the Rational edge
e-zine for the rational community

Cover Story

e-Business: new challenges for testing

august 2002

enter issue | mission statement | archives | subscriptions |

Photo: Copyright 2002 Andy Lampitt

Copyright Rational Software 2002

subscribe
contact us
submit an article
rational.com
issue contents
archives
mission statement
editorial staff

Editor's Notes:

Every summer, a couple of weeks before school starts, I have the same dream. I walk into a classroom, and the teacher announces "There's a test today that counts for most of your grade." As the subject is always something I never studied -- maybe Chinese or nuclear physics -- I fall into a dead panic and wake up in a cold sweat.

When I joined the software development world, it was delightful to discover that, here, testing is nothing like those dreadful school events that still have the power to haunt our dreams. Ideally, testing is an overwhelmingly positive force for project teams. It's an activity you actually *look forward* to because it enables you to clean up your mistakes and move ahead. And it's an endeavor that deserves more recognition, resources, and rewards.

This month, *The Edge* examines the positive testing force from several perspectives. An interview with industry testing experts Theresa Lanowitz of Gartner, Hung Nguyen of LogiGear, and Rational's own Sam Guckenheimer examines the future of testing and its increasing importance in developing complex, multi-tier systems. Then, Sam's interview with testing guru Cem Kaner (second installment) shows us why real world experience is a hallmark of the best students enrolled in Kaner's unique testing curriculum at Florida Institute of Technology. Finally, we learn that unit testing was a key to infusing best practices into the dysfunctional dotcom where Rational's Raj Kesarapalli worked for a while during the bubble.

There's another real-world tale in this issue from Clair Cates -- about how SAS migrated Rational PurifyPlus® from Unix (story in last month's *Edge*) to Windows NT. And practical advice from Philippe Kruchten on how to leverage the Rational Unified Process® for ISO 12207 compliance. The doctor is in -- Dr. Use Case, that is -- diagnosing the relationship between function points and use cases. To round out our offerings, there's an article on "Ending Requirements Chaos," a review and sample chapter of *Developing Enterprise Java Applications with J2EE and UML* by two Rational professionals, and a preview of a forthcoming book on documenting software architectures.

OK. I'm off to the Rational User Conference -- come visit *The Rational Edge* in the Solution Center! And if you'll be doing any testing over the coming weeks, don't worry: The force will be with you.

Marlene Ellin
Senior Editor

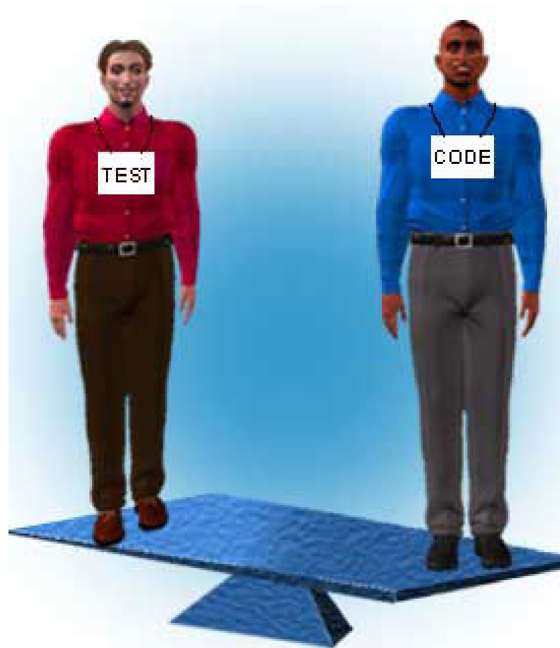
Copyright Rational Software 2002 | Privacy/Legal Information

Q&A with Industry Experts How Are e-Business Trends Impacting Testers and Testing Teams?

Part I: Challenges for Testers

by **Jack Wilber**
Rational Edge Writer

In the April and May issues of The Rational Edge*, industry analysts shared their views on how e-business trends are affecting both individual developers and development teams. In this issue, we begin another two-part series focusing on how these trends are affecting the testing community. Developers aside, of all the professionals that comprise an extended development team -- project managers, analysts, testers -- perhaps none feel the impact of current trends more than testers. They are the people tasked with ensuring the quality of complex applications with very limited resources and in the face of rapidly approaching project deadlines.*



*For insights and opinions, I turned to a panel of three respected testing experts and analysts: **Theresa Lanowitz**, Research Director at Gartner; **Hung Nguyen**, President and CEO of LogiGear® Corporation; and Rational's own **Sam Guckenheimer**, Senior Director of Technology for Automated Software Quality. In this first installment, they share their thoughts on the challenges testers are facing and the technologies, skills, and strategies needed to meet them. Part II will focus on how changes in architecture have affected testing and automated testing tools.*

Jack Wilber for *The Rational Edge*: Let's begin by talking about skills. In the last two to three years, how has the explosive growth

in distributed applications affected the skills and domain knowledge that testers need to be effective?

Hung Nguyen, LogiGear: I think the effect of this growth has been tremendous. In the past, everything in the testing environment was very self-contained: You got a deliverable, you ran the installation program, and you started testing. But when you go to a more distributed, or e-business, model, there are two main problems for testers.

First, on the technology side, everything has changed. You don't have control over your environment because your system might have components distributed all over the place, some that your team developed, and some third-party components. So just trying to understand the environment and figuring out how to test effectively within it is a big technical challenge.

Second, on the business side the rules have changed as well. In the old days, users bought a package, installed it, and used it. Now you have users who might buy a package, or they might just use your e-business infrastructure to conduct business transactions. So on the business side, testers need a lot of education to be effective. For example, consider performance, just one dimension of testing. In this new environment, the tester needs to understand non-functional issues such as, What is performance? How do I come up with a "reasonable" response time and test for it? That is something they don't always see in the functional spec. Another area of concern is security testing. Testers need to ask, How do I know that my users are protected or that the business is protected? Right now, that's a gray area for testing, because few testers know how to do it effectively.

We have begun to realize that in order to test effectively, you need technical skills. Because the field is not mature enough, we still have non-technical people doing testing. Now, there is nothing wrong with that at the business logic and user level. But you also need to fill the gap on the technology side. Until everyone understands that we need skilled people to do the job, I think that testers will, unfortunately, continue to be underdeveloped, and earn less on average. Ideally, the more skilled testers would know as much about the technology as a developer and would therefore deserve to be paid comparably or commensurate to their ability -- and management needs to understand this. If the salary structure shifts and there is a budget for bringing more talented people into the mix, then more developers will be interested in becoming test engineers.

Theresa Lanowitz, Gartner: Even though we have seen explosive growth in distributed applications, we have not seen explosive growth in skills, either for developers or for test engineers. With many distributed applications, the application *is* the business. Suddenly, the enterprise has all these customer-facing applications, and the IT organization in the traditional enterprise is now responsible for creating revenue-producing products, not just applications.

But skills have not grown; in fact I'd argue that they have diminished, because as more and more enterprises rushed to create these customer-

facing, revenue-driving products, they could not find enough skilled people, so they hired inexperienced people. We saw a lot of that back in 1999 and 2000, which accounted for a lot of high-profile Web site failures. Around the same time, you would hear a lot of hype about testing tools that were so easy, you didn't have to be technical to use them. But I think that is the wrong message to send; you really *do* need technical skills to do what we're expecting testers to do with these applications.

And distributed applications are only getting more complex. From an evolutionary perspective, with mainframe applications you knew who the users were, you knew what the architecture was. Then there was client-server, and then the Internet world, and now you have wireless applications. And in order to handle the new complexity on the testing side, you want skilled quality engineers -- people who understand process and what quality engineering is all about.

Companies know this, but few act on it. Through surveys, we know that getting people with solid technical skills is a top concern for enterprises. However, one of the things they're *least* likely to spend money on is training. So it's a constant conundrum.

Another problem is that testing is often the first thing a development organization cuts when the budget needs to be pared back. Also, testers may be perceived as entry-level people, or testing regarded as a position you accept first, before moving on to become a developer. As Hung pointed out, testers are really not given the professional equity and respect that they deserve. So organizations constantly have the same problems over and over again because they don't do enough to keep a core group of testers with institutional knowledge.

Sam Guckenheimer, Rational Software: So what we're saying is that once upon a time people believed that you could test without having deep technical knowledge of the software under test, but when you are looking at a distributed application -- on the Web in particular -- that assumption breaks down. Hung's book[1] on the subject of testing Web-based applications is excellent on this point. Testers need to understand how the technology affects the kinds of errors and risks that they can see. They need an understanding of technology issues -- such as the deployment topology -- as well as understanding of and the kinds of errors inherent in the technologies they're examining. Even understanding details like the difference between bean-managed and container-managed persistence on an application server -- all these issues affect what kinds of faults you are going to find. Today, testers need to understand the technology and the domain as well as generic testing techniques.

For example, suppose you see an error message that says "404 - Page not found" in the browser. That error might be caused by a broken link, or it might be because some service has become unavailable. A good tester will not only suspect the unavailable service, but will also be able to confirm his suspicion -- for example, by looking at other pages that depend on that service. This is a critical technique for isolating a bug.

Another skill that has gotten a fair amount of attention recently is the ability to be a good explorer. Historically, a lot of what was described as

testing was very scripted and planned, but in reality good testers are good explorers. They see things that may be hints, and they know how to follow up on them. It may be something as simple as a page that takes surprisingly long to load. A good tester will ask, Why would that be? and knows what paths to go down. James Bach has written the best material about exploratory testing and has the best exercises on the subject. I think it certainly is a critical skill, and one that a testing team needs to have.

JW: For years, as organizations have tried to develop software "faster, better, cheaper," testers have been there to ensure the "better" dimension. Is there now more pressure to help with the "faster" and "cheaper" dimensions?

TL: What we're really talking about is the age-old triangle of choices: budget, schedule, or quality. Your question asserts that testers have been there to ensure the "better"; but have they really been able to do that? Consider the role that test engineers have been forced into. In traditional waterfall development, testing occurs only during a brief period before the application goes live. The test engineer really never has much input into developing either the use cases or the test cases. And if the schedule slips during engineering, it's the test engineer who feels it on the back end. I would argue that testers have *not* always been able to ensure the "better."

To do so, they really need to be the customer's advocate. And I don't think they have been given the respect, time, tools, or even the right cultural settings for this. Organizations are always more concerned with faster and cheaper than better, and it usually takes a catastrophic or near-catastrophic event for most to realize that their development abilities were not as good as they had assumed. We've seen this over and over again, with all the high-profile outages and site failures we've had over the past few years.

Building a high-quality application, within budget and on time, takes a very disciplined organization -- in terms of both management and process. And that kind of culture is not yet pervasive in the industry.

What are the cornerstones for that culture? Skilled professionals; processes and procedures that you can document and repeat; strong tools and services. Often people think that a tool is going to be a panacea, but that's not the case. If you are focused on delivering faster, then you are probably sacrificing quality and maybe exceeding your budget as well. This is what we saw during the dotcom boom. The sad truth is that you are not really getting to market much faster either, because over time, new development costs will get out of control, surpass maintenance costs, and prevent you from getting to market with the right product at the right time.

HN: I think this issue can be traced to a lack of budget for testing groups. Management always wants to build better quality products -- I have not met one who says otherwise -- and that requires better process, better development methodology, and better testing strategy. Yet if you look at most business budgets, there is no line in there for testing; it all goes to R&D or development. So there's no visibility for testing within the

organization and no budget at the business strategy and management level, but testers still have all the responsibility of making sure the system works.

Another problem is that there are really very few reliable metrics to show how much you've done before, so there's no traceability you can use to determine whether you are doing better or worse. If you have a huge failure in the product, where do all the fingers point? At first they point at testing, but eventually the blame spreads all over the place, and no one is accountable for one single thing. I think that is the number one problem from a management perspective. If you want "better," then you have to increase visibility for testing and quality engineering, and you can effectively do that through a budget. Team up testing folks with development staff to figure out how to get the job done. The testing budget can be a percentage of the development budget or, preferably, of the business budget; the actual amount is up for debate but it has to be something. That is how we allocate funds for marketing, sales, and R&D. So why not testing?

Developing "cheaper" is not easy, either. Tools can certainly help, and so can process. So can education, particularly on how to use the tools effectively. Actually, this goes back to the skills issue we just talked about. Finding good testing education is a problem; serious, skill-based software-testing curriculum is limited. Off the top of my head, the only example of a good program available today in the U.S. is the one offered by the Florida Institute of Technology, where Cem Kaner and James Whittaker teach. Programs delivered by the University of California at Berkeley and Santa Cruz Extension, LogiGear and SQE are also examples of limited useful course offerings on software testing. Other than that, I think there is a huge skills gap, and adding more education at the college level would be good step. Companies like Rational are constantly developing tools that support new technologies, but testers need to understand them in a larger context. Tools are just a means to solve a problem. To use them effectively, I need to know I have a problem, how the problem is defined, and that there are a number of ways to solve it. That is the kind of education I am talking about.

SG: The key to faster and cheaper is an iterative development process that brings testing forward in the development cycle, making it possible to find defects when they are cheaper and easier to repair. However, I don't think testers are well trained to work in iterative processes. Nor are project managers well trained to consider the testing role; that's why we've added a lot and are continuing to extend the Rational Unified Process and Rational University training to show how testers can work iteratively.

But even if you're not doing iterative development -- if you're doing waterfall -- the same concept applies: To save time and money, test basics first. You want to validate the spec and do function-level testing from simple tests first, in early iterations, and build up to complex scenarios and configuration testing and multi-variant combinations in later ones. Ideally, you build up a growing repertory of automated tests, though you also need to refactor them as you go. For example, automating tests for interface contracts is absolutely critical, and those should be run in

regression all the time. But in user scenarios that may change based on usability, test feedback, or design changes, you also need to be sure that you're clear about what you're automating and how you're going to refactor the tests when the application under test changes.

What's really important is to understand the power of testing at many levels and not to think of testing just as something you do from the GUI on a finished system. As testers, we need to think carefully about unit testing and interaction testing, as well as what kinds of tests are appropriate and where.

