honor that implied warranty.

Inappropriate change processing. Changes can be a source of dispute if:
  - Either the supplier or acquirer makes unilateral changes in breach of the contract or does not follow the change process agreed upon in the contract.
  - The acquirer (or one of the acquirer's agents), as a result of action or inaction, creates the need for a change that forces the supplier to perform in a manner that is more costly or difficult than that prescribed or contemplated in the original contract.

Terminating the Contract

Under normal circumstances, the acquirer closes the contract by issuing a final acceptance -- and paying the bill -- once the supplier has performed all the specified work.

Under certain circumstances, however, you might want to or have to terminate the contract:

- By mutual agreement -- because you want to replace the contract with another in light of new information, corporate changes, or a condition specified in the original contract.
- Because of mutual frustration -- if the work proves impossible to perform or because of radically altered circumstances.
- Because of legal requirements -- if the supplier files for bankruptcy, for example.
- For breach of an essential contractual term -- a material breach on the part of one party may discharge the other from any further obligations.

Understand Progressive Acquisition, but Work with Experts

In this five-part series on progressive acquisition, we have given you a lot to digest, but do not be intimidated. If you have a knowledgeable and sympathetic procurement and legal staff, they will take care of most of it. Your job is, first, to be aware of the advantages that progressive
acquisition offers and how it works so that you can advocate for this approach, whether you are on the supply or the acquisition side. In addition, if you are aware of the various contractual requirements and issues that we have identified, you can ensure that your solicitation and contract documentation are complete and provide maximum risk protection for you, whether you are acting as an acquirer or a supplier. On the acquisition side, that includes specifying how you want the workflow to unfold and how you want it administered.

The basic concept to get across to your procurement and legal staff is the idea of establishing a Head Contract that encompasses the essence of the project venture and then creating a series of CWOs. If you issue the first CWO along with the Head Contract, it can lay a foundation with detailed requirements and possibly business modeling. Then, based on the results of this work, you can issue subsequent CWOs for Elaboration, Construction, and Transition with a much higher degree of certainty. This will be true for both technical content -- that is, functionality -- as well as cost and schedule parameters. In short, this approach yields a higher degree of control, certainty of outcome, and likelihood of product success.

**Appendix: Glossary of Terms for Progressive Acquisition**

As a project manager, you have probably found that arriving at common definitions for the terminology used in software development projects can be problematic. It becomes even more so when people of different backgrounds are involved. And the challenge is even greater when you are introducing a new discipline, such as progressive acquisition, to contract procurement and legal professionals. Here are some definitions that you can use as a starting point for promoting a common understanding of progressive acquisition.

**Acquirer** -- Also known as the purchaser or buyer. An organization that acquires or procures a system, software, or software service from a supplier.

**Acquisition** -- The process of obtaining a system, software product, or software service through contract. Also known as buying, contracting, procurement, or purchasing. Note that acquisition is not restricted to obtaining standard, "off-the-shelf" software.

**Agreement, Legal Agreement** -- A legal document setting out the terms of a contract between two parties.

**Bid** -- See **Offer**.

**Change Order (CO)** -- A formal document issued by the organization's contracting officer, directing the contractor to make a change that is specified in the governing contract as permissible, even without the contractor's consent.

**Contemplated Contract Work Order** -- A proposed **Contract Work Order (CWO)** issued by the acquirer with a view to initiating and
negotiating, if necessary, the technological content of a specific project phase and obtaining the supplier's price for that part of the work.

**Contract** -- A document that represents a binding agreement between two parties, enforceable by law, for the supply of a software service or the development, production, operation, and/or maintenance of a software product. In contract law, the most common terms used to designate the parties to a contract are "buyer" and "seller."

**Contract Work Order (CWO)** -- A short-term contract defining specific deliverables. Multiple CWOs are specifically linked to a longer-term **Head Contract**.

**Contracting Officer** -- An individual with an official position within the acquiring organization and with the organizational authority to conduct the acquisition process.

**Contractor** -- See **Supplier**.

**COTS Software** -- Customizable, off-the-shelf software.

**Developer Organization** -- An organization that performs software development activities, including requirements analysis, design, testing through to acceptance, during the software life cycle process.

**Evaluation** -- A systematic determination of the extent to which a product or part of a product meets its specific criteria.

**Evaluator** -- A person who conducts an evaluation.

**Head Contract** -- A longer-term contract that contains the basic terms of the supplier-acquirer relationship, such as payment cycles. The Head Contract is formulated before work begins on the project and is then linked to a number of **Contract Work Orders (CWOs)** specifying a series of deliverables to the client. These are formulated in conjunction with specific milestones, as the work progresses.

**Increment** -- A simple term used to refer to the deliverables covered by a specific **Contract Work Order (CWO)**. Each increment evolves the existing functionality as necessary and adds new value to the product under development.

**Monitoring** -- The process of examining the status of activities that are the responsibility of a performing organization (supplier) to determine progress toward product delivery.

**Offer** -- Also called a *tender, bid, proposal, or quotation*. A document prepared by a supplier that responds to a solicitation. If accepted, the offer binds the supplier to the terms of the resulting contract. However, *quotation* sometimes refers to an approximate estimate rather than a firm price. Note that an acquirer can also make an offer in response to a supplier's offer; this is usually termed a *counter offer*. 
**Procurement** -- This refers to all stages involved in the process of acquiring supplies or services, beginning with the determination of a need for supplies or services and ending with contract completion or closeout.

**Progressive Acquisition (PA)** -- A strategy to acquire a large and complex software system, the specifications for which will predictably change during the development lifecycle. The objective of PA is to minimize many of the risks for both parties associated with such projects. The final system is realized through a series of operational increments that advance the system capability.

**Project Director** -- Also known as the project owner, owner's representative, or project sponsor. The individual in the acquiring organization who has authority and responsibility for directing the project -- and the business process that the new product will support. The project director provides overall direction to the supplier's project manager within the terms of the governing contract.

**Project Manager** -- The person responsible for the project management process -- in other words, for executing the project work day by day. In practice this is not a discrete responsibility, because often both the acquirer and the supplier have their own project managers. Ideally, the person responsible for the project within the acquirer's organization should have the title *project director*, while the equivalent position in the supplier's organization should be called *project manager*.

**Proposal** -- See **Offer**.

**Purchase Order (PO)** -- A formal document issued by the acquirer to a supplier that commits the organization to the purchase according to specified terms and conditions. A PO becomes a contract when accepted by the supplier but typically contains much less legal language than a formal contract. Generally, a PO is used only for the acquisition of "off-the-shelf" products, or if the work to be done is of a minor nature; most companies place limits on the cost of work that can be performed under a PO. A PO is different from a change order, although it may be modified to double as a change order.

**Purchaser** -- See **Acquirer**.

**Quotation** -- See **Offer**.

**Request for Information (RFI)** -- A formal document prepared by the acquirer and presented in the marketplace, typically to elicit "expressions of interest," from suppliers that have the capacity, capability, and availability to undertake and bid on, or tender for, work described in the request. The work description is typically a preliminary version of an eventual request for proposal.

**Request for Proposal (RFP)** -- A formal solicitation prepared by the acquirer to elicit proposals from potential suppliers. An RFP generally consists of a solicitation letter, instructions to suppliers, evaluation criteria, a statement of requirements, and/or a product performance specification. Selection of the supporting technology for the product may be left to the
Supplier.

**Request for Quotation (RFQ)** -- A formal solicitation similar to an RFP but commonly used for simplified acquisition procedures. An RFQ is appropriate for acquiring a standard "off-the-shelf" item that may be purchased in varying quantities (e.g., software licenses) and requires little or no customization. May also be used if the work to be done is of a minor nature. The corresponding contract document is a **Purchase Order (PO)**.

**Request for Tender, Request for Bid (RFT, RFB)** -- Also known as an invitation to tender, or tender invitation. A formal solicitation prepared by the acquirer to obtain firm prices from potential product suppliers for a product based on a technical specification. An RFT or RFB typically consists of a letter of invitation, instructions to bidders (suppliers), a specification about the form of contract (the accepted offer becomes part of the contract), a system specification and/or statement of work, and possibly other technical documentation. This form of solicitation demands the most thorough legal and technical specifications from both acquirer and supplier. The supplier is selected on the basis of best price.

**Scope, Contract Scope** -- A document used by the acquirer to describe the product to be delivered under the contract, along with the product's attributes. These may include such considerations as functionality, performance, quality, operability, maintainability, and so on.

**Seller** -- See **Supplier**.

**Solicitation** -- The process of issuing a document requesting submission of an offer, quote, or information. **Solicitation** may also refer to the document itself.

**Statement of Work (SOW)** -- A document used by the acquirer to describe and specify the tasks to be performed by the supplier under the terms of the contract.

**Subcontract** -- A contract between the prime contractor and another supplier. Whereas a *subcontract* is also a *contract* in the legal sense, a contract is not a subcontract unless there is a "main" or **Head Contract** in place. That is, unless there is a contract between the original acquirer and a supplier. Even though an acquirer may parcel out separate pieces of work in a project, the use of the term *subcontract* in this context is incorrect.

**Supplemental Agreement** -- A document that sets out terms to modify an existing contract, reached by mutual agreement between the acquirer and supplier. Generally used for changes that fall outside the contract's change provisions. The Supplemental Agreement must cover both the work to be done and the price to be paid.

**Supplier** -- Also known as the *contractor* or *seller*. Any organization, including contractors and consultants, that supplies services or products to the acquirer.
**Tender** -- See Offer.

**User (End User)** -- The individual or group who will use the system when it is deployed in the intended environment. Note that users do not necessarily have the same interests as official acquirers, and their specific areas of interest -- or indeed their involvement -- are not always spelled out in the contract. Failure to involve end-users may lead to poor acceptance of the product and consequently a less-than-successful project.

---

*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](#)
PRINCE2 and RUP: Loose Coupling Works Best

by Russell Norlund
Principal Consultant
Ortia House, UK

This article describes how to combine the PRINCE2® methodology with the Rational Unified Process®, or RUP®, to achieve a "best of both worlds" solution. It proposes a loosely coupled approach that minimizes the time, effort, and cost involved in integrating the two methods.

PRINCE2 (Projects IN Controlled Environments) provides a non-proprietary best practice approach for project management. The PRINCE2 method was developed by the CCTA (Central Computer and Telecommunications Agency), now part of the Office of Government Commerce (OGC), as a UK government standard for IT project management. It is being used successfully in both the public and private sectors.

The Rational Unified Process, or RUP, is a software engineering process developed by Rational Software, which is now part of IBM. RUP is founded on a set of software best practices (develop iteratively, manage requirements, use component architecture, model visually, continuously verify quality, and manage change) that in combination eliminate major development problems and enable teams to deliver better software.

Software development is not the primary focus for many of the projects run by large organizations. Figure 1 shows a project structure for a company working on improving customer service capability. This involves: feasibility work, increasing capacity by building new -- or improving current -- call center facilities; and upgrading telecom and IT infrastructure. Software development is only a small part of a much bigger...
piece of work.

Organizations working on such complex projects often adopt a tiered approach to project management, with the upper tier responsible for the management and coordination of a disparate collection of suppliers, and a lower tier responsible for the creation of specialist products such as software. Typically, organizations manage the upper tier activities with a standard project management method such as PRINCE2.

![Figure 1: Tiered Project Management Structure for a Complex Project](image)

However, many organizations that have adopted PRINCE2 perceive great value in RUP’s iterative, risk-driven approach to software development, which enables them to exert greater control over software development projects through early discovery of risk, improved scope management, and greater visibility of progress and quality.

Although, as Figure 2 shows, PRINCE2 and RUP are mainly complementary -- because RUP focuses primarily on specialist activities such as business modeling, design, and architecture -- there is some overlap between the two processes with respect to:

- Project Organization and structure
- Documents
- Planning components (e.g., risk management, problem management, quality management, etc.)

In suggesting ways to resolve this overlap when integrating PRINCE2 and RUP, the suggestions in this article will focus on three primary areas:

- Process model alignment and integration: the impact of RUP on the PRINCE2 process model.
- Documentation: resolving discrepancies between RUP and PRINCE2 documentation.
Organization: the impact of RUP on the PRINCE2 organization structure.

The in-depth tailoring of specific activities, documents, and roles is outside the scope of this article.

Figure 2: PRINCE2 and RUP Are Mostly Complementary

The next section examines adoption strategies for integrating PRINCE2 and RUP.

Adoption Strategy for PRINCE2 and RUP

At Ortia, we have defined the following principles for PRINCE2/ RUP integrations:

- **Keep it simple.** The integrated process should be understandable to both the PRINCE2 and RUP communities.
- **Make it "do-able."** The process should be easy to implement in a short time frame with the minimum amount of effort.
- **Maintain separation of concerns.** RUP focuses on the development of software products, and PRINCE2 focuses on the non-technical aspects of the project; the integration should not muddy these distinctions.
- **Maintain flexibility.** The integrated process should permit additional methods to be used with PRINCE2 and RUP.
- **Create synergy.** The integration should result in a "best of both worlds" solution that emphasizes each method's strengths.

These principles explain why we recommend a loosely coupled integration strategy, as shown in Table 1.
As you can see, the loosely coupled strategy aligns with all the principles. We have seen this approach work successfully in real world projects. And because it supports the integration of additional methods, it is a useful approach to adopt on complex projects involving a number of third parties using disparate methods.

**Process Model Integration**

This section outlines the alignment between the PRINCE2 process model and RUP.

**The PRINCE2 Process Model**

Figure 3 shows the PRINCE2 Process Model, including the path and sequence through various project activities to produce a final product.
Table 2 provides a brief overview of these activities.

Table 2: Overview of Project Activities Defined in PRINCE2

<table>
<thead>
<tr>
<th>Activity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directing a project</td>
<td>This is concerned with the high-level governance of the project; i.e., it runs for the whole life of the project.</td>
</tr>
<tr>
<td>Starting up a project</td>
<td>This encompasses the pre-project work that includes identifying a project and ensuring that project prerequisites are in place (i.e., organization, people, documentation etc.).</td>
</tr>
<tr>
<td>Initiating a project</td>
<td>This process covers the initial planning and definition of the project in terms of, for example, business case, costs, and so on.</td>
</tr>
<tr>
<td>Planning</td>
<td>This activity outlines how plans will be drawn up at various stages of the project.</td>
</tr>
<tr>
<td>Managing stage boundaries</td>
<td>This provides senior management with key decision points to assess the continued viability of the project against the business case.</td>
</tr>
<tr>
<td>Controlling a stage</td>
<td>This is concerned with the day-to-day management of the project.</td>
</tr>
<tr>
<td>Managing product delivery</td>
<td>This activity is concerned with the allocation of work to the project team, ensuring that the work is carried out, and that it meets the defined quality criteria.</td>
</tr>
</tbody>
</table>
The next section will explain what *stages* mean in the PRINCE2 context.

**PRINCE2 Stages**

In PRINCE2, a stage is defined as:

- A division of the project for management purposes.
- A collection of activities and products that can be managed as a unit.

A project board grants formal approval for the project to continue, one stage at a time.

In PRINCE2 there are two types of stages:

PRINCE2 recognizes that all the planning cannot be done up front and that it is sensible to plan in detail only for a limited planning horizon. So, just as in RUP, only the current stage is specified in detail; detailed planning for the following stage is done just before the end of the current stage. The project board exerts control over the project by approving all major plans, conducting reviews at each management stage, signing off the completion of each stage and authorizing the start of the next stage, and acting as the final escalation point for major risks, issues and changes.

The next section will show how PRINCE2 management stages can be aligned with RUP phases.

**Aligning PRINCE2 Stages with RUP Phases**

The PRINCE2 process model allows for great flexibility in establishing stages. In fact, PRINCE2 mandates only the initiation stage, which covers initial planning and definition for the project; other stages are defined at the discretion of the organization.

Figure 4 shows an alignment of RUP phases and PRINCE2 stages that emphasizes the following points:

- PRINCE2 planning covers the entire lifecycle. Incorporating RUP, an iterative process, means some of the PRINCE2 planning becomes more iterative as well -- that is, as each software iteration is completed, project managers should review, revise, and detail appropriate PRINCE planning documents accordingly.
- Overall project governance (i.e., high level objectives, business case approval, funding, resource allocation, go/no go decisions) are
undertaken by the project board and not by project management.

- PRINCE2 project management (at a high level) directs and monitors the software development effort.
- RUP planning manages the software development aspects of the project at a day-to-day team level.
- RUP phases roughly equate to PRINCE2 management stages.

Note that a number of other teams may be involved, who may or may not be using RUP.

![Figure 4: Alignment of RUP Phases and PRINCE2 Stages](image)

The idea of the alignment is to assign high-level governance and management of the project to PRINCE2 and the management of iterative development and incremental delivery to RUP.

**Detailed Process Model Integration: The PRINCE2 Perspective**

This section takes a more detailed look at how to integrate the PRINCE2 process model and RUP.

**Integrating Stages and Iterations**

The PRINCE2 activity Managing Product Delivery (MPD), shown in Figure 5, allows a controlled break between the project manager and the creation/provision of a product by (internal/external) suppliers. The supplier may not be using PRINCE2, so MPD defines the required interface between the team manager and the PRINCE2 method.
Figure 5: Interfacing Managing Product Delivery (MPD) in PRINCE2 with a RUP Project

By treating the RUP project as a supplier to the PRINCE project, the acquiring organization can use the MPD interface to manage aspects of the RUP project. The main advantages of this approach are:

- It minimizes the tailoring effort required for the two methods.
- It enables iterative planning to take place in the context of overall control, as we discussed above.
- The loose coupling between PRINCE2 and RUP permits the integration of additional methodologies.

The PRINCE2 project manager approves work by authorizing work packages. A work package is a set of information about one or more required products assembled by the PRINCE2 project manager and is used to delegate responsibility to a team manager. Although the content of a work package may vary greatly according to the formality of the relationship between the project manager and the team manager, it should cover:

- Date
- Team or person authorized
- Work package description
- Product description(S)
- Techniques/processes/procedures to be used
- Interfaces to be satisfied by the work
- Interfaces to be maintained during the work (risks, issues, change control, quality)
- Quality checking method to be used
- Stage plan extract
- Joint agreements on effort, cost, start and end dates
- Sign-off requirements
- Work return arrangements
- How completion is to be advised
- Any constraints to be observed
- Independent quality checking arrangements
- Reporting arrangements (progress/tracking)

The team manager is empowered to delegate and schedule the team-level work at his/her discretion. Depending on individual circumstance, the work package can range from a verbal instruction to a full-blown document.

Managing Product Delivery consists of the following three processes:

- **Accepting a work package.** This means the project manager and the team manager reach agreement on what is to be delivered.
- **Executing a work package.** Creation and management of the work is delegated to the team.
- **Delivering a work package.** The team manager notifies the project manager that the work package is complete.

The following sections explore these processes in more detail.

**Accepting a Work Package.** Figure 6 shows the process by which the work package is allocated to a RUP team manager.

![Figure 6: Accepting a Work Package](source: Managing Successful Projects with PRINCE2, UK Office for Government Commerce (OGC), 2002)

The RUP team manager has to agree on the work package with the project
manager. This involves producing and agreeing on a team plan that shows that the work package can be completed within the specified constraints for schedule, cost, and quality.

The work package is focused at about the same level as a RUP phase: It outlines the major goals and deliverables to be achieved. The software development plan that the RUP team manager returns will constitute the team plan.

**Executing a Work Package.** Figure 7 shows the process of executing a work package.

![Figure 7: Executing a Work Package](source: Managing Successful Projects with PRINCE2, UK Office for Government Commerce (OGC), 2002)

The actual execution of the RUP disciplines takes place in the Executing a Work Package process. The RUP team manager is assigned responsibility for managing, co-ordinating, tracking, and reporting on the software development work. Under normal circumstances, the PRINCE project manager is not concerned with the detailed activities occurring in this black box, but only with the information and products coming out of it. If progress is not being made or the development team is having difficulties, then the project manager and team manager need to draw up and agree upon an exception plan. The information the project manager requires to monitor progress for the work package is provided by the RUP team manager. This means the team manager must:

- Capture and record the effort expended.
- Determine the status of each product in the work package -- use cases, for example.
- Monitor and control the risks associated with the work package.
- Evaluate the amount of work remaining.
- Feed the progress and status information to the PRINCE project manager in checkpoint reports.
- Ensure that the required quality checking procedures are carried
Delivering a Work Package. Figure 8 shows the process of delivering a work package.

![Figure 8: Delivering a Work Package](image)

The RUP team manager must notify the PRINCE project manager when the work package is complete.

Typically, this involves three elements:

- Ensure that the products have been quality checked.
- Hand over the completed products; return of the approved products will be handled by the project's Configuration Management System.
- Notify the PRINCE project manager that the work package is complete.

The level of formality for this process depends upon:

- The size and complexity of the project.
- Whether the software development has been outsourced (which demands a higher formality level than for "in house" development).