JW: Let's talk more about process. What changes have there been in the way testers work with the rest of the extended development team? Agile development processes have promoted awareness of test-first design and unit testing. Are test teams now getting more involved in code-level and model-driven testing?

SG: Let's take these one at a time, starting with the way testers work with the extended development team. I am a firm believer that testers need to be closer to developers; they should be working in a tight loop, iteration by iteration. I think that about half the market works that way now. The other half thinks that testers should be independent, and a lot of them outsource their testing. In my opinion, you lose half the benefit of testing when you do that. You get people to find bugs, but you do not create a process based on continuous and exploratory learning. If your testers are working close to your developers, then all can learn as they go, and they can both contribute to making a much better product. If you throw testing over the wall to outsourced testers or a test team outside your project, then people can find mechanical bugs and report them back to you, but you have limited ability to really evolve the product or process in an iterative way.

This leads into the "agile development processes" part of your question. The notion of evolutionary development is fundamental to Extreme Programming (XP), which has grown into the agile movement. Testing in agile development is not well defined, and there are many views on what it might be. I tend to line up with the definition that Brian Marick and Bret Pettichord have been working on, which is based on six principles -- actually they call them slogans -- that capture practices. One is that you develop tests as the embodiment of design specifications; essentially, the tests *are* the design specifications. So what the Rational Unified Process calls use-case realizations, they accomplish through tests. At the same time you do exploratory testing on the software that is built, and you continually iterate and refactor, focusing hard on design for testability. I think these are all great practices, and a lot of testers are starting to pay attention to them.

Do I see testers getting more involved in code level testing? Here, the nomenclature is a bit confusing. People who are called testers in one organization are called developers in another, and vice versa. In most organizations, testers do not get involved in testing directly from source code unless they and the developers are working in pairs. I think that is appropriate, because developers should take responsibility for the quality of the source code. We've known for a long time that the best person to

test the source is the person who wrote it, and Rational offers strong tools to support developer testing activities.

Model-driven testing is another issue. Models offer a great way to document a system, visualize system behavior, and communicate shared work across the team in an accessible way that also reduces complexity. Interest in using models for testing is growing exponentially, and that is a fantastic trend. Model-driven testing has a few meanings. One is that models can be developed specifically for testing, separate from the code development, as a way of generating high-volume tests. (That is the meaning Harry Robinson of Microsoft uses on the Web site he maintains: www.model-based-testing.org.)

From Rational's point of view, on the other hand, model-driven testing means that the model depicts the software under test -- its structure and behavior. The same model captures the definition of what to test and can also capture test results. We are actively contributing to the development of model-driven testing. Rational® Test RealTime, for example, shows the behavior of the software under test in a UML sequence diagram. Our concept of model-driven development is consistent with the work that's being done in the OMG[2] working group on a test profile for UML. Once the UML test profile is adopted by OMG, I predict that we will see an explosion in the use of models for visualizing results and defining tests.

We haven't talked yet about the way testers work with analysts. There has always been a relationship between these two roles, and even in the most "waterfallian" of processes (e.g., IEEE 829), people understand about testing requirements. The evolution of modeling into an analysis and development practice tied analysts and developers together, because it enabled developers to translate requirements into designs with progressively greater levels of specification. On the flip side, it also allowed them to visualize these designs at progressively higher levels of abstraction. Testers weren't originally considered in that loop, but all of the same benefits apply. And indeed, everyone wins when they realize models can not only describe intent, but also capture actual system behavior. Frequently people skimp on use-case realizations in a model, but if teams could apply the same kind of roundtrip engineering to behavior that they apply to structure, that would change. And that is exactly where we are going. If you look at Rational Test RealTime, you'll see that is exactly the kind of value you get from capturing system behavior in a sequence diagram.

TL: Process, including a test-early approach, is critical to the success of any organization, but many still haven't realized that. Two or three years ago, Gartner heard a lot of organizations saying, "We developed this application for [fit-in-your-favorite-vertical-industry-here], and we want to take it commercial. Our plan is to sell it to others in the industry and spin ourselves off from the parent organization." But after we had a conversation with them on what it takes to be a commercial software company, they would retreat. We never saw any spin-off.

But in fact, enterprises do need to be able and willing to behave much more like commercial software companies. They need to understand the build cycle, requirements, and schedules; they need product managers

who can serve as liaisons with the engineering group, and so on. So far, we have not seen enterprise organizations *en masse* adopting this more structured behavior.

To make such a change, you need a culture that supports it. Practitioners often tell me that management does not want process because they think it will take up too much time. To have a good process, you have to understand what it should be, and keep the management and the philosophy intact long enough to get through the initial stages of adoption. You also have to keep in mind that the end goal is to deliver a high-quality application, on time and within budget, that everyone thinks about as a product. Unless the emphasis on quality is infused and travels from the top down, the organization will tend to run in a chaotic or reactive mode.

It's also important to remember that coding is really only a small part of any development project. Identifying the correct architecture, getting the process in place, making sure you are following the standards that have been established for the organization -- those are the key things.

As for code-level testing, developers are now writing more unit tests, and that's a positive thing. Some really good tools have come on the market to help developers create unit tests. However, I still believe that, over time, testers need to become more technical, and the organization needs to invest more in training and keeping testers. Then, as their skill sets keep growing, so will parity and respect for the testing function within the organization. And testers will most definitely be more involved in code-level testing.

And for model-driven testing, the UML is a great thing for that. Once you have the use cases written, you have the test cases written. And it's a very positive thing if you can integrate a good solid process all the way through your software development lifecycle.

HN: Certainly the degree to which testers are involved with the rest of the development team varies greatly by company. One organization might have test engineers that are not very technical, but they have a great process and are able to get the testing, development, and business teams together to talk about requirements and features and document it all. But industry-wide, there's definitely a shift toward getting testers involved earlier in the process and working more with business analysts and the development team.

It's good to have the development team thinking about testability of their code at the source level, and thinking about unit testing. But if you look at where testing takes place -- at the requirements level, source level, interface level, component level, and system level for integration tests -- where testers are not doing well is at the source, interface, and component levels. I see some good collaboration at the interface (API) level, but at the source level, it is still a developer thing; testers have yet to understand how to be useful in that environment.

For example, Rational® Purify® is a dynamic tool that developers often use, and that is good. But to have better test coverage you need to

execute more of the code, and developers don't have time to do that. So it would be wise to integrate the testing team into that process and have testers use Rational Purify during their tests as well. Likewise, it makes sense to have the developers do unit testing in one pass, and then let testers do it in another pass. We need to close the gap between development and testing people, although I still see code-level testing as mainly a developer activity, probably because of the lack of education in testing. But testers can just run the tests, log all the errors, and send the results to the developer; they don't even need to be able to interpret the results.

I believe model-driven testing has a very important role in test design and analysis. For example, Rational® QualityArchitect can generate tests based on models and dependencies. And once you have an error, you can actually use the model to shorten the path to deduce the failure. So model-driven testing is key to test design and generation, and as a knowledge base for automating failure analysis and pinpointing problems.

JW: People often talk about process as a means to reduce software failures. Much has been written about the increasing cost of failure associated with public-facing e-business sites. Has this business change affected testing practice in significant ways?

TL: Absolutely. When it comes to e-business, failure is not just a matter of people not being able to use the software; it is a matter of public image. Because you have non-technical people using these applications, and because the applications are moving toward ubiquity, the software has to be foolproof. With a less sophisticated audience you only get one chance. If they try to use something -- like a Web service -- that doesn't perform or doesn't work at all, then they'll just abandon it and move on to another site. And that speaks directly to the need to build higher quality into things like Web services.

SG: I think the increasing cost of failure has raised management awareness of the importance of testing and quality. Fortunately, we've moved beyond the practices of some former dotcoms that ignored quality and focused entirely on speed.
Management is more savvy and more careful since we had those highly visible dotcom failures.

HN: I don't think the high cost of failure is really new; we've faced it before. It does affect testers; it puts pressure on us to be more effective in finding errors. But the problem is more closely related to the quality assurance process. How do we implement a quality process that capitalizes on people and technology? How do we get QA and development to work together to develop better practices?

In the context of e-business failures, the way we do testing now is different from the way we used to do it. Now, we don't stop when the product is released; we test on an ongoing basis. That is why a new *monitoring* market segment has opened up, and we're putting mechanics in place to alert us if there is a failure.

Also, the public is better educated now. They understand that if they pay

for it, then you have to give them good stuff. They have options; there is so much competition that they are just going to walk if you don't give them a good, quality product.

A very positive result of these failures is that management has begun viewing quality issues in terms of dollars and cents. They are telling their development organizations, "I don't care if you call it a high-quality or low-quality product. If it shuts my site down for two minutes, it costs me a million dollars, and I don't want that to happen. So, you go back and figure out how to prevent that from happening." And management also knows that if they give the testing group a decent budget, then the testing group can be held accountable. You want to put testing at the top of the list when you create the yearly budget because that is one of the primary avenues to get quality. In the end, that will give testers authority, responsibility, and accountability.

Stay tuned for Part II of this series in next month's issue!

Notes

[1] Hung Nguyen, *Testing Applications on the Web* (Wiley 2000)

[2] Object Management Group: http://www.omg.org/

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

Copyright Rational Software 2002 | Privacy/Legal Information

Copyright Rational Software 2002

An Interview with Cem Kaner, Software Testing Authority

Part II: How to Educate and Train Testers

by **Sam Guckenheimer**
Senior Director of Technology for Automated Test
Rational Software

***Cem Kaner**, Ph.D. J.D., is Professor of Computer Sciences at Florida Institute of Technology. He is perhaps the world's most prolific and widely read author, consultant, educator, and attorney in the field of software testing.*

In Part I of this interview, featured last month, I discussed with Cem Kaner, Professor of Computer Sciences at Florida Institute of Technology, his notion of context-driven testing and the course development he has done for Rational over the past year. As perhaps the world's most prolific and widely read author, consultant, educator, and attorney in the field of software testing, Cem concludes this interview with his insights on testing education, its relationship to consulting practice, and his views on "agile" software development.

Guckenheimer: Now that you have been a professor at Florida Tech for two years, what have you learned about educating software testers? Is there anything different now about the way you train software testers in a university setting compared to a commercial setting?

Kaner: As a university professor, I have two luxuries now that I didn't have when teaching in an industrial setting. First, I can actually give my students tests, and they are motivated to take and pass them. I can also

give them homework and evaluations. In an industrial course, you just can't do that. Even if you have a very light test at the end, it's not the same thing as giving someone an assignment that will require a week of intensive work with a colleague. Through giving and grading student assignments, I've learned that some of the concepts I thought were very clear are very confusing to people with little testing experience. For example, looking at a situation and assessing "What are the boundary conditions in this case?" takes a remarkable amount of practice -- at least three to four assignments before most students get really good at it. They need practice, via ungraded or lightly graded assignments, dealing with similar problems time after time. You can talk about it over and over, but the main concept has to spark in the student's head so they go, "Oh, I get it." That typically only happens with practice.

A lot of what we're doing now at Florida Tech is drafting self-paced, self-answering homework questions. For example, I give you a data entry field; you analyze this field and come up with a boundary case, and then I'll give you what our analysis of the same field was. Then we will give you a word problem that asks you to figure out what the field is, or what the variable is, that you're studying, and then we will extend it.

Consider the way we teach boundary analysis. A student enters the highest number possible for a given field, then enters the highest number plus one and tests both of those. What's the reason for these specific values? Historically, we know that the program is a little more likely to fail under these conditions than with a valid number that is big but not the biggest, or with an invalid number that is too big, but not right at the edge. So, as in this example, we teach a theory of error. And what we're doing in boundary testing is identifying a class of test cases: all the valid numbers, all the numbers that are too big. Then we find representatives for these classes: the biggest valid number, the smallest invalid, too-large number. And we say this is a representative of the class that is slightly more likely to show a failure than other members of the class, and since you can't test all members of the class because there's an infinite number of tests you could run -- nobody ever has enough time. Typically, you're restricted to using one or two or some very small number of members of any class you could test. And so you're always looking for better representatives, representatives more likely to produce a problem.

Once students practice with simple boundary analysis and with the question of combining boundaries across several different variables, we start pushing them onto the next notion: What other ways are there of identifying risks? How do you find classes that will expose the risk vs. classes and tasks that will not expose the risk, and how do you come up with representatives that are worth testing?

In my experience, the more practice I can give students with this sort of exercise, which they can do at home, the more likely they are to get the principles behind it. So I have graduate students who are spending a lot of time trying to figure out how to create useful practice exercises. Ultimately, we'll probably come up with a set of materials like you see in Schaum's Outlines, which everybody who has studied either math or physics has probably used. They're just light summaries of technical material with worked examples, then lots of exercises that you practice

until you can finally solve a certain class of problem.

As a consultant, I had thought that people needed more practice with these concepts than they were getting. But there was no way I could experiment with a different style of teaching in a corporate setting, and there was no way that employees with real deadlines would come to a course that included a lot of drills. And it takes a remarkable amount of time to envision the real tasks that require practice and then come up with good exercises to provide that practice.

As a professor, I have the time and a series of involuntary subjects, as it were, to research a better curriculum. I get to try things out that I hope will improve the course, and most of them actually do. I also have students who have gone through the course and are quite enthusiastic about trying to develop practice materials, a squad of intellectuals who will get some academic credit but whom I could never afford to pay if I were a stand-alone consultant.

Guckenheimer: What kind of background do your students have, and where are they headed?

Today, I deal only with students who can write code, and we teach them how to test their own code or the code of a peer. Everybody who comes into my course is in a software engineering or computer science program and has already taken several programming courses. The first testing course covers traditional black-box testing, and the second course starts them off, first day, working with JUnit.

Many of our students at Florida Tech graduate and become professional testers in software development organizations. So a lot of what we think we're trying to do is to train the next generation of testing architects. Typically, these are people who have a lot of software development insight who either need to build tools themselves or evaluate tools and train their own staff in how to use tools really well, and to write the kind of support materials that make a specific tool useful. There is no test automation tool that solves *all* of an organization's problems, or works perfectly on its own. There is always plenty of work that needs to be done inside a company, either to change the vision of testing or to organize data or code in a way that makes it more compatible with their tool of choice. We're trying to train a generation of folks who can go out and help do that.

Guckenheimer: Are you implying that in the area of software testing, students who lack a certain real-world awareness or experience are at a deficit?

Kaner: I actually do believe that people without practical experience have a lack of perspective in tests. Earlier in my career, when I was a hiring manager, I was very disappointed when I would interview someone who came out of a traditional computer science program, and find that their testing course was fundamentally theoretical. They had no idea how to

apply that theory. We have to work very hard when we teach the testing course to provide a lot of real-life examples. We also go out and get a sample application -- some software that is under development -- and structure the assignments and much of the course around beating this program into the ground. We used Star Office last year, we used Microsoft PowerPoint once, and we used the Texas Interactive Calculator. I'm not sure what application I'm going to use this fall, but it's absolutely essential for these students to get experience with something real, or everything we teach will be academic and not necessarily very useful in the future.

I also teach the brand new metrics course here. I had a class of 15 students, mainly graduate students, and only five of them had substantial, real-life experience in software development. As I talked about when something is used, how it's used, how it can be misused, the risks to the organization of applying this measurement method, and so forth, they would understand what I was saying, because they had lived it. The other ten had incredible trouble understanding what I was getting at. Plus, unless you have the experience to understand which measures are useful when, what risks are associated with a given measure, when a given measure will have some validity, and when you can learn something from the numbers you collect, then you're like a loaded gun in the hands of an organization that really hasn't had any training in how to use it.

The folks who teach software architecture courses experience the same gulf in assimilation of theory between students who have attempted to design a moderately large program under real-world circumstances versus those who have not. So I don't think this phenomenon is unique to testing instruction. I think that, in many fields, returning students who have real-life experience are much more likely to grasp the subtleties than students who are going straight through.

Guckenheimer: I think the National Science Foundation has recently awarded you a grant to provide useable educational materials in software testing more broadly. Is that targeted to working professionals in the field? What can you tell us about that grant?

Kaner: The grant, Improving the Education of Software Testers, focuses on academic instruction for software testing. My application emphasized that there is very little in the way of academic resources -- few courses, no good textbooks, and no practice materials -- in software testing. There is no well-understood method for testing instruction as there is for teaching calculus, for example. So I wanted to put together materials that would help people build testing courses more effectively: practice exercises, and sample course notes and test tools. For example, we're writing a test program for "all pairs," a technique for dealing very efficiently with circumstances involving many variables to test together, and it lets you find a very large percentage of configuration problems with a much smaller series of tests. There's a very fine all pairs test tool on the market, but it's expensive for testing a small number of variables, such as ten, in combination. So one of my students, Nadim Rabbani, in collaboration with another Florida Tech student, Hugh Thompson, is almost finished writing an all pairs test tool that will handle up to ten variables in

combination that have maybe ten values each. These tools will be somewhat useful in industry, where some people have problems on this scale that they can't work out by hand. But where it will be most useful is in a classroom setting, where you can say to the student, "Here's the concept of combination testing, here are some thorny combination problems. Try to work these out by hand first, then use the tool and compare your results." They'll learn what this free software tool can buy them, and if they get into more complex circumstances, they'll understand why they might want to have their company invest in something more expensive.

In addition, two of my master's students, Giridhar Vijayaraghavan and Ajay Jha, are studying how programs fail. *Quality Week* will soon publish Giri's taxonomy of shopping cart software problems, which classifies a broad range of risks. If you just went to Amazon.com and imagined how to test a shopping cart, you'd come up with a few examples of what might go wrong. But with Giri's taxonomy you can start thinking by analogy about how particular programs might fail and come up with hundreds of test cases that will uncover real problems.

Though the focus of the funded work is academic, testing is an applied area; it would be foolish to think about how to teach it without considering how testing is conducted in the world. Any of the materials that we make available to faculty we're also making available to corporate teachers and trainers through a site we will soon be opening called "TestingEducation.org" Anyone will be able to download materials, like my course notes, for free. People who teach, whether in a commercial or university context, will be able to get a special password and access things like examination materials, exercises, and teaching tips that students won't have access to, but eventually we'll have practice exercises for students. The public pays for my National Science Foundation Grant, so they're entitled to this Web site.

Guckenheimer: That's great news for the testers out there. Of course, a lot of *Rational Edge* readers are not testers and test managers. How does your work touch other players in the development life cycle: requirements analysts, developers, and others.

Kaner: Everyone who goes through the software engineering program at Florida Tech is required to take two full courses on testing -- whether they want to become architects, requirements analysts, programmers, or testers. That's because we think testing is a core competency for anyone doing development. A programmer who tests his own code -- and most people do -- is going to learn better testing strategies in this course. Another takeaway from a testing course is wisdom on how to manage a project that involves many testers. And the Rational course I helped develop offers a lot of wisdom regarding where testers fit in the lifecycle and how they will interact with the rest of the company.

Our Web site will focus more on practiceable and trainable skills, which means the site is going to be very boring for somebody who doesn't want to learn how to do the technical parts of testing really well.

Guckenheimer: One final thread. We've just been talking about the connections among different participants in the development lifecycle. Through the course at Florida Tech and your own research, you've had some exposure to the Rational Unified Process.® I'm interested in your perspectives on RUP® and other process movements, such as the Agile community, and how they address testing.

Kaner: I don't want to speak to Agile Development in general, but I will speak to Extreme Programming (XP) and say that, like RUP, it has a very strong vision of lifecycle. It also has a very strong vision of some types of testing. But most of the most skilled testing that my colleagues and I know how to do doesn't fit in the XP approach. In place of strong, test-first programming (which is a wonderful practice), XP substitutes customer stories and either testing by a customer or testing by a customer's advocate, against what really look like scenarios based on use cases. This approach can expose a whole lot of problems, but it will also miss a whole lot of problems, and the framework for having an open, intelligent discussion about what the other methods of testing are and how they might fit into this scheme just isn't there. XP has a fairly narrowly patterned "right way" to go about doing things -- it's pretty good for many contexts, and not so good for others.

The Rational Unified Process is much more flexible. It's more tailorable to many circumstances; you can imagine using its iterative lifecycle approach on very small projects like computer games. And it can scale up to large telephony systems. The testing styles would have to be very different for those larger and smaller systems, and that poses a challenge to the RUP authors in terms of describing different styles and when they're needed. For example, a boundary-condition style tester will interact with folks and produce one kind of deliverables through a particular set of questions, whereas a scenario tester who bases most of his work on use cases and models developed for the system is going to come in with a whole different series of questions. And different styles of testing might be called for on a large project at different points in the lifecycle.

I was motivated to work on Rational's *Principles of Software Testing for Testers* course because I would like to see Rational extend the practical guidance available for testers in RUP. Two of my graduate students are also writing RUP extensions to provide guidance on some of the testing techniques covered in the course.

In particular, I'd like to see RUP go deeper on this problem of how testers in an iterative development lifecycle will do different kinds of testing at different times, and how they can adapt to a project team that is following a lifecycle that has a traditional basis, but is really its own variation. Over time, RUP needs to extend the library of templates and checklists, and cover skills that we drill in the course, such as bug advocacy, i.e., the effective communication of change requests so that other teams members will act on them appropriately.

Guckenheimer: We're really glad to have worked with you on the course and we're looking forward to incorporating those extensions. Thanks very much.

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

Copyright Rational Software 2002 | Privacy/Legal Information

Copyright Rational Software 2002

How the Rational Unified Process Supports ISO 12207

by **Philippe Kruchten**
Director of Process Development
Rational Software Canada

"My organization must comply with the ISO Standard 12207; can the RUP help me achieve this?"

The short answer to this question is "Yes!" The RUP provides great support for many critical coverage areas, which I'll detail in this article.

The international standard ISO/IEC 12207:1995-- Information Technology -- Software Lifecycle Processes (we'll refer to it as ISO 12207) establishes a common framework so that software practitioners can speak the same language when describing their software processes. It is not a complete, ready-to-use process, but only a framework that identifies, names, and relates various (sub)processes within the larger process domain.

The Rational Unified Process® (RUP®) is a process framework, but unlike ISO 12207, it comes not empty, but rather prepopulated with a wealth of guidance, methods, techniques, templates, and examples, out of which a concrete process can be instantiated.[1]

The purpose of this article is to:

- *Provide a brief overview of ISO 12207.*
- *Point to some differences between RUP and ISO 12207 terminology that may throw off the RUP practitioner (or the ISO 12207 literate).*
- *Describe how the RUP supports various parts of ISO 12207, and identify where and to what extent it fills in blanks.*[2]

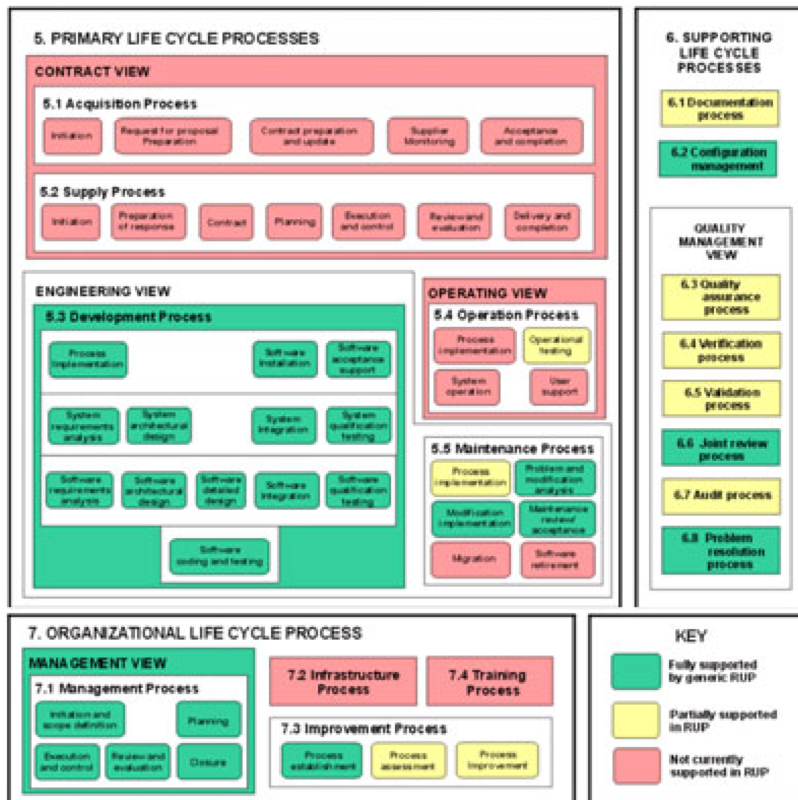
An Overview of ISO 12207

Figure 1, which is extracted from ISO 12207,[3] represents a good map of what is covered in this standard.

It shows three classes of processes:

- Primary lifecycle processes (Section 5)
- Supporting lifecycle processes (Section 6)
- Organizational lifecycle processes (Section 7)[4]

These classes can be organized in *views*, and decomposed into *activities*, which are themselves decomposed in *tasks*. ISO 12207 stops, however, at the level of activities and only occasionally mentions specific tasks, never indicating that they are mandatory.



Click to enlarge

Figure 1: ISO 12207 Processes, Views, and Key Activities (Source: Figure C.2 in the Standard)

ISO 12207 only defines, names, and indicates activities that *should take place* -- it never prescribes how they should be accomplished. It is completely neutral in terms of methods, techniques, languages, tools, and organizational structure.

It is important to note that the focus of ISO 12207 is primarily on the *acquisition* and *supply* of software, and only secondarily on software *development*. The Standard is intended for use in a two-party situation, but "may equally apply when the two parties are [ý] the same organization." This is apparent in Figure 1, which emphasizes the *Contract View*. In contrast, the generic version of the RUP focuses primarily on software development.

Differences in the RUP and ISO 12207 Terminology

There are a few differences between the way RUP (and the SPEM Software Process Engineering Metamodel for that matter) and ISO 12207 use terminology. Sometimes they use different terms that mean essentially the same thing; sometimes they use the same word or phrase but assign different meanings to it. If you are applying the RUP to help you achieve ISO compliance, it is important to understand these distinctions, so that you can apply terms correctly. Using the wrong terminology can potentially mislead assessors and delay certification.

- **Lifecycle**.[5] ISO 12207 uses the term *lifecycle* to describe the structure (i.e., the "architecture") of a complete process, that is, the collection of processes (in the ISO 12207 sense) needed to take a body of software all the way from initiating acquisition to retirement (see Figure 1), whereas in the RUP the term *lifecycle* is used to describe the unrolling ("enactment") of the process over time. In RUP, the focus is on development cycles, phases, iterations, milestones, and so forth, on a timeline; therefore *lifecycle* is related to planning. The RUP speaks of an iterative lifecycle or waterfall lifecycle, for example. ISO 12207 is silent on the shape of the process.
- **Tasks and Activities**. In ISO 12207, a *task* is a "set of elementary or atomic actions to be performed." These correspond to the RUP *Activities* and *Steps*. In ISO 12207, however, *activities* are sets of cohesive tasks, and are therefore more akin to the RUP concept of *Workflow Detail.*
- **Process**. An ISO 12207 process corresponds roughly to the RUP concept of a *Discipline*, but there are more processes in ISO 12207 than there are Disciplines in the RUP.
- **Output**. This is the term ISO 12207 uses for what the RUP calls an *Artifact* -- which results from an activity (the SPEM calls this a *Deliverable Workproduct*). In ISO 12207, artifacts that are not delivered are referred to as *non-deliverable items.*
- **Supporting *and* Organizational Processes**. ISO 12207 establishes a distinction between *supporting processes* and *organizational processes,* whereas the RUP treats them all as *Supporting Disciplines*. In ISO 12207, configuration management is a supporting process, and project management is an organizational process.
- **Infrastructure Process**. In ISO 12207, this term corresponds to the RUP *Environment Discipline*. The word *infrastructure* does *not* refer to the infrastructure of the software (OS, middleware, etc.).