Integrating PRINCE2 and RUP Documents

Figure 9 shows that there are different levels of plans and documentation at the project and team levels. In general the lower the level of the plan the greater the detail contained within the plan.
Figure 9: Levels of Planning and Documentation for an Integrated Process

For PRINCE2/RUP integration, the general strategy with regard to documentation is that:

- PRINCE2 provides the planning documentation at the project, stage, and exception levels.
- RUP provides the planning documentation at the team level to support the development of software products.

Exception plans are jointly agreed upon by the project manager and the RUP team manager.

Project Level Documents

As Table 3 indicates, most PRINCE2 planning documentation can be retained for planning the overall project. However, the RUP Configuration Management Plan document is more technically detailed than the PRINCE2 equivalent and therefore can replace it. It is also important to note that RUP is an iterative process, so integrating PRINCE2 and RUP means that many planning documents become, in a sense, works in progress throughout the project lifecycle. The PRINCE2 project team may need to adjust plans in response to the state of the software under development, and some team documents, such as software iteration plans (see Table 4), are not developed in detail until shortly before the iteration is scheduled to begin.

Table 3: Project Level Documentation for PRINCE2 and RUP Integration
<table>
<thead>
<tr>
<th>Start Up</th>
<th>Initiation (Inception)</th>
<th>Stages (Elaboration, Construction, Transition)</th>
<th>Closure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Mandate (P)</td>
<td>Project Initiation Document (P)</td>
<td>Work Package (P)</td>
<td>Project Notification Closure (P)</td>
</tr>
<tr>
<td>Project Brief (P)</td>
<td>Quality Plan (P)</td>
<td>Checkpoint Report (P)</td>
<td>Post Project Review (P)</td>
</tr>
<tr>
<td>Risk Log (P)</td>
<td>Business Case (P)</td>
<td>Highlight Report (P)</td>
<td>Follow-On-Actions Recommendations (P)</td>
</tr>
<tr>
<td>Project Approach (P)</td>
<td>Communication Plan (P)</td>
<td>Project Communications (P)</td>
<td>End Project Report (P)</td>
</tr>
<tr>
<td>Acceptance Criteria (P)</td>
<td>Issue Log (P)</td>
<td>Lesson Learnt Report (P)</td>
<td></td>
</tr>
<tr>
<td>Initiation Stage Plan(P)</td>
<td>Project Filing System (P)</td>
<td>Stage Plan (P)</td>
<td></td>
</tr>
<tr>
<td>Project Plan (P)</td>
<td></td>
<td>End Stage Report (P)</td>
<td></td>
</tr>
<tr>
<td>Quality Log (P)</td>
<td></td>
<td>Exception Plan (P)</td>
<td></td>
</tr>
<tr>
<td>Configuration Management Plan (R)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Key**

- Overlap resolved in favor of PRINCE2
- Overlap resolved in favor of RUP
- No major overlap
- P PRINCE2
- R RUP

**Team Level Documentation**

Table 4 shows RUP planning documentation for the team level. RUP documents with a broader focus -- for example, the Business Plan, Quality Plan, and Project Plan have been pushed up to the project plan level.

*Table 4: Team Level Documents*
### Initiation (Inception) vs Stages (Elaboration, Construction, Transition)

<table>
<thead>
<tr>
<th>Initiation (Inception)</th>
<th>Stages (Elaboration, Construction, Transition)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Development Plan (R)</td>
<td>Iteration Plan (R)</td>
</tr>
<tr>
<td>Risk Log (P)</td>
<td>Iteration Assessment (R)</td>
</tr>
<tr>
<td>Issue Log (P)</td>
<td>Work Order (R)</td>
</tr>
<tr>
<td>Acceptance Criteria (P)</td>
<td></td>
</tr>
<tr>
<td>Project Measurements (R)</td>
<td></td>
</tr>
<tr>
<td>Measurement Plan (R)</td>
<td></td>
</tr>
</tbody>
</table>

### Key

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Overlap resolved in favor of PRINCE2</td>
<td></td>
</tr>
<tr>
<td>Overlap resolved in favour of RUP</td>
<td></td>
</tr>
<tr>
<td>RUP artifact re-scoped to team level</td>
<td></td>
</tr>
<tr>
<td>No major overlap</td>
<td></td>
</tr>
<tr>
<td>P PRINCE2</td>
<td></td>
</tr>
<tr>
<td>R RUP</td>
<td></td>
</tr>
</tbody>
</table>

PRINCE2 also defines methods for risk management and issues management, so the RUP risk and problem management documentation is not required.  

### Organization

Figure 10 shows a proposed structure for a PRINCE2 project that is integrating RUP. There are two levels in this structure -- project and team -- and the roles are split as follows:

- PRINCE2 defines the project manager role.
- The RUP project manager role maps to the PRINCE team manager role; a team manager is responsible for a team that produces specialist products such as software.
- The PRINCE configuration librarian role maps to and replaces the RUP configuration manager role.
- Team members in other specialist roles, such as system analyst, report to the RUP team manager.
This article has discussed the desirability of a loosely coupled integration of RUP, the way to merge the RUP and PRINCE2 project management documentation sets, and the impact of RUP on the PRINCE2 organization structure.

The benefits of this integration approach include:

- **More leverage for your PRINCE2 investment.** You can use your existing process, and "you don't have to throw the baby out with the bath water."

- **A best of both worlds solution.** You get the "big picture" control PRINCE2 provides for overall project management, plus RUP's iterative approach, which is suitable for developing a creative product such as software and may apply to other aspects of the project as well.

- **A simple solution.** The integrated process is understandable to both PRINCE2 and RUP communities and reduces the learning required by each practitioner.

- **A "do-able" approach.** The integrated process is possible to implement within a short time frame with a minimum amount of effort.

- **Separation of concerns.** RUP focuses on the development of software products, and PRINCE2 focuses on the non-technical aspects of the project.

- **A flexible, open process.** Loose coupling allows other methods to be used with PRINCE2.

A loosely coupled approach to the integration of PRINCE2 and RUP can
speed up the tailoring effort, reduce costs, and leverage your investment in existing PRINCE2 skills and experience.

Notes

1 "PRINCE® is a Registered Trade Mark of the Office of Government Commerce (OGC). All PRINCE2 products published by the Office of Government Commerce (OGC) are © Crown Copyright Value Added Products which fall outside the scope of the HMSO. © Crown copyright material is reproduced with the permission of the Controller of HMSO and Queen's Printer for Scotland

2 PRINCE2 uses the term specialist to denote any activities or deliverables produced outside the scope of PRINCE2.

3 The project board provides overall direction and management for the project. An equivalent body in RUP is the project review authority.

4 The team manager on a RUP project is typically the project manager.

5 When there is an indication that a plan is predicted to exceed significantly the agreed time and cost, an exception plan is produced. The exception plan takes over from the plan it is replacing and contains additional information on the cause of the exception.

6 These PRINCE2 documents (and others) are also upward-compatible with program management, where a program is a portfolio of projects that collectively deliver the required outcome. See OGC's Managing Successful Programmes at http://www.tso.co.uk/bookshop/bookstore.asp?FO=39139

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

*Component Software: Beyond Object Oriented Programming, Second Edition*

by Clemens Szyperski

Addison-Wesley, 2003  
ISBN: 0-201-74572-0  
Cover Price: US$54.99  
608 Pages

What are software components? We all have a preconceived notion of what they are but have difficulty putting our thoughts into words. It is very challenging to define something that is so abstract. That is why Clemens Szyperski, a well-established researcher and writer on the subject of component technology, has taken 589 pages to explain it. In the very beginning of his book he states that:

> Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system.

Don't worry if you don't understand this, because the book goes on to explain in great detail what components can be and how they are used in today's industry. First, Szyperski introduces the basic theory behind components; then he covers many technological practices, including object-oriented principles and Web services; and finally, he closes by introducing even more theory and then wrapping everything up in a conclusion.

The book remains demanding, though. Szyperski warns readers early on that some of the upcoming passages are difficult reads, and I did find that some of his points went way over my head. He targets mainly computer theorists, systems architects, and integrators, software developers, and CTOs, and assumes familiarity with object-oriented principles. As the book focuses on theory rather than code, it is not a good reference for specific languages such as C#. It does, however, explain why programmers use each specific language. I definitely recommend it for anyone searching for an in-depth look at today's component technology. If you have a fear of computer-related acronyms, *Component Software* will become an invaluable tool for you.

Are Objects Components?
In his opening discussion of component theory, Szyperski posits that a component has three characteristic properties:

- It can be deployed independently.
- It is a unit of third-party composition.
- It has no (externally) observable state.

Since most programmers today use object oriented-based languages such as Java and C++, they intuitively think of objects and libraries as components. There is little dispute that libraries should be classified as components, but there is much debate over whether objects are components. Szyperski says that an object:

- Is a unit of instantiation; it has a unique identity.
- Unlike a component, may have state, and this can be externally observable.
- Encapsulates its state and behavior.

Note the conflict: Components cannot, technically speaking, have externally observable states. So, although some objects can act as components, it is not accurate to classify all objects as components, Szyperski explains. Also note that not all objects used in an application are from third parties; many are developed in-house. Just because an object can be instantiated many times within an application does not necessarily mean it will have any functional value within another; in other words, it is not necessarily a reusable component. Objects created by third parties are typically contained within libraries, which are built for reuse. Those objects are considered to be components -- but again, as only some objects act as third-party components, it is simply inaccurate to classify all objects as components.

**The Component Dilemma**

To be a component or not to be a component. That is the big question Szyperski poses in the theory section. Basically, he argues that if you can't foresee reuse for a piece of software, then it may be more efficient and cost-effective to create custom methods.

Typically a component includes an interface that a client uses to communicate with the component. If the component is based on a black box model, the client may have very little knowledge of the component's inner workings, which makes the interface crucial to understanding what the component does and how to communicate with it. The problem is, as Szyperski explains it, that the more specifically a component's function addresses your particular needs, the more intricate the interface becomes; this can result in less potential for component reuse and unjustifiable development costs, especially if you need only a small portion of that functionality yourself. The unofficial "standard" in the component field says that a component must be used at least 2.5 times in order to prove its
value and justify your investment. If you can't foresee this much reuse, then it may not pay to build a component interface.

On the flip side, if your component has less functionality and an easier interface, you may need more components to perform your desired task, and tracking them may become an even larger challenge. The key to creating good component architecture is to find a happy medium and create components that are both manageable and reusable.