RUP Coverage of ISO 12207

Refer again to Figure 1. The various colors indicate the level of support that an organization seeking to implement or comply with ISO 12207 will find in the RUP for each ISO 12207 process or activity.

- **Green**: The RUP provides in-depth coverage of this area. This is not to say that the RUP should be used "as is," out of the box. It should be tailored to suit the development conditions of the project, usually by eliminating some aspects, not by adding more.
- **Yellow**: The RUP provides some coverage, but it is likely that the organization will need to complement it with process elements: artifacts, activities, guidelines, and so on, that are specific to its domain, industry, or company, or from other processes.
- **Red**: The RUP does not provide anything significant in this area, beyond very general elements such as reviews, principles, and some techniques.

Let's take a brief look at the coverage RUP provides for specific areas.

Primary Lifecycle Processes (Section 5)

This is the area for which the RUP provides much substance, particularly in the *Engineering View*. There, the RUP provides an organization with all it needs to define the Development Process (5.3), and most of what it needs for the Maintenance Process (5.5).[6]

The RUP does not cover the Operation Process (5.4) except for Operational Testing. But as noted above, the current RUP does not cover the *Contract View:* Acquisition and Supply Processes (5.1 and 5.2).These are outside the main focus of the RUP. It should be noted, however, that the RUP provides extensive guidance in Requirements Management, which plays an important role in the interactions between supplier and acquirer.

Supporting Lifecycle Processes (Section 6)

The RUP provides great support for Configuration Management (6.2), and good to moderate support for all other processes in this category (6.2-6.8).

Organizational Lifecycle Processes (Section 7)

The RUP provides full support for the Management Process (6.2),[7] Infrastructure Process (6.2), and Improvement Process (6.3) in what it calls the Environment Discipline. It does not cover Training Process (7.4), beyond the development of training material.

Table 1 gives the ISO 12207-literate reader a few entry points into the RUP for each process.

Table 1. Where to Find ISO 12207 Processes in RUP 2002

ISO 12207 Process	**Corresponding RUP Elements** (some ISO 12207 *activities* are in *italics*)
5.1 Acquisition Process	Not covered, except elements related to Requirements.
5.2 Supply Process	Not covered, except elements related to Requirements.
5.3 Development Process	Disciplines: Requirements, Analysis and Design, Implementation, Test & Deployment. *Process Implementation* is covered by the creation of a Development Case (Role: Process Engineer) and a Software Development Plan (Role: Project Manager).
5.4 Operation Process	For *Operational Testing* see Role: Deployment Manager.
5.5 Maintenance Process	Selected activities in the Disciplines: Requirements, Analysis and Design, Implementation, Test (subset of the development process). *Problem and modification analysis* is covered by activities in Discipline: Configuration and Change Management. *Migration* is not covered, nor is *Software Retirement*.
6.1 Documentation Process	Note that all disciplines produce artifacts that are documents. See Templates. For delivered product documentation, see Role: Tech Writer, Graphic Artist, Course Developer, along with their respective associated activities.
6.2 Configuration Management Process	Discipline: Configuration and Change Management and parts of Deployment.
6.3 Quality Assurance Process	Discipline: Project Management. See concept: Evaluating Quality.
6.4 Verification Process	Discipline: Project Management.
6.5 Validation Process	Discipline: Project Management. Activity: Project Acceptance Review.

6.6 Joint Review Process	Discipline: Project Management; see various reviews. See PRA and CCB.
6.7 Audit Process	Discipline: Project Management. See its nine reviews and assessment activities. The RUP explicitly calls for Configuration Management Audits and also allows other kinds of audits to be performed as the owning organization or customer requires them. These audits are included in the QA Plan but not called out explicitly (in addition to the nine reviews).
6.8 Problem Resolution Process	Discipline: Project Management. Activities: Develop Problem Resolution Plans and Handle Exceptions and Problems. See also several activities in the Discipline: Configuration and Change Management, such as Submit Change Request Review Change Request, Make Changes, and so on. Also note that many Change Requests are the outcome of review activities.
7.1 Management Process	Discipline: Project Management. *Process implementation* is also covered by the creation of a development case (Role: Process Engineer) and several plans. These plans, which are part of the Software Development Plan, are developed by other roles in other disciplines.
7.2 Infrastructure Process	Discipline: Environment. Role: Tool Specialist and System Administrator, along with their associated activities.
7.3 Improvement Process	Discipline: Environment. Role: Process Engineer and its related activities.

7.4 Training Process	Role: Course Developer and its associated activities. Also Step: Train Project Staff, within activity: Acquire Staff.

A "Leg Up" on Compliance

There are many compelling reasons for using the RUP to assist in ISO 12207 compliance. The few differences in terminology between the two should not be a stumbling block. And although the RUP does not currently cover the acquisition and supply of software -- except in the area of Requirements Management, which plays an important role in supplier-customer interactions -- it does provide especially strong coverage in the Development Process, most of the Supporting Processes (e.g., Configuration Management), and the Project Management Process. All in all, for an organization that wishes to comply with the ISO 12207 standard, adopting the RUP will provide a serious "leg up" in the form of very detailed process guidance in many critical coverage areas.

---

[1] For more information about the Rational Unified Process, see http://www.rational.com/products/rup/index.jsp

[2] Throughout this article I will reference RUP version 2002.05.

[3] See Figure C2 in Annex C of the Standard.

[4] The numbers in Figure 1 refer to sections and subsections of the Standard; hence they do not start with 1.

[5] Moreover, the RUP treats "lifecycle" as one word, while ISO 12207 treats it as two words: "life cycle."

[6] See *The Rational Edge* article "Software Maintenance Cycles with the RUP."

[7] The RUP does not cover financial and human resources aspects, but neither does ISO 12207.

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

Copyright Rational Software 2002 | Privacy/Legal Information

Copyright Rational Software 2002

Promoting Component Architectures in a Dysfunctional Organization

by **Raj Kesarapalli**
Product Manager
Rational Software

When I first began my career as a software developer, I didn't quite understand what component architecture was all about. But after spending a few years doing software development, I now have a deep appreciation for it. As it turns out, there is only one right way to develop software, and that is by using component architectures. Unfortunately, it is still far from a universal practice. When I talk to my friends in Silicon Valley about component-based development practices, they all seem to understand it. But when I probe further, they all complain about the poor coding practices at their respective companies. They are all too familiar with the terms "band-aid" and "spaghetti code."

As developers, most of us at some point have worked late hours debugging someone else's code, because there was no way to debug our own code in isolation. Iin the age of component architecture, this should be a thing of the past, but shorter release cycles and deadline pressures cause developers to take shortcuts that defeat the promise of component architecture for a development team. In many cases, the initial designs for a team-coding environment are based on component architectures: All the major functionality is well componentized and meant to be tested in isolation. Over time, however, most of those systems initially based on components stray from the original designs, resulting in a monolithic piece of code that is hard to debug, test, and reuse. When this happens, the result is frustrated teams and delayed projects.

Why does a project team stray from an initial component-based design? If you are a developer frustrated about your teammates not adopting proper coding standards, how can you address that without sounding like a know-

it-all, or worse, insulting your colleagues? And if you are not a hands-on development manager, how can you be assured that your team is following the best practices regarding component-based architecture?

During the '90s, I had an interesting experience trying to promote component architectures and code reuse. In this article, I'll explain the hurdles I ran into and the approach I took to promote component architectures and code reuse. I hope you'll find it useful.

The Diverse Team Environment

Today, development teams are made up of developers from different backgrounds with different experiences and motivations. They do not all think alike. At first glance, everyone may seem to understand component architectures and code reuse very well, but everyone will interpret these things differently, and without proper caution, the resulting code will be hard to debug, test, and reuse.

I learned this the hard way when I spent a year at a dotcom developing a Windows-based application that lets users run applications over the Internet without having to install them on a desktop. When I first started there, the application was well designed; it was organized into many modules, each representing a core piece of functionality with well-designed APIs that other modules could call. (I call them modules as opposed to components because they are implemented as libraries with exposed APIs, not COM components. You can think of these modules as logical components.) Everyone on the team was assigned a module, and we held design reviews to discuss proposals and agree on the APIs. According to our agreements, team members would use only these APIs to call into a given module. Our agreements were based on trust, and this approach worked fine -- initially.

Over time, requirements changed, so we needed to add new functionality. As usual, time was short and pressure was great, so instead of carefully re-designing the application and re-designing the modules, developers took shortcuts. In the process of adding the new functionality, developers created dependencies between different modules by accessing data in other modules directly. They'd change some of the private methods to public methods to borrow functionality instead of moving that functionality to a shared module (as good code reuse practice dictates). In other cases, they'd borrow functionality by duplicating code in multiple locations, thus creating multiple instances of the same bugs. Over time, some modules that should have been re-designed and broken down into multiple modules grew monolithically huge instead.

Because of the dependencies introduced between modules, unit testing and unit test development became too cumbersome and time consuming. Eventually, the project team did away with unit testing, which meant you had to debug the entire application. In my case, this was extremely painful: Debugging the entire application meant rebooting the machine every few minutes. Had I been able to test my module in isolation, I wouldn't have had to do all this rebooting, which resulted in long, unproductive debugging sessions. And when new developers moved on to

the project, they had a very rough time coming up to speed and invariably introduced many new bugs in the process.

Keeping Teams Aligned with Component Architectures

At the next opportunity to add functionality, I created a new module (a library). This time, I didn't want to run into the same problem, so I took a different approach when my module was ready to be added to the application I was working on. By then, we had two other applications under development, so in addition to adding my new module to the project build that I was working on, I also added it to the other two application builds. The other teams knew they would eventually need that functionality, and since I did all the upfront work to make sure that all three applications built fine with the new module, they didn't have an issue with the early addition.

I added my new module right away (as soon as I created it) as opposed to waiting until my colleagues needed it for several reasons. First, bear in mind that my module was now part of the builds for three different applications. Each time a developer took a shortcut and added dependencies in my module to other modules, the immediate product the developer was working on built fine, but the other two failed. This forced developers to make changes to my module the right way -- coding three different shortcuts to fix the problem in the three builds is harder than coding once the right way. It prevented developers from taking shortcuts and making mistakes, and it helped my module remain componentized, so maintaining it was a breeze. Since it stayed componentized, the module was always ready for re-use, and new projects used it right away. One year later, the module was being re-used in seven different projects. This would have been impossible if I hadn't created that reuse situation up front.

Component Architectures from a Manager's Point of View

Component architectures promote code re-use, and, conversely, once a development team commits to the concept of code re-use, it becomes relatively easy to adopt the principles of component architecture. The challenge for most development teams lies in continuing to follow the principles of component architectures over a product's life cycle.

If you are a manager wanting to make sure that your team is developing code the right way, here is something you should know: Developers have a lot to focus on, and your average developer doesn't think of component architectures and/or code reuse unless asked or perhaps forced to do so. The average developer is more likely focused on getting the work done as quickly as possible before the upcoming deadlines.

As a manager, you should invest some time and effort in creating an environment in which it's hard for developers to make mistakes. The programming languages and IDEs we use today don't *enforce* the

principles of component architectures. And even in cases where these environments do *support* component architectures, there is additional work that developers need to do -- for example, in some popular IDEs, many developers feel that the frameworks supporting component architectures are restrictive and time-consuming to work with, and this is enough to prevent them from building proper components. Fortunately, as programming languages and IDEs become more sophisticated, the additional work that developers must do manually today will be automated in the future. But until then, it is up to you as a manager to make sure your team uses proper techniques for code reuse.

Consider also the scope of a given component; for example, what is a well-designed component? How much code should a given component contain? Make sure all API changes are reviewed and designed. It will make sense to add some changes to existing modules, and some changes will require the creation of new modules. The trick is to promote reuse early in the design phase, because designs that factor in reuse result in good components. This is good management practice, and it will lead to good coding practice as your team adopts the principles of component architectures and code reuse.

Try Unit Testing

If you don't have a reuse situation as I did, try unit testing to help you keep your modules componentized.[1] Should the unit test break at any given point, it is likely that someone coded dependencies into the module that don't belong there. The key is to have the developer create unit tests (at least one) before the module is made available to the rest of the team.

Developers often skip unit tests, complaining that unit test development is difficult and a waste of time. You should pay careful attention to such complaints. If a module is based on component architecture, unit test development should be trivial. These complaints may be a tip-off that the damage is already done, in which case creating a system for unit testing will represent a huge investment of time and human resources. If this is, in fact, your situation, then you should at least identify the few core functional pieces (which are usually the candidates for reuse) and componentize them one by one over time. Once you componentize them, you can add unit tests to each of these modules to keep them componentized. This will greatly help you localize bugs within modules, which means you can debug your own module in isolation, as opposed to debugging the entire application. And because developers will only change code they are familiar with, they will be less likely to introduce new defects. This will lower the defect count and reduce maintenance costs.

Parting Thoughts

As I mentioned earlier, component architectures promote code reuse and make unit testing trivial. Conversely, unit tests ensure your code stays componentized. In combination, these practices will have a positive long-term effect on your code.

Note: If you have other strategies for introducing best practices that you'd

like to share, I'd be interested in hearing about them. You can reach me at Raj.Kesarapalli@rational.com

---

Notes

[1] Many managers take unit tests for granted, and all but ignore the results. By implementing unit tests, you are effectively creating a reuse situation similar to what I did by adding my module to two other project builds. It's even simpler to use unit tests to achieve the same goal.

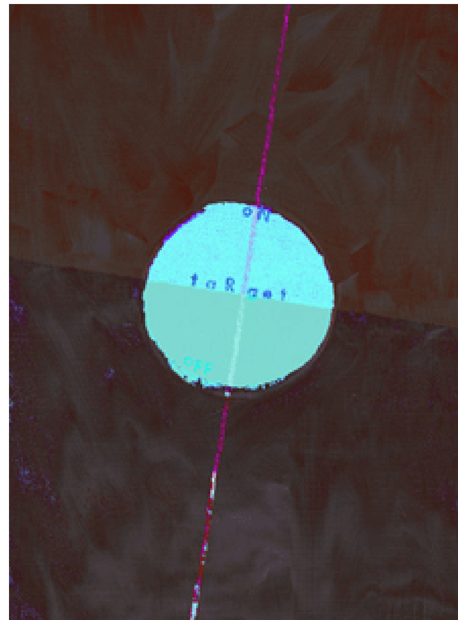
For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!

Copyright Rational Software 2002 | Privacy/Legal Information

Ending Requirements Chaos

by Susan August
Rational Software

Ask yourself the following questions:



- Does your team produce an unrealistic number of requirements at the beginning of the project, or experience "requirements creep" late in the project?
- Do your developers and testers complain that the requirements aren't precise enough?
- Are the requirements' priorities poorly defined or constantly changing, leading to team arguments?
- Do developers build what they want to build rather than what your customers think they asked for?
- Do individuals joke about the "use-less cases" floating around?

If you answered yes to any of these questions, your project is suffering from **requirements chaos**. Chaos is defined in Merriam-Webster's dictionary as "a state of utter confusion," and it follows that requirements chaos is "a state of utter team confusion caused by befuddled project requirements."

This article starts by elaborating on the cost and other consequences of requirements chaos and how this problem might be avoided ý primarily through the role of a Requirements Analyst bridging the communication gap between the customer's vision and the developer's implementation. The remainder of the article advises current and future Requirements Analysts how to set up a project environment to enable requirements success, and then outlines a requirements analysis approach designed to end requirements chaos.

From Diagnosis to Treatment

Fortunately, the diagnosis of requirements chaos doesn't have to be fatal, although it can be. Don't fool yourself into thinking that requirements chaos left to itself will somehow lead to a thriving project environment, where order mystically emerges from chaos. Authors Daryl Kulak and Eamonn Guiney said it best in their book *Use Cases: Requirements in Context*: "[We] have grown to care about requirements because we have seen more projects stumble or fail as a result of poor requirements than for any other reason."

The cost of leaving requirements chaos untreated is often expressed in terms of frustration and lost opportunities. Consider how frequently your developers and testers ask impatiently, "Now, tell me again, what exactly did you mean when you said you wanted the system to do [whatever]?" If you think this cost is hard to quantify, you might try using Alistair Cockburn's recommended term **argh-minutes** as "the unit of measure for frustrating communications sessions" and a bit of improvisation, yielding the following formula:

> Cost of unclear requirements = (argh-minutes) * (number of "arghuers" involved in the conversation) * ($2.10 a minute)

where $2.10 is Cockburn's figure for how much a programmer costs a company each minute.

At this point you likely agree that requirements matter and that the cost of requirements chaos is high. You may also be thinking, "But my team doesn't have the time to carefully specify requirements; we have to get the product out!" This is a commonly held belief among many practitioners ý but it is also a lie.

Experience tells us that an organization will always be able to specify requirements faster than those requirements can be implemented. Case in point: Your thought leaders go offsite for a "vision session," and three days later they return with enough features and functionality to keep your team busy coding, testing, deploying, and refactoring for *at least* the next year.

In his book *Agile Software Development*, Cockburn alludes to this phenomenon with his idea of **bottlenecks**, suggesting that "The nonbottleneck people can spend some of their extra capacity by starting earlier, getting results earlier, doing more rework and doing it earlier, and doing other work that helps the person at the bottleneck activity." Translated, this means that if requirements analysis is the nonbottleneck and implementation is the bottleneck, then the analyst is *obligated* to help the people at the bottleneck by specifying requirements in a formal, rigorous, and precise manner ý in other words, take the necessary time to translate the vision into a set of actionable requirements. And by actionable requirements we don't mean a grocery list of "The system shall..." statements, or some use-case ellipses and stick people dropped onto a diagram. Actionable requirements are the result of disciplined analysis that takes energy and time. Such requirements work is not for

the faint of heart.

Traditionally, requirements work has been left to a customer or customer representative (typically from the Marketing or Product Management department) responsible for telling the development team what's wanted, or to an architect or developer responsible for figuring out what the customer wants. Allocating this work to these beleaguered individuals is flawed because:

- Customers tend to create requirements specifications that look like ambitious holiday-gift wish lists.
- Marketing representatives tend to create requirements specifications that look like product data sheets or other marketing collateral.
- Product Managers tend to create requirements specifications that look like vision documents, business plans, or user's manuals.
- Architects tend to create requirements specifications that look like deployment diagrams.
- Developers tend to create requirements specifications that look like design documents.

Rarely are any of these individuals passionate about requirements and use cases; instead, they see the work as "something my process tells me I have to do before I can get on with my real work." The result is those much-feared "use-less cases."

This mismatch of role and task isn't a new phenomenon. The introduction of Information Architects and User Experience Designers into the modern software project team is an acknowledgement that designing usable software systems is difficult and requires specialized skills. Similarly, the Requirements Analyst is an emerging role poised to become the universal translator bridging the communication gap between the customer's vision and the developer's implementation. What this means is that the Requirements Analyst will work closely with stakeholders to elicit, gather, abstract, reconcile, capture, refine, and manage requirements. She will be responsible for authoring use-case and supplemental specifications as inputs to user experience designers, developers, and testers ý and she will not be faint of heart.

Preparing for Requirements Success

Suppose you're battling requirements chaos; you recognize that requirements matter, and you're determined to rein in the mess. Before tackling the requirements head-on, however, you need to take a close look at your project methodology to decide how requirements will be addressed.

Cockburn eloquently writes that "Your 'methodology' is everything you regularly do to get your software out. It includes who you hire, what you hire them for, how they work together, what they produce, and how they share. It is the combined job descriptions, procedures, and conventions of

everyone on your team. It is the product of your particular ecosystem and is therefore a unique construction of your organization."

Volumes of information are available on how to select, implement, customize, and refine your methodology. When it comes to requirements, we can limit the methodology discussion a bit and ask:

- How will you capture and track requirements?
- How will you capture and track changes?

Managing Requirements

You'll need a mechanism to track requirements, their attributes, and their related artifacts. When making this decision, you should ask yourself these key questions:

- What is the requirement lifecycle for your organization?
- What traceability is necessary between your artifacts?
- What attributes of a requirement need to be tracked?

Consider this example of artifact traceability:

- A high-level vision document contains feature requirements.
- A feature can map forward to 1.. *n* use cases or to a test plan.
- A use-case specification contains detailed use cases for a given functional area.
- A use case can map back to 1.. *n* features, or map forward to 1.. *n* test cases.

As an example of attributes that need to be tracked, a feature requirement and a use case may have the following:

- Priority
- Status (for example, Proposed, Analyzed, Designed, In Development, Checked-In, Validated in Build, Postponed, or Rejected)
- Planned Release
- Customer Contact
- Engineering Contact
- Quality Assurance Contact

Once you've answered the key questions and established your requirements management methodology, you need to decide what part visual modeling will play in your requirements capture process. As a firm believer that a picture is worth a thousand words, I'm a strong advocate of

the Unified Modeling Language (UML). Those of you who are willing to take the UML and visual modeling path should consider the following advice:

- **A fool with a tool is still a fool**. No tool will make you or your requirements smarter. It can, however, help you work through your requirements in a more organized and rigorous fashion.
- **Let others teach you.** By and large, people want to help others and share knowledge. Ask for help, and don't be afraid of those people I refer to as "UML tyrants," who absolutely insist on well-formed models. They'll knock you upside the head until you get it right; in return, you should take them out for a cup of coffee in kind appreciation.
- **Be courageous!** You model to understand the requirements and to be able to communicate the requirements to others. You don't model to make a pretty picture that's 100% accurate or complete. This is a tenet of Agile Modeling, and it's a reminder that if you're courageous and creative you may just learn something new that will help the project team.
- **Furnish your toolkit.** As a Requirements Analyst, add the following UML diagrams to your "toolkit" (to borrow Scott Ambler's term): use-case diagrams, activity diagrams, statechart diagrams, and type diagrams (described in more detail later).

Managing Changes

In any given project, the one thing you can count on is that your requirements will change. You can't avoid these changes, but you can deal with them responsibly by addressing the issues described below.

When Will Changes Be Considered?

Projects can fail when overwhelmed by change. A team receiving a constant stream of "change noise" doesn't know what signal to focus on, and as a result spins its wheels without any forward progress. This is tiresome and extremely frustrating. We might naively assume that management would protect or buffer the team from this noise, but experience tells us that management tends to make the most noise of all!

What's a team to do? Lay down the law. Decide when changes and reprioritization will be considered using "timeboxed" development. As Cockburn explains, "timeboxing guarantees that the team has the time and peace of mind to develop working software."

How Will Changes Be Evaluated and Approved or Tabled?

You'll need a defect- and change-tracking tool, and a Change Control Board also comes in handy to resolve any conflicts or planning issues that may arise. From a requirements perspective, it's often helpful for the Requirements Analyst to author an **impact assessment** that identifies the consequences of making a given change, by capturing:

- new use cases that are required
- other affected use cases (perhaps a new post-condition must be met, for example)
- other known impacts (perhaps to the architecture or performance)

The impact assessment helps the Change Control Board gauge the associated risk and effort, and if the change is approved the impact assessment can be incorporated back into the main requirements repository and related artifacts.

Anticipate Changes!

One of the more brilliant practices of Extreme Programming (XP) and Scrum is the daily stand-up meeting. It's important to recognize that this is a distant relative of the boring "status meeting" you're probably familiar with. The daily stand-up meeting is:

- Yes, held standing up. Keeping the attendees on their feet makes the meeting less likely to stray off topic. Attendees are also unable to read their e-mail or instant-message others if they're on their toes.
- Fifteen minutes or less. If the conversation strays off the topic, it can be taken offline between the people who both care and can do something about the issue.
- For core team members only. Attendees come prepared with what they know, and with what they don't know but need to find out from others. Attendees are also responsible for communicating critical information back to their subteams.
- No-nonsense. The goal should always be to get a read on the project's vital signs, find out what's brewing that the team should know about, and discover what's currently blocking the effort and what can be done to get things moving again.

The Road to Healthy Requirements

Once you've prepared your team and your environment for the requirements work ahead, it's time to roll up your sleeves and get to work. Here are the suggested steps to follow (in this order):

1. Choose a use-case specification template.
2. Gather stakeholder input.
3. Brainstorm and build out the requirements model.
4. Review and refine the requirements model.
5. Hand off the requirements to the implementers.

Choose a Use-Case Specification Template

Like virtually every practitioner, I have a preferred use-case specification template. It has the following sections:

I. Document Title and Revision History
II. Introduction
III. References
IV. Terms and Definitions
V. Use Cases
VI. Room for Supplemental Specifications
VII. Futures

Of particular interest to our discussion at this point is the Use Cases section. Below we'll look at the details that should be captured for each use case within the Use Cases section. Later I'll describe another section of interest, Futures, in more detail.

Title

The title of the use case should be reasonably general as well as self-explanatory. A good use-case title is, in most instances, a simple verb-noun combination ý for example, "Place Order." In contrast, "Agent Places Order Using the XYZ Fulfillment System" is not so good, because it's too narrow: it doesn't allow for someone (or something) other than an Agent to place an order, or for a different fulfillment system to be used in the future.

Related Models

The power of visual modeling should not be underestimated. Where appropriate, you should add visual models to support the corresponding use-case text.

Description

The use-case Description section provides an introduction to the functionality. This section can also be used to describe influencing factors not captured elsewhere or worth reiterating to the reader, such as:

- priority or schedule
- stakeholder concerns or political issues
- historical information or future direction
- technological constraints already identified

In reality, the implementers may not read the vision document, project plan, or risk list. The Description section can therefore act as a reminder to the team of all those other things they need to know about but are probably too busy to go find out about in a separate artifact.

Actors

Existing use-case literature discusses how to discover and represent Actors. Remember that Actors need not be limited to people; systems or processes can also be represented as Actors in your use cases.

Pre-Conditions and Post-Conditions

The Pre-Conditions and Post-Conditions sections identify what must be true before the use case can occur and after it has occurred, respectively.

Business Rules

There doesn't seem to be much consensus among practitioners about how to represent and use business rules. My experience has led me to use the Business Rules section of the use case to capture items that don't fit well into the pre- and post-conditions, as well as procedures and guidelines related to the system's business domain.

More often than not, I use a business rule to restate or clarify what must be true before or after the use case. Here's a (much abbreviated) example:

- **Use case**: Register to Rational Developer Network
- **Pre-condition**: Registrant is aware of a Rational Account Number that's tied to an active maintenance contract.
- **Post-condition**: Registrant has a Member Account with Rational Developer Network.
- **Business rule**: Rational Developer Network access is granted to any Rational customer with an active maintenance contract.

UX Comments

Ideally, your use cases will be handed off to a User Experience (UX) team that can help storyboard and prototype the user interaction flows. In most cases, you don't want to tell the UX team what that experience should look like. In some instances, however, you may be worried about a particular flow; for example, you might want to use a double confirmation before a significant business event (like "Delete Member Account"). This section of the use case can be used to communicate such concerns and recommendations to your UX counterparts.

Basic Flow

The Basic Flow section identifies the most common success flow. It may take the form of a prose story or a series of numbered steps. In the latter case, try to keep your flow between 7 and 12 steps.

Alternate Flow(s)

The Alternate Flow(s) section captures requirements related to the

following:

- alternate success paths
- inaction or timeout flows
- internal failures
- performance failures
- validation failures

Gather Stakeholder Input

Gathering stakeholder input is the single most important part of managing requirements. It's also the activity that defies a step-by-step approach. How input is gathered depends almost entirely on the customer and the development team. It's further complicated by individual communication techniques and a global, distributed business environment.