**Comprehensive Technology Coverage**

The technology section, which comprises most of the book, is dedicated to specific uses of components with today's technology and principles; it even discusses components in the software market. Szyperski starts with principles such as inheritance (how to avoid it), polymorphism, and subtypes and explains how they come into play when you build components. Later, Szyperski compares his own definition of components with those of other industry experts such as Rational's Grady Booch. In his book, *Software Components with Ada: Structures, Tools and Subsystems* (1987), Booch includes this definition:

> A reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction.

Written more than sixteen years ago, this reflects Booch's visionary thinking, although the definition does not acknowledge environmental dependencies or require that a component be independently deployable.

The most useful part of this section is near the end, where Szyperski analyzes how some of the industry's leading technologies -- J2EE, .NET, CORBA, COM, SOAP, and XML -- relate not only to components and Web services, but also to each other. He explains the purposes behind Servlets, Enterprise JavaBeans, Swing, AWT, and different Java platforms such as J2EE, J2SE and J2ME. He does an excellent job of summarizing .NET, J2EE and Component Pascal and explaining the different ways that companies cope with designing component architecture. Though Szyperski now works for Microsoft, I detected no evidence of bias toward Microsoft products.

**A Definition?**

Don't expect to find a conclusive definition of components in this book. That is not Szyperski's purpose. Instead, he provides a great deal of helpful detail about the theories, people, and market forces driving component software. At the end of the book, I felt satisfied. Szyperski showed me why it is impossible to pin down a single, formal definition of a component, and he did what was appropriate: educate the reader and leave the issue open for further discussion.

*Jeff Livingston*
Rational Software
IBM Software Group
For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!

Copyright Rational Software 2003 | Privacy/Legal Information
At first glance, a reader looking at this book might think, "Humph! Another book on software engineering!" But this is definitely not a typical software engineering book.

Designed for use in the classroom, the author's mission is to teach newcomers to the field the fundamental facts of software engineering and point out fallacies in the thinking of both practitioners and theorists. A distillation of Glass's half century of software experience, the book offers a personal, somewhat biased, look at the field that may actually be more helpful to people who have worked in the industry for a few years, including those involved in maintenance, development, and project management.

Glass groups his fifty-five facts about software engineering into four broad categories: management, lifecycle, quality, and research. He groups fallacies into three categories: management, lifecycle, and education. Within these categories, he classifies by topic, so readers can easily focus on the facts and fallacies of particular interest to them. (Although there is no prescribed order for reading most of the facts in the book, Glass does occasionally refer to previously mentioned facts.)

In each instance, Glass states the fact or fallacy clearly and simply, explains what it means, discusses the controversy surrounding it -- drawing on various books, articles, and research papers published by credible authors -- and concludes with sources and references. Many of these facts and fallacies will be familiar to readers, and I found it gratifying that Glass articulates some of the opinions that I've formed, based on my own industry experience.

He also consciously avoids domain-specific words (jargon), although he does refer to debuggers, coverage analyzers, and the like -- and the book could use a Glossary to define these references. The book concludes by
classifying and presenting four underlying themes for the facts and fallacies: complexity, estimation, disconnect, and hype.

A Few Facts and Fallacies

Not all the facts are fully explained, however. Take this one for example:

Fact-36: Programmer-created built-in debug code, preferably optionally included in the object code based on compiler parameters, is an important supplement to testing tools.

Although the term "testing tool" might convey the impression that this is a technique testers should use before production (which is true if the testing team does white-box testing), in my experience, the built-in debug code is also useful for debugging after a system goes into production.

Some of the facts also seem counterintuitive upon first reading:

Fact-45: Better software engineering development leads to more maintenance, not less.

But then Glass goes on to clarify that more maintenance is required because end-users want to change the software so that they can use it longer -- longer than their creators might have predicted. This was true for much of the software that caused so much alarm as the year 2000 approached, for example. In some of his other facts, the author clearly reveals his bias that software maintenance is more challenging than software development.

Actually, many of Glass's "fallacies" are simply his way of packaging the personal wisdom he wants to pass on to novice programmers. E.W. Dijkstra once observed that programmers should prove the correctness of their programs before they program. But given the complexity of today's software systems, says Glass, it may not be practical to follow this advice. So in Fallacy 10, he suggests that, instead, programmers should study the programs of other proficient programmers and learn from them.

Glass also believes that a programmer has a natural attachment to his/her programs; hence:

Fallacy-3: Programming can and should be egoless.

He also thinks it's a mistake for organizations to try fitting square pegs in round holes:

Fallacy-4: Tools and techniques: one size fits all.

The set of tools and techniques you use should be adaptable for the project at hand, he goes on to explain; some tools and techniques may not be suitable at all for a particular type of project.

But he quickly adds that we already have quite enough methodologies to adapt to a variety of situations, in the form of:
Fallacy-5: Software needs more methodologies.

Just as with the facts, however, some of the fallacies are not explained fully enough.

For example, there is:

Fallacy-6: To estimate cost and schedule, first estimate lines of code.

Based on my experience, this statement is certainly true, but Glass provides no detailed, real world examples to prove it, and the discussion in the controversy section for this fallacy does not clarify his position or present the right way to estimate cost and schedule.

I also found it irritating that Glass uses a nintumber of culture-specific words, such as "betcha," which is not listed in many popular dictionaries or at www.dictionary.com. As the book is for an international audience, it seems to me he could have avoided such words.

Conclusion

This book has significant strengths: It consolidates wisdom the author gained through decades of practical experience and covers various disciplines within software engineering; in addition, the facts and fallacies are presented in simple language and are typically easy to translate into practice.

It's a highly personal view of the software world, though; many of the references are to the author's own books and papers, and many facts and fallacies are simply platforms for Glass to express his biases.

I would not suggest this as a text for either college students or those just entering the industry. However, I would use it in software project management workshops for experienced practitioners as a way to launch discussions and review important facts and fallacies. I'd also recommend it as a useful reference for practitioners to consult during "live" projects.

-Saya Sreenivasulu
Rational Team Lead, Bangalore
IBM Software Group

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
Tracking Test Projects with a Virtual Status Portal: Leveraging Rational ClearCase and the Web

by Susan McVey
Software Quality Engineer
Rational Software
IBM Software Group

In any testing project there is a tradeoff between the time you must spend managing the testing -- that is, planning tests, documenting their results, and keeping track of progress -- and the time you are able to spend actually testing the product. When a testing team is divided among different work sites, the problem is compounded. That is why my team, the Rational® PurifyPlus for UNIX Quality Engineering group, developed a lightweight, adaptable system of test management documents. By putting these documents under the control of Rational® ClearCase and creating a local Web page at each site to display a summary of the current testing status, we improved communication among engineers and managers located at different sites, made the state of the product visible to everyone on the team, and streamlined our test planning process -- all with very little overhead. In this article, I'll briefly describe what we did as well as the lasting benefits it has produced.

Lightweight Documents for Test Planning

Let me begin with a brief description of the project and team. Rational PurifyPlus for UNIX is supported by a group of software engineers and a companion team of software quality engineers. Initially, they were all located together in Cupertino, California, but in recent years the team has grown to include another set of developers and testers in Bangalore, India. Although some testing projects are based in one office or the other, the two sites share many of the tasks.

Like any mature product, ours is a mix of old and new code that has
grown with time. Although the core of Purify is ten years old, the application is subject to continual change in order to support new compilers and operating systems, and new features are added every year. To test the product efficiently, we needed a fast and lightweight planning process to use when testing incremental changes in old code, but a rigorous one for testing new features. Our goal was to produce enough documentation to enable us to test reliably and thoroughly, but not to waste time being more formal than was useful.

After experimenting with different levels of formality in our documents over the course of several iterations, we settled on a system of quality engineering checklists. A checklist is a document that contains both the test plan and the test records for one area of the product -- usually a new feature, support for a new compiler or operating system version, or a release validation project. Although the industry standard is to have test reports in separate documents from test plans, we found it useful to combine these documents for easy viewing in a Web browser. A quality engineering checklist contains a description of the feature under test, instructions for setting up the test environment, and a list of testing tasks. Each task has a brief description and a place to record the test execution. The checklist also contains testers' notes and pointers to problem reports in the bug-tracking database (see Figure 1).

Figure 1: Quality Engineering Checklist for a Rational PurifyPlus for UNIX Project

Testers keep detailed notes in their lab notebooks, but the checklist provides the broad view. For repetitive tasks like validating a build, testing a new version of the installation program, or testing Rational Purify with a new compiler, there is a template checklist that can be instantiated and modified easily for each iteration. For testing new product features, we follow a more rigorous and traditional process of reviewing the functional specification and design, writing a test plan, and then reviewing that plan before beginning to test. As a rule, each Rational PurifyPlus release has about twenty-five to thirty-five checklists, of which more than half are instantiations of standard template checklists. We may also create checklists for test infrastructure projects. A master checklist for each release lists all the other checklists and includes Web links to each one.
The checklists are stored as `.html` files under Rational ClearCase. Files can be edited either with Netscape Composer or a text editor such as vi or emacs, but the editor must be invoked from a shell window inside a ClearCase view so it can see the files. In order to ensure that only one copy of the records exists, the test documents are merged directly into the main ClearCase branch and are not split off onto release branches as the product source code is. Instead, the checklists for each release are contained in a separate subdirectory.

To make the checklists visible from a Web browser, we created a link from the departmental intranet Web site to the master checklist for each release. And, as we noted, links from the master list lead to each of the feature or release checklists. Although most of the people using the checklists are engineers who have access to the files in Rational ClearCase, there are other users who want to view the checklists from a PC over the intranet. Since these users only need to view the information, not edit it, we simply export a copy of the checklists onto a disk that is shared among the UNIX and PC systems. A Perl script updates the exported files by starting a view and copying the files out from the Versioned Object Base (VOB). This can be invoked either by a cron script or by using a trigger to execute the export script whenever a checklist is merged in from anyone’s personal branch to the main branch. Hypertext links in the checklists are relative to the current directory and do not include a full pathname, so they work equally well in the VOB and in the exported copy.

**The QE Whiteboard**

In order to give a big-picture view of testing and product status, we use a modified form of the whiteboard system proposed by James Bach in his exploratory testing method ([http://www.satisfice.com](http://www.satisfice.com)). We began using
this system when the whole team was at one site. On a large whiteboard in our hallway, we made a chart with one line for each QE checklist. The current state of the checklist was summed up by a number indicating the amount of testing completed, an indication of whether the testing was taking place that week, and a small, painted magnet that indicated whether that the feature was ready to ship (a smiling green face), had serious bugs (a frowning red face), was still being evaluated (a neutral black face), or was waiting for testing to begin (a blank grey circle). For the frowning red faces, we also wrote bug numbers from the bug-tracking database on the board. Over the course of a release cycle, the faces would gradually change from red to green; when everything was green, the product was ready to ship.