Sources of stakeholder input include (but are not limited to) the following:

- conversations with customers anywhere and at any time
- issues raised to Technical Support
- competitive intelligence and analysis
- existing change or enhancement requests
- conversations with subject-matter experts in the field
- conversations with the project's sponsor or sponsors
- conversations with the development team

The last point may come as a surprise, but I've noticed that the development team often knows what's unstable, missing, or in need of additional functionality before your customer does. It's also a sound practice to ask your development team for their input from time to time. In addition to giving you a fresh perspective, this has the added benefit of reengaging the developer in not just the "how" of coding but the "what" of the end product.

Some enabling technologies (both low- and high-tech) for the requirements gathering process include:

- whiteboard, hallway, e-mail, and phone conversations
- discussion boards and other threaded conversations
- Web conference conversations

Always keep in mind the artifacts you hope to produce: a vision document; use-case and other UML diagrams; requirements with use-case text; and supplemental specifications.

In a nutshell: Gather stakeholder input any way you can, get that input captured in key artifacts any way you can, and refine both your work and your understanding over time.

Brainstorm the "Day in the Life"

Once you've had the opportunity to digest what the customer is asking for, you need to figure out how to structure the requirements in a way that others will understand. Sometimes this process comes naturally, particularly if conversations have gradually gone from a high-level vision to lower-level scenarios. More commonly, however, the conversations stall out somewhere after the vision has been articulated. After some vague hand waving and overhead presentations, the customer says "Make it so." This is where projects fail. Software engineers are good people who aim to please, so they'll attempt to "make it so" even if they're not so sure what the "so" is really all about. This is also where the Requirements Analyst is obligated to make an appearance and translate the vision and overhead presentations into actionable requirements.

One way to approach the work is to brainstorm the "Day in the Life" system-context use case. The goal here is to provide the outermost framework of what the system provides. Over time, the Requirements Analyst will drill down into the core use cases, thus refining the outermost use cases.

Establishing the system-context use case is important because it provides a framework for all future work, as well as a visual and conceptual point of reference for all team members. You'd be surprised how many teams can't describe the big picture from a functional point of view. I firmly believe this is why teams have a tendency to tack on functionality in truly odd and obscure corners of the product. Failing to see the forest for the trees, a misguided team will plant some exotic plants along the perimeter and proudly view its accomplishment. Not surprisingly, the customer is less than thrilled.

When working through the "Day in the Life" brainstorm, try the following:

1. Ask "What happens during a day in the life of the system?"
   - Who are the actors that will interact with this system?
   - What does each actor do during a typical day? (Don't forget the system administrator.)
   - What might the system do by itself on a typical day? (Think about system maintenance or reporting tasks that may need to occur on a scheduled basis.)
2. Logically group functionality into high-level "Manage [Item]" or "Provide [Service]" sets of use cases.
3. For each high-level use case, drill down to the next layer of use cases. (This step will be repeated again and again over time.)

The visual output of the brainstorm (minus the actors) might look similar

to Figure 1.

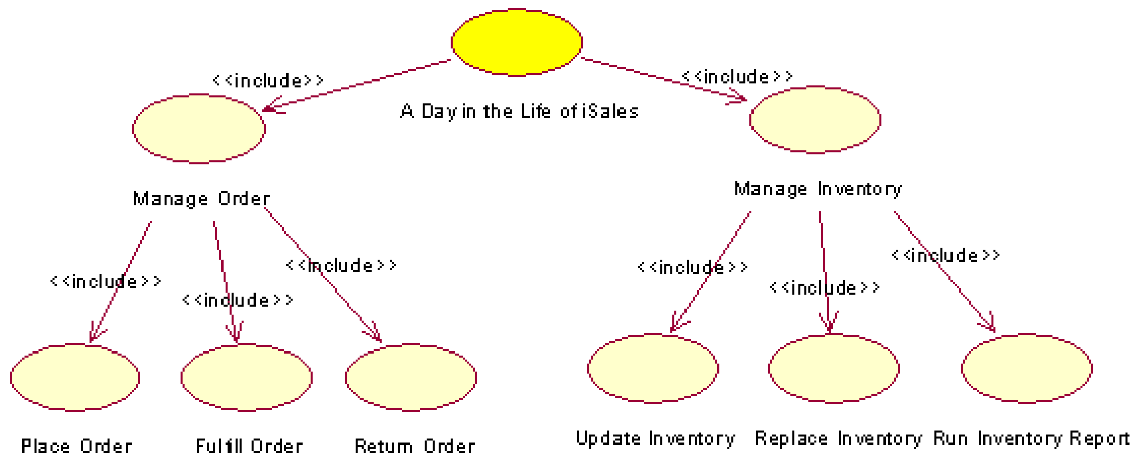


Figure 1: Visual output of "Day in the Life" brainstorm

Build Out the Requirements Model

What, exactly, do we mean when we refer to the requirements model? In his paper "UML Meets XP," Alan Cameron Wills states that the requirements model captures "everything we think we are going to provide that will be visible to the customer. That document will typically include a UML model of the functional requirements; but also non-functional requirements (performance, robustness, usability, etc.) .... It captures everything the developer needs to know to create the software." Wills also makes the important distinction that "requirements models define the behavior visible from outside a system or component .... Requirements models say nothing about the internal implementation."

Basically, the requirements model is the collection of artifacts that tell the implementation team what to build. It's the bread-and-butter deliverable for the Requirements Analyst. To create the model is to fill out the use-case specification template described earlier.

While working on the requirements model, the Requirements Analyst should do the following:

1. Start building out the visual models early.
2. Gradually drill down into the core use cases, always keeping the system-context use case in mind.
3. Focus on the pre- and post-conditions of each use case first and worry about the nuances of the basic and alternate flows later.
4. Add placeholders for future elaboration whenever a stumbling block is encountered.
5. Capture all questions, thoughts, and worries in writing and keep moving ahead!

It's important to work through your requirements with "beginner's mind" ý that is, with few preconceived notions of how the system will work. Focus

instead on what the system might and should do. Early on, you'll also see inconsistencies, gaps, and other oddities that you'll become blind to as time passes. Note these issues wherever you find them, and use a standard convention so that your team becomes accustomed to your questions and observations. I typically put my comments right next to the area in question, prefacing them with "Q:" ý for example, "Q: Are there any other required fields to capture at the time of registration?"

Review and Refine the Requirements Model

The requirements model is an evolutionary beast that will grow, contract, and mature over time. At any point you can assess the overall completeness of the model by asking these questions:

- Does the model capture and describe what the system under design is to provide?
- Does the UML use-case model make sense at both the macro and micro levels; does it hold together?
- Will the use cases be flexible and extensible over time?
- Are the use cases as free of user interface details as possible, and as free of implementation and technological details as possible?

Obeying the last item is second nature to a seasoned Requirements Analyst but is a common stumbling block for those new to the discipline. As Cockburn states in *Writing Effective Use Cases*, "Describing the user's movements in operating the system's user interface is one of the more common and severe mistakes in use case writing and is related to writing goals at too low a level ... [making] the requirements document worse in three ways: longer, brittle, and overconstrained." In other words, separate the "what" of your requirements from both the "how" of the user interface implementation and the "how" of the technical implementation. Leave room for other individuals and teams to refine your work.

It's also useful to test the completeness of your requirements model by employing "scenario testing" and "usage narratives." Taking a real-world example and working through the model to ensure that everything has been addressed is both effective and a boon to your Quality Assurance organization, since the team now has ready-made acceptance tests. Here's an example: "Today Mary is in a hurry; she wants to place an order quickly and view status on an order she placed last week. She starts by ..." If your model can't explain how Mary will be satisfied by the system under design, something is missing.

Still not convinced that your requirements model is ready? The article "What Does 'No Time For Requirements' Mean?" refers to the IEEE 830 Documentation Standard for a Software Requirements Specification and describes a good requirements set as one that's complete, consistent, correct, modifiable, traceable, unambiguous, and verifiable. I'd add to this list "prioritized" and "understandable to the intended audience." All are laudable goals for a requirements model.

Hand Off the Requirements to the Implementers

Suppose you've built out the requirements model, there have been reviews of the requirements model among nonbottleneck individuals, and there have been refinements to the requirements model based on the feedback received. Now the bottleneck individuals are ready for your input; the team is ready to implement.

The next steps can be summarized as lock down, distribute, and review. First, lock down the requirements model so that all additional changes must go through your established change control process. Next, distribute the model to those individuals responsible for the implementation work. Finally, review the requirements model together until everyone is both literally and figuratively on the same page. As the review process unfolds, try to keep the following guidelines in mind:

- **Negotiate**. Kulak and Guiney have observed that "Projects work better when the project plan is nothing more than an ongoing negotiation." Enter the review process knowing both what you want to achieve and what can be sacrificed this time around. If you've already prioritized the work, you're ahead of the game. Recognize, however, that prioritizing all of the work with the same high priority level is usually both a waste of time and an insult to the team.
- **Be responsible**. If a decision is made, stick to it for the duration of the current timeboxed development period; try to keep the change noise to a minimum. Remember that you'll be able to refactor the requirements the next time around. If questions arise that require additional requirements work, you may want to defer the work to the next cycle. On the other hand, if you simply can't afford to wait for the functionality, or if there's room in the schedule, revise and refine the requirements model until it's correct and complete and answers all questions raised by the implementers.
- **Set up for the next round**. Cockburn reminds us that "Software development is a (resource-limited) cooperative game of invention and communication. The primary goal of the game is to deliver useful, working software. The secondary goal, the residue of the game, is to set up for the next game. The next game may be to alter or replace the system or to create a neighboring system." As soon as the implementation team moves forward with the requirements you've provided, you must set up for the next round. As you remain available to answer questions about the existing requirements model, begin work on its next incarnation. Reuse what you can, and remember that capturing new functional requirements may be as simple as adding a new post-condition to an existing use case.

Tips for a Speedy Recovery from Requirements Chaos

Until now, this article has focused on the procedural aspects of requirements analysis: what to do, when to do it, and to some extent how

to do it. This section is intended more as a set of helpful hints and best practices, touching on what it means to be an analyst and to be passionate about requirements analysis. I encourage you to incorporate and build on these practices on a daily basis.

Have Fun

Abstracting complex systems into visual models that clarify and communicate can be fun, as can: taking creative risks as you learn to apply new UML diagrams; learning to trust your instincts as you intuitively feel your way through a complicated business domain; and relaxing into the reality that a healthy requirements model will evolve over time, and thus "done" is only a state of mind. In short, you really should try to have fun, because the alternative is tedious requirements work!

Color Outside the Lines

The main goal of the requirements model is to capture and communicate what the system under design must provide. Whatever devices you come up with to convey this ý including tables to capture CRUD (Create, Retrieve, Update, Delete) requirements, or hand-drawn mockups of reports ý are all perfectly acceptable as long as they improve communication within and across teams.

Honor Simplicity and Be Succinct

To quote Dee Hock (founder and CEO emeritus of Visa International, "Simple, clear purpose and principles give rise to complex, intelligent behavior. Complex rules and regulations give rise to simple, stupid behavior." Enough said.

Focus on "The Right Requirements for Right Now"

I joke with colleagues that I intend to tattoo this personal motto on my forehead. It's a powerful reminder to focus on the right items at the right time. Resist the temptation to focus on easy, low-hanging fruit ý unless of course it happens to be the right requirement for right now. Listen politely to others who hand-wave about the *n*-year vision, but focus on what you're delivering to the customer tomorrow and the day after that. In the end, the right requirements right now incrementally lead to the right *n*-year outcome.

Know Your Audience

As a colleague of mine once put it, is your audience "rubber meets the road" (implementation) or "rubber meets the sky" (vision)? Based on the answer, you'll need to incorporate the appropriate mix of text and visuals. I've found that engineers love pictures, testers loves pictures and text, documentation teams love text, and everyone else loves Microsoft PowerPoint presentations. Plan accordingly.

Describe What, Not How

The analyst's job is to capture the "what," not the "how." Leave functional implementation details to Engineering, and leave user interface implementation details to the User Experience Designer. For example, "User indicates her selection" is good, whereas "User selects her choice from the drop-down menu and clicks the Submit button" is not.

Live the Pre- and Post-Conditions

Pre- and post-conditions are the most powerful elements of your use case. They define what must be true before and what must be true after a given functional event. The post-conditions for one use case may also be the pre-conditions for another use case, thus specifying a sequence or necessary flow of activities. For improved requirements precision, try to author your use cases by relying more on the pre-conditions and post-conditions and less on the basic flow.

Keep an Eye on Granularity

Always work breadth-first and then drill down. Remember that use cases aren't a user's manual. Also be forewarned that alternate flows, although very powerful for revealing obscure business rules and important exception handling, can be a glorious time sink. Constantly ask yourself, "Is this refinement activity the best use of my time right now?" and move on to a new use case when the answer is no.

Keep an Eye on Scope

Carefully distinguish system functionality from behaviors and processes not governed by the system under design. Human-driven steps should be noted as such. Also be careful not to hide new functionality in the Business Rules or Alternate Flow(s) sections; generate new use cases instead.

Maintain a Futures Section

The Futures section is a parking lot for ideas they may be the right requirements for the future. I also suggest adding a caveat along the lines of: "This section is a placeholder for ideas and notes relating to future XYZ functionality. This section is not an implementation request to Engineering; rather, it is a statement to stakeholders that their issues have been captured and will be considered moving forward. Please recognize that this section is neither complete nor an actual statement of functionality to be delivered in the future."

Don't Overlook Unglamorous Requirements

Some requirements aren't particularly flashy, and you'll be tempted to ignore and avoid them. Please don't. I have yet to meet a system that didn't need at least one of the following:

- authentication and authorization of users
- auditing and event logging
- backup and recovery

- reporting

Tackle the Nonbehavioral Requirements Early

Frank Armour and Granville Miller note in their book *Advanced Use Case Modeling* that "Nonbehavioral requirements are not the functions or the behaviors themselves; they are attributes or characteristics of the system behaviors, such as security, reliability, and performance." It's important to address and capture these nonbehavioral requirements early, because they're essential for building the right architecture. The article "Capturing Architectural Requirements" addresses this very fact and is a must read for every analyst.

Using Type Diagrams

In his paper "UML Meets XP," Wills discusses a technique whereby use cases are specified in terms of **type diagrams**. This technique effectively refines stick-figure actors into types with attributes. The advantages of this approach are that:

- The analyst can articulate and describe the requirements space more clearly.
- The UML can be used to describe relationships between types.
- This increased precision aids in the requirements discovery process.

Figure 2 shows an example.

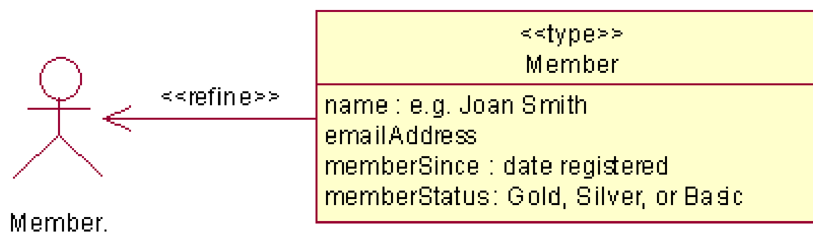


Figure 2: Specifying a use case in a type diagram

The type diagram also uses pidgin-programming-speak where necessary, as opposed to the much more formal (and much less comprehensible) Object Constraint Language (OCL).

Although types may later be refined and implemented as classes by Engineering, a type is not a class. If the type `Member` has an attribute of `emailAddress` (as in Figure 2), it doesn't mean there has to be a class `Member` with an attribute by that name. By specifying types and attributes in a visual model, the analyst is simply using a convenient, precise shorthand for representing requirements; for example, in this case it's easy to see, without any accompanying text, that the system under design has a member with a corresponding e-mail address.

As a practitioner, you should use multiple modeling techniques to assist

you during the requirements specification process. As noted earlier, in addition to use-case diagrams and type diagrams, activity diagrams and statechart diagrams are also worth exploring. See the UML tutorials for a good overview.

There Is a Cure

Requirements chaos is both a common and a costly affliction among software development teams. Fortunately, the chaos can be eradicated by enabling and empowering the Requirements Analyst to develop a rigorous and precise requirements model. If "art is the triumph over chaos," as John Cheever wrote, then artful analysis is indeed the key to triumphing over requirements chaos.

References and Other Resources

Books:

- *Advanced Use Case Modeling: Software Systems, Vol. 1* by Frank Armour and Granville Miller (Addison-Wesley, 2001)
- *Agile Software Development: Software through People* by Alistair Cockburn (Addison-Wesley, 2001)
- *Use Cases: Requirements in Context* by Daryl Kulak and Eamonn Guiney (Addison-Wesley, 2000)
- *Writing Effective Use Cases* by Alistair Cockburn (Addison-Wesley, 2000)

Rational Developer Network resources:

- "What Does 'No Time For Requirements' Mean?" by Jim Heumann
- "Capturing Architectural Requirements" by Peter Eeles
- "UML Tutorials"

Other Web resources:

- Agile Modeling site maintained by Scott Ambler
- "UML Meets XP" by Alan Cameron Wills

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

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

Copyright Rational Software 2002 | Privacy/Legal Information

Book Review

Developing Enterprise Java Applications with J2EE and UML
by Khawar Zaman Ahmed and Cary E. Umrysh

Addison Wesley, 2002
ISBN: 0-201-73829-5
Cover Price: US$39.99
330 Pages

Ahmed's and Umrysh's book provides a high-level overview of Java Platform 2, Enterprise Edition (J2EE); Unified Modeling Language (UML); and how UML-based representations of J2EE systems can be evolved using an analysis-and-design process that is a subset of the Rational Unified Processý (RUPý). The book can be a useful addition to your technical collection if:

- You are a Java developer looking for an easy-to-read introduction to J2EE and UML.
- You are a J2EE developer who needs an initial introduction to UML.
- You have a good understanding of UML, and you need an introduction to J2EE technologies.

If you are looking for a reference that provides technical depth on subjects such as the details of J2EE source code or how to address common J2EE issues using standard design patterns, then this is not the final book for you. It still can, however, provide background knowledge and understanding that will enable you to digest the more in-depth technical references.

Organization

Although the book is not formally organized as such, it can be divided into three conceptual sections:

- Concepts definition
- Process overview
- Concepts application

The first two sections prepare the reader to benefit from the third, which contains the true meat of the book. There, Ahmed and Umrysh employ a simple analysis-and-design process to progressively evolve UML diagrams that describe a piece of a J2EE-based system.

Concepts Definition

The first four chapters of the book provide some useful conceptual background on the enterprise applications problem space addressed by J2EE, the key technologies and APIs that compose J2EE, UML basics, and UML representation of basic Java constructs.

Java developers should be able to absorb the UML introduction easily. Ahmed and Umrysh present only the pieces of UML that are used in diagrams they build across later chapters. Further, they illustrate many of the UML concepts using figures that include Java code and the resulting UML representation.

One type of UML diagram they do not illustrate in the foundations section is statecharts. This might cause some confusion for readers who are not familiar with state-based modeling, because the discussions of component lifecycles in later chapters often document lifecycle models using statecharts. Readers should refer to one of the more complete UML texts (e.g., *The Unified Modeling Language Reference Manual* by Rumbaugh et al., Addison-Wesley, 1999) for background information on these diagrams.

Process Overview

Chapters 5 and 6 provide brief surveys of different software development processes and means for describing system architecture. Chapter 6 is particularly useful for readers who are struggling to justify why they should spend time defining and documenting a consistent systems architecture. This chapter also provides useful discussion of key concepts, such as decomposition, frameworks, patterns, and layering, that are central to defining and describing such architectures.

Chapters 7 and 8 provide details about the analysis-and-design approach the book adopts in walking through the diagrams. This process is a subset of the activities defined for RUP's Analysis and Design Discipline. Specifically, the authors:

- Start with the assumption that the development team has defined requirements in the form of system use cases.
- Use RUP's Use-Case Analysis activity to define how high-level analysis elements interact to satisfy the system's functional requirements.
- Use RUP's Use-Case Design activity to transform the analysis elements into implementation elements. This transformation progresses throughout the Concepts Application section of the book, as the authors present aspects of J2EE that address different systems issues.

The process steps used in the book are sufficient to support the book's purpose, which is to illustrate how to represent a J2EE system using UML. The authors do point out that their process is just a small portion of RUP, but some readers might have been better served if the authors had described the full RUP analysis and design discipline from which they extracted their process.

Concepts Application

The major value of this book resides in Chapters 10-14, which have a consistent format:

- Presentation of a specific J2EE technology -- either Servlets, Java Server Pages (JSPs), Session Enterprise JavaBeans (EJB)s, Entity EJBs, or Message EJBs.
- Types of issues addressed by the technology.
- Modeling the technology using UML.
- Adding instances of that technology into the diagrams that describe the sample system.

The authors have made a concerted effort to adopt either generally accepted practice or approved specifications for their modeling guidance. Specifically:

- For the most part, the chapters on modeling Servlets and JSPs (Web components) follow the guidance provided in Jim Conallen's book, *Building Web Applications with UML*. Conallen's work is largely accepted as the de facto standard approach for Web application modeling.
- The authors' UML-to-Java mappings follow the version of Java Specification Request-26 (JSR-26) that was undergoing final review while the book was being written. Ahmed's and Umrysh's approach to illustrating the mappings is easier to follow than is the specification itself.

It is gratifying to see this usage of recognized best practices. Our industry needs convergence rather than divergence in approaches.

Chapter 15 provides the first clear discussion that I have seen regarding using UML to model the deployment of J2EE-based systems. This chapter also provides an enlightening discussion of how the simple UML model that has been developed throughout the book facilitates the traceability of artifacts from requirements through deployed modules.

Finally, Chapter 16 presents the case study that is used throughout the book.

Things I'd Like to See in the Second Edition

I've talked with a number of other readers of this book; all have found it a

delightful introduction to the subject matter. There are a few things, however, that I would like to see in a second edition:

- Better in-text attribution of original references. Although the reference list is reasonably comprehensive, the authors provide limited in-text attributions to these references. For more conscientious readers, this makes it difficult to identify which particular reference to examine for more in-depth understanding of a given topic or technique.
- Update the EJB modeling guidelines to reflect the contents of the version of the JSR-26 specification that currently is under review. One unfortunate aspect of documenting a quickly evolving technology such as J2EE is that the lifespan of any definitive reference is fairly short. This is true of the initial release of JSR-26, which is based upon the EJB 1.1 standard. JSR-26 is being revised to incorporate new features of the EJB 2.0 standard and to clean up some holes in the first release of the specification. The changes are not radically different, but it would be great to have an updated edition of the book-under-review that provides popular access to the new aspects of JSR-26.
- Incorporate discussion of the Core J2EE Patterns into the book. In the time since the authors wrote their book, Sun has published a series of design patterns that are recommended for addressing commonly occurring J2EE system design issues. The authors discuss design patterns at many points in their book, so recasting these discussions in terms of the de facto standard J2EE design patterns would be in line with their emphasis on applying recognized best practices.

Final Thoughts

Developing Enterprise Java Applications with J2EE and UML most likely will retain a spot on my reference bookshelf for the foreseeable future. Its high-level approach to the UML and J2EE make it a quick and easy read for brushing up on the key aspects of modeling J2EE systems. It is a recommended read for anyone who needs an overview either of J2EE or UML, or on how to use the two together effectively. It will not be a primary reference for readers who need technical depth on those subjects, but it provides an initial foundation upon which deeper study can be built.

-**Todd Dunnavant**
Rational Software

For more information on the products or services discussed in this

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

Copyright Rational Software 2002 | Privacy/Legal Information

subscribe
contact us
submit an article
rational.com
issue contents
archives
mission statement
editorial staff

Book Review

Documenting Software Architectures

by Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, Judith Stafford

Forthcoming: Addison-Wesley, September 2002
ISBN: 0-201-703726
Cover Price: US$59.99
640 Pages

Ten years ago, I was brought in to lead the architecture team of a new and rather ambitious command-and-control system. After a rocky beginning, the architectural design work started to proceed full speed, with the architects finally forging ahead, inventing, designing, trying, and resolving, in an almost euphoric state. We had many brainstorming sessions, filling whiteboards with design fragments and notebooks with scribblings; various prototypes validated -- or invalidated -- our reasoning.

As the development team grew in size, the architects had to explain the principles of the nascent architecture to a wider and wider audience, consisting not only of new developers but also of many parties external to the development group. Some were intrigued by this new (to them) concept of a software architecture. Some wanted to know how this architecture would impact planning, organization of the teams and the contractors, delivery of the system, and acquisition of some system parts. Some parties wanted to influence the design of this architecture. At a further remove from development, customers and prospects wanted a peek, too. So the architects had to spend hours and days describing the architecture in various forms and levels and tones to varied audiences, so that each audience could better understand it.

Becoming such a center of communication slowly stretched our capacity. On one hand, we were busy designing, and validating the architecture; on the other hand, at the same time we were communicating to a large audience what the architecture was, why it was the way it was, and why we did not choose some other solution. A few months into the project, overwhelmed, we began having a hard time even agreeing among ourselves about what it was we had actually decided.

This led me to the conclusion that "If it is not written down, it does not exist." This became sort of a *leitmotiv* within the architecture team for the following two years. As the ancient Chinese poet Lao-Tsu says in the *Tao*

Te Ching,

```
Let your workings remain a mystery.
Just show people the results.
            (Tablet #36)
```

The architecture *could have been* whatever we had talked about, argued, imagined, or even drafted on a board. But in the end, the architecture of this system *was* only what was described in one major artifact: the *Software Architecture Document (SAD)*. Architectural elements and architectural decisions not captured in this document simply did not exist. This one rule -- "*If is not in the SAD, it does not exist*" -- provided incentive to evolve the document and keep it up to date, almost to the week. It also gave us an incentive not to include anything and everything, such as untried ideas, in the SAD, which became the project's definitive arbiter and a central element in the life of the project. It was our display window for showing off our stuff, our comfort when we were down, and our shield when attacked.

The key questions we faced at the time were: What should we document for our software architecture? How should we document it? What outline should we use? What notation? How much or how little information should we include? There were few exemplars of architectural descriptions for systems as ambitious as ours. Driven by necessity, we improvised. We made mistakes, and corrected them. We rapidly discovered that architecture was not flat, but rather multidimensional, with several intertwined facets. Some facets -- or views -- were of interest to only a few parties. We found that many readers would not even open a document that weighed more than a pound, and that we would have a hard time updating it anyhow. We realized that, unless we captured the reasons for our choices, we were doomed to reconstruct them again and again, every time a new stakeholder with a sharp mind came around. We picked a visual notation that was neither too vague and fuzzy nor too esoteric and convoluted, to avoid discouraging most parties.

Today, software architects have a great starting point for deciding how to document their software architecture: With this book, you will have what you need in your hands. The authors went through many experiences similar to mine and extracted the important lessons learned. They read many software architecture documents. They reviewed the academic literature, studied all the published books, checked the standards, and synthesized all of this wisdom into this handbook, which describes the essential things you need to know in order to define your own software architecture document. You will find guidance for defining the document's scope and organization, and on the techniques, tools, and notation to use (or not to use), as well as comparisons, advice, and rules of thumb. You'll also find templates to get you started, and continuing guidance for the times you get lost or feel despair along the way.

This book is of immense value. Description and communication about a software architecture is crucial to that architecture's many stakeholders. This handbook can help you with both and save you from months of trial and error, lots of undeserved hassle, and many costly mistakes that could potentially jeopardize your entire endeavor. It is an important reference

for the shelf of any software architect. If you have read and understood the IEEE Standard 1471-2000: "*Recommended Practice for Architectural Description of Software Intensive Systems*," but still wonder how to actually implement that practice, you will find this book is a complete Users' Guide that explains not only the "4+1 views" from the Rational Unified Process, but other approaches as well.