The whiteboard is helpful to several different users: The engineering manager looks at the board to see how well the product is working and to make decisions about when to ship patch releases and betas. Engineers working on a product feature use it to see whether their code is being tested and whether any serious problems were found. The QE manager can track the progress of testing tasks, decide where the testers need to concentrate their effort from week to week, and see whether any problems are blocking test work. Quality engineers often gather around the board to discuss the tests and decide whether to change the faces and other information.

Like many things in the software world, this system worked reasonably well until conditions changed. When the team in Bangalore joined us, we had two problems to solve: Both teams needed to use the same checklist files at the same time, and the engineers in Bangalore could not see the whiteboard in our hallway in California. The solution for file sharing was obvious: The product group was already using Rational® ClearCase MultiSite® to share source code between the two sites, so we decided to share the test documents using this application as well.

**An Online Testing Status Portal**

The physical whiteboard was only accessible in one location, so it had to be replaced by an intranet web page that could be viewed from both sites. The whiteboard entries summarized the contents of the checklists, so we decided to generate a virtual whiteboard Web page from the checklist files in Rational ClearCase. Not only would the board be visible and editable in the ClearCase files at both sites, but also we could expand the board to include task completion information that would have been too time-consuming to update continually on the physical board (see Figure 3).
An earlier incarnation of the QE documents had included schedule and status information encoded in lines with a tag that could be recognized by a Perl script for report generation. I decided to use the same strategy for the whiteboard. At the top of each checklist we added a set of tag lines containing whiteboard information. Lines marked with "+++" and a keyword were designed to be easy to extract and parse with a script but still easy for people to read as well (see Figure 4).

---

**Status:**

+++Release: 2003.06 McKinley
+++Description: Solaris 9 OS support
+++Effort: SHIP
+++Coverage: 2+
+++Assessment: GREEN
+++Comment: Test checklist complete.

---

Each checklist contains tags for the information displayed on the board:

- The name or one-line description of the checklist (+++Description).
- The current level of effort devoted to running tests for this checklist - - or "ship" if the checklist is complete (+++Effort: HIGH, LOW, WAITING, START, SHIP).
- The amount of test coverage achieved so far for this feature, expressed as a number from 0 to 3, and the target number the
team expects to reach (+++Coverage).

- The assessed quality of the feature or release being tested, using the same symbols we used on the original board. We changed the neutral face from black to yellow for better visibility and replaced the blank grey circle with a simple "N/A" for features whose quality has not yet been assessed (+++Assessment: RED, NEUTRAL, GREEN, N/A).

- One or more comment lines, added by team members. These appear in a "Comments" column and typically convey information such as "Cannot test further until bug 12345 is fixed" or "Waiting for new compiler to be installed." (+++Comment)

In addition to these checklist-level tag lines, there is a "+++ Status" line associated with each testing task in the checklist with the options Done, Not Done, Paused, Waiting, or Deleted. If the test plan has a group of related tasks, such as running a test with several different compilers, then a single status line reports the status of that entire group of tasks. The whiteboard reports the number of tasks completed and the total number planned. This task completion number differs from the test completion number mentioned above: The former is a measure of plan completion, whereas the latter is a measure of test coverage for the product. For an incremental change that is well understood, the target level of product coverage might be low, indicating that we will run only a simple regression test; however, we would still expect the test plan coverage to reach 100 percent execution before the product is approved for release.

The main Web page for a release consists of a description of the product and its changes from the previous release, plus a list of the test checklists. It also contains tag lines marked "+++Header:" which are displayed at the top of the virtual whiteboard. These are used to identify the release and to display schedule and status information.

A Perl script developed by my colleague, Tom Arnold, reads all the checklists for a release, parses the tag lines, and generates a Web page for the whiteboard as well as an index page with links to all the currently active whiteboards. Multiple whiteboards can be active if work is being done on more than one product release at a time (see Figure 5).
When a user requests an update by clicking **Update** on the whiteboard Web page, the Perl script starts a Rational ClearCase view, copies the latest version of the checklists from the VOB to the export area, and regenerates the whiteboard from the checklists. There is an export area and a copy of the script at each of the sites, generating a local copy of the virtual whiteboard. This enables Web users at either site to receive up-to-date information quickly, whether they have direct access to the VOB or not.

**Problems and Opportunities**

Our initial system worked well, but with each release we have made a few changes to optimize it. One improvement was the addition of multiple quality assessments (faces). During the beta iteration of the project, sometimes a feature was judged good enough for the first beta (green face) but still needing additional work before final release (red face). The board grew crowded with comments explaining which iteration each assessment applied to. To simplify, we implemented the ability to assign a different quality assessment for each iteration. Instead of having one "+++Assessment" line, a checklist can have "+++Beta1_Quality", "+++Beta2_Quality", "+++Final_Quality", and so on. We also added a new tag -- "+++Show_Quality" -- to the master checklist so we could remove tags for past iterations from the board.

When a file is edited at both sites between one day's update and the next, Rational ClearCase can do an automatic merge, as long as people at both sites have not edited the same lines in the file. In the first version of the checklists we used a text file instead of .html, and there were tables in the file that had tests for Solaris in one column and HPUX in the other. Since Solaris and HPUX are tested at different sites, both sites had updates on the same line of the file, which meant they could not be merged automatically. By separating Cupertino and Bangalore tasks into different lines or paragraphs in the document, we avoided the merge problems and did not have to assist the update process.
An extension of this was the problem of scheduling the daily merges. California and India are in nearly opposite time zones, so there is no time of day or night when we can guarantee that no one in either place will be updating a checklist. Inevitably, there are times when someone at one site is in the middle of editing a file when an update from the other site comes in. To avoid problems because of this, we eventually began splitting feature test checklists along geographic lines: Instead of one checklist for a new gcc compiler we have two copies: one with the Solaris (Cupertino) work and another with HPUX (Bangalore).

We also added the ability to hide information that is no longer useful. Each prototype or beta release has one or two checklists, so the board gets crowded after a while, and there is no need to keep displaying information for something that has already shipped. We put a "+++Closed" tag into obsolete checklists to remove them from the whiteboard display; if team members need to consult a closed checklist, they can still follow the link to it from the master checklist.

A final improvement of the virtual board over the physical one was to include Web links. Clicking on any information in the board will show you its definition. Clicking on the name of a feature takes you to its checklist. Clicking on the test plan completion shows you a list of the tasks (individual "Status" lines in the checklist) that indicates whether they are done.

**Is Virtual Better than Physical?**

The change from a physical whiteboard to a virtual one was forced upon us by circumstance. But what about other teams, especially those that are not split geographically? Would this system work for them? I think it would. Even though I liked that board in the hallway, I have to admit that the virtual board and shared Rational ClearCase files are working much better for us than the old board ever did. For one thing, when someone checks in a change in a checklist, no one has to remember to go update the whiteboard. Managers don't have to come knock on my door and ask how the testing is going, because they can see how it's going right from their own desktop. The numbers that track task completion help us judge how well we're doing on meeting our deadlines.

Best of all, the problem I had worried about most never happened. When we had the board in the hall, the quality engineers used to gather around it and talk about how things were going. I worried that when we went online, the conversations would stop, and we would lose some of the "glue" that unites us as a team. As it turns out, though, we still talk -- we just gather around someone's screen or bring a printed copy of the board to our meetings. And now we have better, more up-to-date information to talk about. All in all, I can't see how or why we would choose to do without the new system now.

So here's my parting suggestion: Whatever the form of your own test plans, checklists, or other status reports, I recommend that you put the source data in Rational ClearCase files and generate an online status board. You'll be surprised at how much of your time will be freed up for
testing your products. Just take a look at Figure 6!

Figure 6: An Online Status Board Frees Up Valuable Testing Time

Notes

1 Rational PurifyPlus for Unix includes the Rational® Purify®, Rational® Quantify®, and Rational® PureCoverage® product capabilities.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
Thoughts on the Craft of Programming: Abstraction, Refactoring, and How Changes Introduce Bugs

by Michael J. Harrison
Software Development Manager, USPM
(Unified Software Product Management)
Rational Software
IBM Software Group

This article is about the craft of programming, which comprises both scientific and artistic elements. By "scientific," I mean that programming concerns logical propositions that are objectively true or false, and by "artistic," I'm referring to the many ways to formulate these propositions and the need to make subjective judgments about which are good and which are bad. When discussing software development, we tend to bandy about the grand concepts of modularity and abstraction without a real understanding of how these concepts are reflected in our everyday work. When it comes to improving and refactoring chunks of software code under development, I believe it's worth exploring the reasons -- the motivations -- behind the code improvements. I think there are some general principles that may guide programmers toward producing code of higher quality.

One of the problems, of course, is that in making code changes, we frequently introduce new errors into the code, even when we are being careful, even when the change is apparently simple and the existing code is well-structured. So to make my ideas concrete, I will examine a fragment of code known to contain an error, and I will explain the error and illustrate a simple change that will correct it. I will then make several improvements to the code to make it easier to read and maintain. Finally, I will add a new feature. The piece of code used here is trivially simple, much too small to really show the advantages of modularity and
refactoring, but I believe it is big enough to illustrate the basic principles. It's written in C and uses native types, because that's how the code that originated all this discussion was designed; and I have kept the K&R style because I feel it is, as Tony Hoare said about Algol-60, such an improvement on all its successors. For clarity, I have omitted the `#include` statements that would be required.

### Fixing a Bug

The first code sample (Figure 1) comes from one of the advertisements that Gimpel Software runs every month in *Dr. Dobb's Journal*; each advertisement illustrates a bug that could be found by their enhanced version of the source code diagnostic program lint and that therefore can be discovered by visual inspection of the code. The failure, as stated in the ad, is "This function which is intended to count the vowels in the string provided is taking a long time to do so." I found that under either Windows or Linux the actual failure is that an addressing exception is generated.

```c
/* Version 1: Original code from Gimpel ad. 
 * Gimpel Bug #527, DDJ, December 1999, p. 96 */
int count_vowels(char *s)
{
    int sum = 0;

    for (;;) {
        switch (*s++) {
            case 'a':
            case 'e':
            case 'i':
            case 'o':
            case 'u':
                sum++; continue;
            default: continue;
            case '\0': break;
        }
        return sum;
    }
```

**Figure 1: Incorrect Code from Gimpel Ad**

The error results from the coder's failure to see and appreciate the implication of C syntax: the `break` statement is in the scope of the `switch` statement, not that of the `for` statement, and consequently causes exit from the `switch` but not from the `for`. One is misled by the two `continue` statements, which are indeed in the scope of the `for` loop. So the `for` statement in fact has no termination condition, and thus after a potentially long series of character accesses, an invalid address is generated.

### Making the Initial Bug Fix

An explicit infinite loop (such as the C idiom "for (;;)") should always be viewed with suspicion, for very few loops, if indeed any, are truly endless. Let's put the termination condition in the loop, which is clearer and also fixes the bug. The new `for` statement is another piece of C idiom (a cliché,
if you will) that is instantly recognizable to an experienced C programmer as an instance of the scan-the-whole-string pattern (though some C programmers would omit the explicit comparison to the null character and rely on C’s implicit comparison). Note that we no longer need the "case '\0':", since the loop condition now guarantees this case would never arise. This code is definitely more readable than the original, and an experienced programmer could comprehend the whole thing in little more than a glance.

Although Java's `for` and `switch` statements work in exactly the same way as C's and expose the programmer to the same error, a Java programmer almost certainly would not have made this mistake. That is because Java strings use a length rather than a terminating character, and a Java programmer would automatically write the loop in the Java idiom:

```java
for (int i = 0; i < s.length(); i++)
```

So, by using a termination condition directly in the loop definition, we can fix the bug as shown in Figure 2:

```java
/* Version 2: Use termination condition directly in loop definition. */
int count_vowels(char *s)
{
    int sum = 0;

    for (; *s != '\0'; s++)
        switch (*s) {
            case 'a':
            case 'e':
            case 'i':
            case 'o':
            case 'u':
                sum++; continue;
            default: continue;
        }
    return sum;
}
```

![Figure 2: Fixing the Bug](image)

**Cleaning Up the switch Statement**

That fixed the bug, but the `switch` statement looks a bit silly now, and would look even sillier if I had stripped it to its bare essentials by also removing the `default` case and the preceding `continue` (which is made redundant by the removal of the `default`) -- even though, if compiled using a branch table, the `switch` statement would be quite efficient. In any case, I tend to look askance at `continue` statements. Figure 3 shows how we can replace the `switch` with a more straightforward `if` statement.
Replace the Ugly if

That's reasonable and not too inefficient, but the if statement looks a bit ugly. We now need to switch our focus from the scientific -- getting the code correct -- to the artistic -- making the code good. Let's replace the ugly statement with a nicer test, using the C run-time library: a tiny example of component-based development, with the library acting as a component. Usually, strchr is employed to search for a given character in an arbitrary string; here we invert that to search for an arbitrary character in a given string. Its specification is "strchr searches a string for the first occurrence of a character; if the character is found in the string, a pointer to the first occurrence is returned, otherwise a null pointer is returned." We use it in the logically equivalent way: The character occurs in the string if and only if strchr returns a nonzero value. On x86 CPUs with a decent compiler this version will also be a little faster. Notice that the total amount of text (Figure 4) is now substantially reduced, which also contributes to the code's being easier to read and understand.

Making a Simple Enhancement

As shown in Figure 4, we might notice that only lowercase letters are counted as vowels. Let's improve the code so that it will accept uppercase vowels as well. Given the improvements we've made thus far, this is now very easy and safe to do, since the only change needed is to extend a static array. Note that if we had made an equivalent change to Version 3,
the if statement would have become really ugly and significantly inefficient. The change shown in Figure 5 is by no means the only way to implement this enhancement; arguably it would be just as good, for example, to initialize the `vowels` array with only the uppercase vowels, and pass `toupper(*s)` to `strchr`.

```c
/* Version 5: Deal with uppercase as well. Note how easy this change is, */
/* now that the logic is contained in a static declaration. */
int count_vowels(char *s)
{
    static char vowels[] = "aeiouAEIOU"
    int sum = 0;

    for (; *s != '\0'; s++)
        if (strchr(vowels, *s))
            sum++;
    return sum;
}
```

*Figure 5: Enhancement to Accept Uppercase Vowels*

**Refactor to Minimize the Scope of the Definition of "Vowel"**

That code looks reasonably good, but it does combine two functions that are really quite separate: scanning the string and deciding whether a character is a vowel. Once again, our artistic antennae are alerted. Scanning a string is an instance of the more general function of traversing a data collection, so with the Visitor pattern at the back of our minds we decide to separate the two functions. This also makes sense because the definition of "vowel" may vary between languages (and indeed may not exist at all in some), so this is a part of the program that looks as though it might depend on the target environment and hence one that should be isolated into a small function. With this change, the total amount of text has increased a little, but the conceptual complexity has been reduced because each function can now be viewed in isolation.

Although I will not demonstrate it here, the code shown in Figure 6 is ready for further generalization, because any `is_x` function with the same signature as `is_vowel` could be passed as an argument to the string scanning function, and so all sorts of different tests could be realized by writing only additional code (i.e., not changing existing code). In C++ or Java, instead of passing a function address (impossible in Java) one would wrap each function in its own class, declaring `is_x` as a method and pass as the argument an object of the appropriate class (the functor idiom), and in C# one could use a delegate, but the principle is exactly the same.

Now that it is a separate function, `is_vowel` invites a more efficient implementation -- for example, one based on a 256-byte array of character properties in the manner of the `isalpha` function and its companions\(^5\). This would be worth doing if profiling revealed that this function was an execution bottleneck.

The function `is_vowel` is a candidate for an independent unit test. Indeed,
since there are only 256 possible values of \( c \), this is an example of that rare case: a piece of code that can be exhaustively tested. So we should take advantage of this and write and execute such a test in this instance; this would be particularly worthwhile if we planned to improve the implementation as suggested in the previous paragraph. We'll return to this point a little later.

```c
/* Version 6: Refactor, to minimize the scope of the definition of "vowel". */
int is_vowel(char c)
{
    static char vowels[] = "aeiouAEIOU";
    return strchr(vowels, c) != 0;
}

int count_vowels(char *s)
{
    int sum = 0;
    for (; *s != '\0'; s++)
        if (is_vowel(*s))
            sum++;
    return sum;
}
```

Figure 6: Refactor to Minimize the Scope of the Definition of "Vowel" Code

### A New Feature: Counting the Number of Vowel Pairs

The new feature is to count the number of vowel pairs in the string, or more precisely the number of instances where \( s[i] \) and \( s[i+1] \) are both vowels; for example, the vowel pair count for "beautiful" is 2, for "ea" and "au"\(^2\). Figure 7 shows the new function, `count_vowelpairs`, and its supporting function, `is_vowelpair`:

```c
/* Version 7: Implement a new feature - count vowel pairs. */
int is_vowelpair(const char *p)
{
    return is_vowel(*p) && is_vowel(*p+1);
}

int count_vowelpairs(char *s)
{
    int sum = 0;
    for (; *s != '\0'; s++)
        if (is_vowelpair(s))
            sum++;
    return sum;
}
```

Figure 7: Counting Vowel Pairs

That was easy -- a little copy and paste, a minor edit, a one-line function, and we're done. Note how, because of the previous refactoring, we were
able to use the `is_vowelpair` function to implement `is_vowelpair`. Note also that if in the original implementation of `is_vowel` we had passed a character pointer instead of a character, we could have implemented `is_vowelpair` as an `is_x` function and used the function pointer protocol mentioned previously to implement the new feature without having to implement `count_vowelpairs` as a separate new function. But we were not sufficiently farsighted, and the decision to pass a character was a wise one, based on the principle that it is safer not to pass a pointer if it can be avoided.

But are we really done? No, we have been a little cavalier in our approach, and the program contains an error, one that might well evade testing.

On hearing that, a programmer might first be drawn to the `is_vowelpair` function because it dereferences `(p+1)`. Is that always a valid address? Yes, in fact it is, because the test in the `for` statement guarantees that `p` always points to a character in the string, so even when `p` is pointing to the last character in the string, `(p+1)` is pointing at the null character terminating the string and so is still a valid address. We were lucky when we made the change, because we had not explicitly considered this. There is no way in C to protect `is_vowelpair` from being invoked with a pointer `p` such that `(p+1)` is an invalid address; we resolve to ensure that the specification for `is_vowelpair` contains a prominent notice of this precondition.

So what is the bug? It is this: if the string ends in a vowel `x`, the character pair "x\0" will be incorrectly identified as a vowel pair. The groundwork for this bug was laid in the improvement from Version 3 to Version 4; it turns out that the `if` test and the `strchr` test are not exactly equivalent. The function `strchr` considers the terminating null character to be part of the string for the purposes of the search, so searching for the null character will always succeed and return a non-zero pointer. This is documented in the specification for `strchr` but is not well known; it is not mentioned in the MSDN Library documentation for the Microsoft Foundation Classes method `cString::Find` (which itself calls `strchr`). Yet it is on such minutié that the correct execution of a program may hinge.

Let's return briefly to the point about unit testing in the previous section. The long-term value of such testing is now more clearly apparent, and if we had doubts earlier about whether the investment of time to write and execute the test would be repaid, those doubts are now probably resolved. However, tests must be written with as much care as code, and in this particular case we must be sure to test the case when `c` is zero.

How should we fix this bug?

One way would be to replace the body of `is_vowelpair` with the statement:

```
return is_vowel(*p) && *(p+1) != '\0' && is_vowel(*(p+1));
```

This will fix the bug, and in one sense it is quite correct, because it simply
compensates for the slightly awkward behavior of `strchr` in our context. Another interpretation might be that the error lies in `is_vowel`, which classifies the null character as a vowel, and that this function should be changed to correct the error (I hear Bertrand Meyer -- see footnote 9 -- in the background chiding us for an insufficiently precisely defined precondition for `is_vowel`\(^9\). I maintain, however, that either of these changes treats the symptom rather than the disease. Our original suspicion of \((p+1)\) was justified; I contend that the proper precondition for `is_vowelpair` is that both \(p\) and \((p+1)\) point to actual characters within the string, not to the terminator. Therefore a correct fix for the bug is to leave `is_vowelpair` as it is and replace the `for` statement in `count_vowelpairs` with this:

```
for (; *s != '\0' && *(s+1) != '\0'; s++)
```

It is necessary to test both characters because testing only `*(s+1)` will fail if the string is null (for then `*(s+1)` would reference an invalid address). This is a bit clumsy, and we might want to consider testing for a null string before entering the loop or using a different termination condition, perhaps one based directly on the length of the string. We might also want to consider weakening the precondition for `is_vowel` by making it classify the null character as not a vowel. As the saying goes, these things are left as exercises for the reader.

**An Aside on Programming by Contract**

Programming by Contract\(^{10}\) is a way of specifying the behavior of a function, and the name arises from its formal similarity to a legal contract. The *preconditions* define the conditions whose truth the caller guarantees to ensure before calling the function, and the *postconditions* define the conditions whose truth the function guarantees to establish by virtue of its execution. One of the purposes in this is to avoid redundant validity checks at each level in a stack of called functions.

The functions `count_vowels` and `count_vowelpairs` both receive as input argument a pointer to a C string. Neither function applies any validity check to the pointer, so there is an implicit precondition that the caller supply a valid pointer. To follow the rules of programming by contract we should make this an explicit precondition, albeit one that is so common and obvious that it hardly seems to need stating.

There are four possibilities for the pointer passed in as argument:

- It contains a valid address that is indeed the address of a valid C string (including the null string).
- It contains a valid address but the memory at that address does not comprise a valid C string.
- It contains a bit pattern that is not a valid address (e.g., it is outside the addressing range, or the memory is not readable).
- It contains the null pointer.
Our contract imposes the precondition that the pointer must be valid as specified in the first case above. There is no reasonable way to detect the second case, and for the third case it is usual to delegate detection (and handling) to the exception mechanism of the operating system.

The fourth case is the interesting one. Null is a valid value for a pointer, but it is (by definition) an invalid address that will generally cause an addressing exception if dereferenced. It is, however, trivial for the function to detect that a null pointer argument has been passed. Therefore, we can propose a general rule that makes life easier for callers of a function: If a pointer to a null object is a valid argument, a null pointer should also be a valid argument with the same meaning. In the specific examples we are studying here, we should change the contract by weakening the precondition to allow the first and fourth cases, and implement the function to immediately return the value 0 if a null pointer is passed as the argument. The purpose of this rule is to relieve callers of the need to make the test for a null pointer; for a language like C the gain is not obvious, but for functional style languages like Lisp the gain is significant.

Programming by contract places responsibility squarely on the shoulders of the caller for ensuring that specified preconditions for a function are met, and relieves the function of any responsibility for verifying that preconditions are met. As intended, this has the desirable effect of eliminating redundant validity checking. In real life, however, it also has an undesirable side effect: If the caller makes a mistake and does not in fact meet the preconditions, the function is likely to fail, and the cause of the failure may be difficult to track down. A pointer that is null or contains an invalid address is usually easy to diagnose; since an exception will be caused immediately an attempt is made to deference it, but other errors may cause failures far removed from the root cause. Therefore it is good practice for a function to perform reasonable validity checks for its preconditions, and in the C language the preferred technique is to use assertions. "Reasonable" is, of course, a slippery word; it would be reasonable, for example, to test whether an integer lies within a specified range, but not to test whether a binary search tree was indeed valid.

**Lessons**

The first thing that strikes me is how easy it is to write programs that are almost right, and how hard it is to write ones that are exactly right. The original bug crept in because of some unfortunate choices for C syntax made by the designers of the language and perpetuated by the standards committees (for example, the overloading of the `break` statement). But even apart from the specific error, the whole style of the original program is really bad. However, it's quite possible to get large programs with equally poor style to work, more or less. I know because I have seen them and even, I am ashamed to say, written them (before I knew any better).

It is possible to get away with programming in poor style because, unlike with physical artifacts, the cost of destructive testing of a program is very small. Therefore it is economically feasible to build a program by an
iterative process of construction and testing ("cut and try") in a way that would be out of the question for, say, an automobile or an aircraft. Just imagine what software development would be like if any processor exception caused the CPU to be destroyed.

Unfortunately, it is quite infeasible to perform an exhaustive test of even a small program, and so any program developed by cut and try is bound to contain errors, maybe even serious ones. It's perfectly reasonable to argue that because destructive testing of software is very cheap, the development process should take advantage of this by increasing the proportion of effort that goes into testing and decreasing the proportion that goes into design; but the current sad state of software development is due in large part to driving this argument way past its logical conclusion to an irrational extreme.

To explain this, I'll exaggerate only a trifle: The predominant mode of software development continues to be to slap together a mess of hastily written code, and expect to get it to work by performing tests to find errors and correcting those errors by ad hoc changes to the code. This mode of development is usually accompanied by a management process that anticipates that the defect fixing phase will comprise a very significant portion of the total schedule. Such a management style typically 1) uses defect counts -- and the rate at which defects are fixed -- as the predominant, if not the only, metrics of progress; and 2) lumps all defects of whatever nature into one defect tracking system, thus making them all somehow comparable. Testing is necessary and even reasonably effective, but our goal should be to develop code in such a way that we do not introduce errors in the first place.

When we are talking about component-based development and modularity, what we are really talking about is abstraction. What I see in the `switch` statement in the original piece of code is a total lack of abstraction. To me, the test of whether a given character is a vowel is a simple test of set membership, but I see no reflection of that insight in the original code, and I am forced to conclude that the programmer did not conceive of it in that way at all. Yet it's difficult to see how anyone can make effective use of component-based development without thinking in terms of programming abstractions.

Undoubtedly, part of the problem is the programming languages that are used. C is a fine language in its way, but its direct support for abstraction is limited to functions, macros, and arrays. Pascal provides sets (though they are limited in cardinality), and it would be entirely natural for a Pascal programmer to write `is_vowel` as "c in ['a', 'e', 'i', 'o', 'u']"; thus a Pascal programmer is more used to thinking in terms of set operations, which are inherently more abstract than tests and assignments. My use of `strchr` is, of course, nothing but a C implementation of a set membership test, but how many C programmers would see it that way?

It seems to me that we must encourage developers to think in a more abstract way about the programming task. I suspect, however, that this would be an uphill battle. Languages such as APL, Lisp, or Prolog that encourage or require a more abstract view of programming have never
been widely popular and are usually regarded as hard to learn. On the other hand, Python, which nicely combines aspects of procedural, object-oriented, and functional languages, has proved quite popular, so there are some grounds for optimism.

In my series of transformations on the original program, I was guided, as much as anything, by an essentially aesthetic sense, most clearly in my feeling that there was something wrong about the nonterminating for loop, and my description of the if statement as "ugly." There is a classic quote from the theoretical physicist Paul Dirac: "It seems that if one is working from the point of view of getting beauty in one's equations, and if one has really a sound insight, one is on a sure line of progress." Programs are very rarely beautiful in the Dirac sense\textsuperscript{14}, and we are usually limited to trying to get the ugliness out, rather than the beauty in, but the essence of Dirac's point still applies. We need to instill in developers an intuitive sense of what makes a program beautiful or ugly.

The final lesson I draw is that it is dangerous to make changes to code. It is so easy to unwittingly violate essential preconditions in the existing code, which are usually only implicit and often not completely known even by the original programmers. Furthermore, such violations are often not revealed by even quite thorough testing. The only other known countermeasures, which are not wholly effective, are extreme care on the part of the developers making changes, and thorough review, preferably by people intimately familiar with the existing code.

Notes

1 K&R style: This denotes the style of coding used by Kernighan and Ritchie in their now classic book \textit{The C Programming Language}, Prentice Hall, Inc., 1988, particularly their placement of braces. Tony Hoare: More properly Sir Anthony Hoare. Perhaps best known for his invention of the Quicksort algorithm, he is now an emeritus professor of computing at Oxford University, having made many important contributions to the theory of programming over the course of his long and distinguished career.

2 "lint" is a venerable Unix utility program that finds errors by static examination of source code. Its name is derived, in the typical Unix fashion that some find irritating, from the notion of removing the "fluff" from code.

3 Because \texttt{continue} is a sort of "do nothing" statement, rather similar to an \texttt{if-else} statement with a null \texttt{if} part; and, of course, \texttt{continue}, like \texttt{break}, is a clandestine \texttt{goto}. In production code I occasionally find a need for \texttt{break}, but I have never used \texttt{continue}.

4 I will readily agree that this is an \textit{extremely} tiny example, but the string functions of the C run-time library really do form a component, albeit within the strictly procedural paradigm.

5 Admittedly, this is a rather farfetched and contrived example, but it does illustrate the principle.

6 An 8-bit character has 256 possible values, 128 of which are valid ASCII characters. However, C often promotes a \texttt{char} to an \texttt{int}, and in fact will do so in the course of passing the \texttt{char} argument, so the claim that this function may be exhaustively tested is perhaps a little optimistic.
We checked with our client and that is really how it is to be defined; if you think that once a pair has been identified both members of the pair should be removed from further consideration that's fine, but it's not what the client wants.

In fact, I noticed this bug as I was thinking about the succession of code changes described here, and tested the code only to confirm my deduction of what would happen.

I agree that the implementation using `strchr` is in fact incorrect as it stands, and should be corrected so as not to classify the null character as a vowel. Note that when we originally did the refactoring, `count_vowels` in fact guaranteed that `is_vowel` would never be called with the null character as argument, so the implicit precondition could never be violated. Indeed, we did not consider that aspect of the matter at the time. Also, a C programmer is predisposed never to think of the null character as an actual character and so is unlikely to envision null being passed as an argument. That's why oversights of this sort happen.

Bertrand Meyer has been the most prolific author on this topic. See, for example, his well-known book *Object-Oriented Software Construction* and the paper *Applying "Design by Contract".*

The best way to handle exceptional conditions is a large topic in its own right that I do not have space to cover here.

Given the context, I am sure the code fragment as presented by Gimpel was abstracted from a larger piece of code and stripped to its bare essentials. I'm not sure whether this would make its style better or worse than that of the original.

This hints at the long-running and contentious argument about the utility of formal methods, an argument that's far too complicated to cover here. Close readers may be able to discern where my sympathies lie. Relevant material can be found in:


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

by Cindy Van Epps
RUP Process Implementation Expert

I might as well confess the whole truth:

- I am in the Transition phase of my child-rearing project. In retrospect, I should have had a deployment plan earlier in the project, as the costs were much higher than I had ever anticipated.

- I recently used RUP to update my guest bathroom; that is, I performed the architecturally significant and riskiest tasks first before investing in the "nice-to-haves."

- I have asked my housemate to supply the use-case specifications for our six remote controls. I have no trouble with basic flow of events but those alternative flows are something else.

Yes, I am into RUPý, the Rational Unified Process,ý to an extreme. Perhaps it comes from all my years as a RUP process implementation expert. Through this experience, I learned not only how to apply RUP to challenges in my daily life, but also that the best way to clear up misunderstandings about the process is through real-world examples and analogies. Below are some favorite "RUP lessons" I’d like to share with you, as much for your entertainment as for your enlightenment. Feel free to pass them on to your friends at geek parties.

Horse Sense RUP

Some people are so smitten with the concepts embodied in RUP that they look for ways to apply the methodology to any area devoid of clearly defined process. (I've had RUP enthusiasts tell me about corRUPted
projects, interRUPted projects, and RUPdates on their process. Clearly, the acronym, just like the process, can be taken to an extreme.) Although this enthusiasm is admirable, it should be tempered with the fundamental recognition that RUP is a process framework, designed to be tailored. Although you can apply RUP principles to real life situations, in most cases applying the whole process would have a negative effect. I tried to define use cases for child rearing, the most notable being "Support Mother in Old Age." However, I recognized that once I deployed the "system" I would have no control over its behavior.

Similarly, in a software development project, it is imperative that RUP users understand the goals of the process and determine whether each activity, artifact, and process principle is appropriate for their project.

I recently worked with a project manager who wanted to know if it was acceptable to combine several vision documents into one. "Does it make sense to do that?" I asked, and he said, "Of course, that's why I suggested it."

"And does it align with the goals of why we create a vision artifact?" I asked. "Why yes," he replied, after thinking about it for a moment.

"Then that is exactly what you should do!"

The logic in this conversation was straightforward enough, but we must have repeated five or six variations of it as we walked through the various artifacts required by our default development case.

For example: "My vision is recorded in Microsoft PowerPoint. Do I need to convert it to Word format because the RUP vision document template is in Word?" he asked. "Why on earth would you do that?" I replied. "You can enter whatever features you've defined directly into Rational RequisitePro and then link to the PowerPoint vision file."

**One from Column A and Two from Column B**

Some folks buy RUP as a way to outsource the process stuff. Then they find out that there is this tailoring thing they have to do. It is easy to understand the culture shock they feel if they have been using no process at all.

Here's how I see it. RUP is a bit like the menu at Kim Son, a popular Vietnamese/Chinese restaurant in downtown Houston. This menu is longer than some novels I have read, but you get to make your selections based on personal taste, dietary requirements, and items you have enjoyed before. The "process expert" is your waitress, who walks you through the appetizers and entrees, helping you understand what each item is and how it might meet your gastronomic needs. Often, she will help you steer clear of items that are not appropriate for you.

Just like Kim Son, RUP builds more variety, experience, and risk-avoidance into its selections than you could ever make at home. As a consultant, I'm often in the waitress (process expert) role. Recently, I spent one day
paring RUP down to a handful of artifact templates, a set of guidelines, and a process description for a service pack development project. The result was very powerful because it was grounded in exactly the same terminology and concepts as a full-blown development process; I had simply eliminated all that we chose not to do in developing a service pack. You wouldn't order everything on Kim Son's menu, would you?

Don't Let Those Phases Faze You

The names of RUP phases have created some confusion, particularly when it comes to the distinction between Elaboration and Construction. Often people think, "If we are building code, we are constructing it, so we must be in the Construction phase." This seems to make sense; however, it overlooks a fundamental principle of RUP: that you should build code in every iteration, including those in Elaboration as well as Construction. But isn't the code you build in Elaboration just prototype code? you may ask. Ah, there's a slippery slope. RUP says we should build and test the code as if it were for delivery, but we should also honestly assess its quality, throw away the parts that aren't good enough, and then redo those parts.

Well, this recommendation makes sense to most people, but I confess that I am a simple girl from a small town in Texas, and we like to save our squinting for the blinding light of summer rather than for pondering advanced concepts. Therefore, I have developed the Bubba Version of RUP in which the Inception, Elaboration, Construction, and Transition phases of RUP translate to: Git Money, Git Smart, Git Done, and Git Out.

We like this terminology because it helps us evaluate at the end of Elaboration: Are we smart enough to know within a reasonable tolerance when we can "git done"? The things we need to be smart about relate to architecture and risk. Do we grasp the complexity of installing and configuring our product and all of the other products on which it depends? Do we know how long it will take to set up test environments and document the procedures for our customers? The core development team may not care about these things, as they are busy creating the required functionality. However, it is a great risk to the project if we do not know the size of the beast until it is too late to take back our agreements on delivery. We should plan to create a test environment, not from our "path of least resistance" install and configuration option, but from a customer's perspective. For example, we should try out the installation media and configuration instructions we plan to provide the customer and be sure that the tests are conducted in an environment very much like the customers' environment.

If the Patient Can't Breathe, Don't Pull the Plug

What happens at a typical phase review? If your projects are like ours, then your phase reviews may go something like this: You create a set of artifacts, distribute them for review, and make a presentation. Then, when you ask the executive managers at the meeting for a go/no go decision, they all nod "yes" without hesitation. That is because everyone interprets "no go" as "no good" -- and they fear the project might come to a screeching halt if they question anything. As a result, we send projects
into Elaboration without a clearly defined scope and then find ourselves overcome by risks in Construction. Surely when this happens, the process has failed us.

A project is like a patient in the hospital. Asking whether you're ready to move from Elaboration to Construction is really like saying, "We think the patient might be ready to move from Intensive Care to a regular room, but will she survive if we pull the plug on her respirator?" If the answer is "no," then the next question is whether you should keep her in intensive care. And if so, what in particular should you monitor to determine when she's ready to move? And when is it safe to give the "family" a move date?

Recently, I led the best Elaboration review I'd ever experienced. The development team had proved the architecturally significant use cases and reduced every risk over which they had control. However, the project still had two significant risks in the form of unmet resource needs -- one for a technical writer to create user documentation and the other for a developer to create the installation kit. Clearly, this commercial product could not be released (or, arguably, tested) until these needs were met. The executive managers responsible for staffing the project became the roadblock to declaring the patient "out of the woods."

Exposing this at the meeting gave us leverage with these managers and allowed us to hold them accountable for supporting the project. We agreed to let the development team continue with the set of activities planned for the Construction phase but refused to commit to any dates with product management, sales, or customers until those critical resources were on board and we could predict when their respective tasks would be completed. Although the review was painful for the development team because they did not "pass" Elaboration, they quickly realized the power of maintaining a distinction between phases and adhering to RUP's phase boundary criteria.

**Tide Yourself Over**

One day I received a desperate phone call from one of the project managers with whom I had consulted on the use of RUP. "We have been doing well with the iterative approach," she said, "but we had a glitch and now the team is in a frenzy. I don't know what to do! Can you help?" Well, they had paid a lot of money for a consultant, so naturally they expected me to have all of the answers at a moment's notice.

"OK, no worries. What happened?" I asked calmly.

"We were testing right at the end of the last Elaboration iteration and discovered a pretty big flaw in one aspect of our design. The development cannot continue until we fix it," she explained.

"Easy. Do the work in the next iteration and document what happened." Boy, this consulting gig is easy after all, I thought.

But then she came back with: "No! No! You don't understand!" (Judging
from her stress level, I apparently had not understood.) "Our iterations are two months long. We time-box them and try to keep them the same length, just like YOU told us to do. But two months is just too long to wait to report to management that we fixed this flaw. The team will feel they have a cloud hanging over them for too long. We need to fix the problem and be done."

"Mmmmm. OK. Well . . ." I mumbled, trying to summon the spirit of Philippe Kruchten to help me out. And then it came to me -- what they needed was a tweener!

You see, an Australian co-worker of mine had acquainted me with the beer drinking ritual Down Under. Each person in your group, in turn, buys a round of brew for everyone. But if you finish your serving before everyone else is ready for the next round, you buy yourself a "tweener" to hold you over. It's a beer in between the formal rounds.

I explained this concept to my panicked friend and suggested that a tweener iteration would not only hold over her team until the next formal iteration, but that it would also introduce a little levity to help the team overcome their anxiety. As in all instances of applying the process, I said, you think about the purpose behind the process goals and then do what makes sense in order to meet them.

**Take Time to Save Time**

The last thing I want to share with you is the most common abuse of the iterative approach. Actually, it is more a means of diluting the real strength of the approach than an abuse. Experience has taught me that iterations offer two great advantages: First, they provide time-boxing to help pace the project and allow evaluation and course correction at specified intervals; and second, they allow you to test the capability built in each iteration with the same rigor as you would test the final product. This little lesson focuses only on the first advantage.

Typically, instead of leveraging that first advantage, we make the mistake of allowing iterations to become content-based rather than time-based. As a result, our iterations stretch further and further out as we frantically attempt to complete the planned work without stopping to assess what is going wrong and reset the time frame. When a team knows that on a specified date they will all sit down face-to-face and discuss frankly what did and didn't get done, you can bet that the individual members won't want to be the ones with tasks outstanding.

As a product development manager, I once sat with my team at the end of an iteration and said, "OK guys, we didn't get everything done that we thought we could. What happened?" They mumbled. So I asked, "Do we just need to work more hours?" Ah-ha! That got their attention. "No", they said. "In fact, if you work us more hours than we did on this iteration, we'll quit." Hmmmm, I said to myself. Let's not push in that direction. But I persisted.

"Then why did we get so little done compared to what we had planned?" I
asked. And then the truth came out. We'd had to spend a great deal of time and effort on re-working some poor-quality components we'd received from the common architecture team. Each developer had experienced a piece of this problem, but it wasn't until we stepped back and looked at the issue together that we were able to see the true magnitude of it.

However, once we had pieced together this information, we were able to define acceptance criteria for the components and put the responsibility back on the common architecture team, where it belonged. My team sighed in relief. It was right then and there that I bonded forever with the iterative process. Had we not done an iteration review and had instead continued heads-down on the same path, the problems with those bad components would only have gotten worse and eventually overtaken the project. Had I waited until the planned work was completed to assess what was slowing us up, it would have been too late.

**Happy Paths to You**

I hope that my anthology of stories and analogies has helped you on your path to understanding and using RUP more wisely. Now, if I could just align the stakeholder needs of my daughter's wedding with my resources.

---

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

Copyright [Rational Software](https://www.rational.com) 2003 | [Privacy/Legal Information](https://www.rational.com/privacy.html)
The Privilege and Responsibility of Software Development

by Grady Booch
Chief Scientist
Rational Software
IBM Software Group

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

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

These are the days of living as a netslave\(^1\), a microserf\(^2\). After abiding such days during which we labor to build artifacts that live in the realm of nanoseconds, sometimes we long for a life with "no unit of time shorter than a season."\(^3\)

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

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

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

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

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

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

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

(The article is excerpted from the forthcoming third edition of Object-Oriented Analysis and Design with Applications).
Notes


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