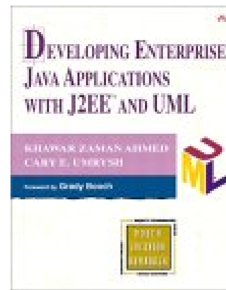
-**Philippe Kruchten**
Rational Fellow
Rational Software Canada

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

Copyright Rational Software 2002 | Privacy/Legal Information

"Servlets" (Chapter 10*)

from *Developing Enterprise Java Applications with J2EE and UML* by **Khawar Zaman Ahmed** and Cary E. Umrysh (Addison-Wesley Object Technology Series, 2002).



Written by a current (Ahmed) and a former (Umrysh) member of the Rational Rose product team, this new book is suitable for anyone interested in learning about the Unified Modeling Language (UML) and how to apply it to development for the Java 2 Platform, Enterprise Edition (J2EE). The first half of the book focuses on UML basics and the second on applying them in a J2EE context, assuming some familiarity with Java. Taking a holistic approach, the authors examine all elements of the development process -- architecture, analysis and design techniques, development processes, and visual modeling -- from a J2EE perspective.

Chapter 10, "Servlets," provides an overview of the Java servlet technology, which is ideal for the request-response oriented Web paradigm. It discusses how servlets are modeled in the UML, and then shows how to apply the UML and servlets to a case study.

* Chapter posted in its entirety by permission from Addison-Wesley.

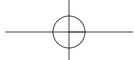
For a review of *Developing Enterprise Java Applications with J2EE and UML*, visit our Rational Reader section in this month's issue.

Chapter 10 pdf file (567 K)

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

Copyright Rational Software 2002 | Privacy/Legal Information

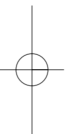
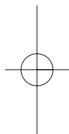
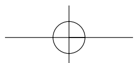
Ahmed02.ps 10/4/01 1:03 PM Page 131



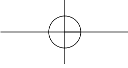
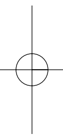
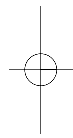
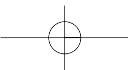
Chapter 10

Servlets

- Introduction to Servlets
- Servlet Life Cycle
- Request Handling
- Response Generation
- HTTP Request Handlers
- The RequestDispatcher Interface
- Modeling Servlets in UML
- Modeling Other Servlet Aspects
- Servlet Deployment and Web Archives
- Identifying Servlets in Enterprise Applications
- Summary

Ahmed02.ps 10/4/01 1:03 PM Page 132

❖ ***Process Check: In this chapter, we focus on design as we progress through the Rational Unified Process (RUP) analysis and design discipline. We also discuss some aspects of implementation in the context of the servlet technology.***

Recall the control object TransferFunds from the discussion in Chapter 6. If you look closely at the final sequence diagram presented in Chapter 6, you'll notice two very distinct types of interactions performed by this class:

- *Interactions with boundary objects to obtain information and perform some basic work*
- *Interactions with entity objects*

Implementing a control class with a dual set of responsibilities and a large scope would make the control class less maintainable and less scalable. To make the control class more maintainable and scalable, it is preferable to partition the control class into two classes, one focused on the external interaction and the other responsible for carrying out the internal coordination and logic.

As it turns out, the externally focused part of TransferFunds evolves to a Java servlet. We introduce the servlet in the next section, and then discuss how you actually determine the responsibilities of the servlet in the context of the HomeDirect case study.

Introduction to Servlets

Historically speaking, servlets have been around longer and have seen much wider use than other Java 2 Platform, Enterprise Edition (J2EE) technologies. In the past, they tended to be large in size and complicated to maintain in comparison to the level of Web functionality they actually provided. Going forward, servlets will likely continue to see wide use for some time. However, their typical size is shrinking, and the level of complexity they tend to deal with is consistently becoming less.

The biggest benefit servlets offer developers is that they are designed specifically to process Hypertext Transfer Protocol (HTTP) requests coming from the Web client and pass back a suitable response. They perform this function well and require few resources to deliver this functionality.

In terms of structure, servlets are specialized Java classes that closely resemble the structure of Java applets, but they run on a Web server instead of a client.

An interesting point to note is that servlets can never have their own graphical user interface. Web servers host these components through the use of a Web container that manages all aspects of their life cycle.

Common Usage

Servlets have the distinction of being the most frequently used J2EE components currently found on the World Wide Web. As stated earlier, they typically involve a compact, lightweight architecture and design. They also tend to work well in cases where the requirements placed on this type of Web component are relatively small.

Most Web developers use servlets as the main point of entry to their server application from the Web client, and in this way, they are simply used as a conduit to pass information back and forth between the client and the server. Allowing client control to add or remove Web pages or files from the server can also be a good use for servlets, as long as the client has sufficient security clearance. Understandably, this usage is less frequently seen in practice.

Best Served Small

In theory, servlets are capable of doing just about anything possible that can be done with Java. The question arises as to why Web developers don't just build everything they need using these components. The problem is that building large servlets to handle complex Web interactions, transactions, database synchronization, and other internal logic is not a very scalable approach. Developers would spend most of their time working out the intricacies of low-level transactions, state management, connection pooling, and so on.

In the past, servlets were often built to perform most or all of the following tasks:

- Check and process user input
- Handle significant business logic
- Perform database queries, updates, and synchronization
- Handle complex Web transactions
- Generate dynamic Web page content as output
- Handle Web page forwarding

More advanced J2EE solutions make use of JavaServer Pages (JSP), Enterprise JavaBeans (EJB), and JavaBeans to split up and offload much of this work, often

Ahmed02.ps 10/4/01 1:03 PM Page 135

using new mechanisms built into J2EE to simplify the more difficult tasks for the developer. Servlets are then responsible for a more manageable set of tasks:

- Gathering and validating user input, but little or no actual processing
- Coordination of output, but with little or no direct generation of dynamic Web page content
- Minimal business logic

As you can see, servlets are best served small.

If constant demand for new Web site functionality did not exist, huge servlets could be built with all the accompanying aches and pains, and they might even stand a reasonable chance of being adequately maintained. However, the fact is that demands on Web sites keep increasing. Every service provider on the Web must continually update and upgrade to give their customers that new bit of data, that new cool feature, or that prized extra that differentiates their service from everyone else's service.

Unfortunately, the bigger servlets come at the cost of an increased challenge of providing adequate code maintenance, not to mention the increased risk of breaking some of the existing functionality. The blessing of a lightweight architecture at the outset can easily turn into a wretched curse later on if you are not careful.

J2EE Versions

The information in this chapter applies equally well to servlets using J2EE 1.3 or J2EE 1.2. The differences between these two specifications are insignificant with respect to the basic Unified Modeling Language (UML) modeling of these particular Web components.

Servlet Life Cycle

As stated earlier, servlets are deployed within a servlet container, which in turn is hosted by a Web server. The particular capabilities and level of compliance of the Web server determines which version of the servlet specification you need to be working with.

The basic behavior of a servlet involves a request-response type model derived from the way the HTTP works; thus, the inherent applicability as a Web component. This behavior is illustrated via a statechart diagram in Figure 10-1.

Servlets are built as Java classes that extend one of two basic servlet implementation classes: **HttpServlet** and **GenericServlet**. The former is the most often

Ahmed02.ps 10/4/01 1:03 PM Page 136

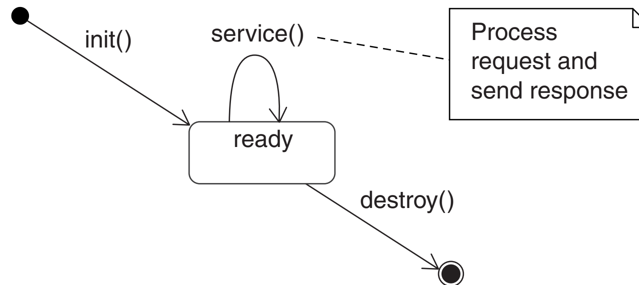


Figure 10-1 Servlet life cycle

used, yet slightly more complex of the two. Both servlet types employ the same basic life cycle.

Life Cycle Methods

The servlet life cycle makes use of three basic request handler methods, of which any or all can be implemented within the extended servlet class:

- **init:** Initializes the servlet
- **service:** Services the client request
- **destroy:** Destroys the servlet

Of these three methods, the **service** method is the most interesting because it actually does the majority of the necessary processing. It typically does the following:

- Receives the request from the client
- Reads the request data
- Writes the response headers
- Gets the writer or output stream object for the response
- Writes the response data

The **service** method is at the heart of the **GenericServlet** type. However, it is almost never overridden and instead is split into lower level HTTP request handlers when used with the **HttpServlet** type.

The **init** and **destroy** life cycle methods are always available to be overridden, but in several cases might not be used if the servlet has no specific objects or connections it needs to initialize or terminate.

Ahmed02.ps 10/4/01 1:03 PM Page 137

A sequence diagram in Figure 10-2 shows a simple example of a servlet. This diagram applies to both the **GenericServlet** and **HttpServlet**. It highlights a simple example where a database query is made to formulate the response to the client. Note that the **service** method is further refined into a specific HTTP request in the case of **HttpServlet**.

Convenience Method

Besides the life cycle methods, servlets commonly make use of what are referred to as convenience methods. One such convenience method that applies for all servlets is **getServletInfo**, which returns a general info string about the particular servlet—normally author, version, usage, and so on.

Required Methods and Tagged Values

When building a servlet that extends the **GenericServlet** class, the **service** life cycle method must be implemented; otherwise, the servlet is invalid. All other methods are optional.

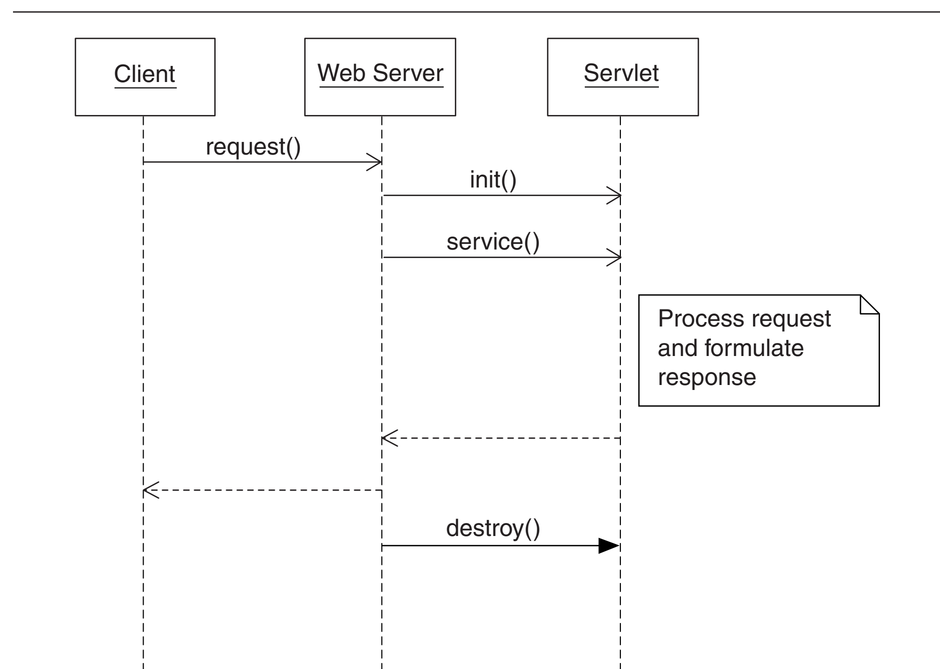


Figure 10-2 Sequence diagram showing servlet life cycle

Ahmed02.ps 10/4/01 1:03 PM Page 138

Multiple threads may call a generic servlet instance's **service** method concurrently. To avoid this, the servlet can implement the **SingleThreadModel** interface, which is really a method of typing the servlet and indicating to the Web container that only a single thread should be allowed to call the method at any given time.

Implementing the **SingleThreadModel** can have a very significant effect on how the container decides to allocate resources when the servlet is deployed on the Web server, which can greatly impact the total number of concurrent servlet instances allowed.

Using this approach may be appropriate if you are dealing with a situation in which the servlet may need to alter information that is not thread safe or access resources that are not thread safe.

It is not recommended that you attempt to serialize any of the servlet methods other than by implementing this interface. The interface itself introduces no new methods.

Request Handling

Servlets are request-driven and have specific capabilities available to them that simplify handling of incoming requests.

Recall that a request to a servlet may consist of several pieces of data (for example, when a form consisting of several fields is filled in and submitted).

When the Web container receives a request intended for a servlet, it encapsulates the incoming data into a **ServletRequest** object (commonly referred to as the request object) and passes it on as a parameter to the servlet's **service** method. The servlet can then use the methods available in the **ServletRequest** interface to query the request object. Some of the queries are contained in the following list:

- **getCharacterEncoding** obtains information about the encoding format used for the request.
- **isSecure** finds out if the request was made over a secure channel.
- **getParameterNames** obtains a list of all parameter names in the request.
- **getRemoteAddr** determines the IP address of the client that sent the request.
- **getParameter** is used to retrieve the first parameter value associated with a named parameter type.
- **getParameterValues** is used to retrieve multiple parameter values associated with a named parameter type.

```
HttpSession session = request.getSession(true);
:
:
// obtain the values for UserID and password
String loginID = rquest.getParameter ("USERID");
String loginPassword = request.getParameter ("PASSWORD");
:
```

Figure 10-3 Using the request object

Several other methods are provided for querying different aspects of the request object. See **javax.servlet.ServletRequest**[1] for more information. A specialized version, **HttpServletRequest**, for HTTP based servlet requests is also available. See **javax.servlet.http.HttpServletRequest** for more information.

Figure 10-3 shows a simple usage scenario involving a request object.

Response Generation

A request generally warrants a response, and servlets are no exception in this regard.

Servlets make use of **ServletResponse** to simplify this common task. The **ServletResponse** object, commonly referred to as the response object, is in fact provided to a servlet alongside the request object as a parameter to the **service** method.

Output can be written in either binary or character format by obtaining a handle to either a **ServletOutputStream** object or a **PrintWriter** object, respectively. Some of the other methods provided by the **ServletResponse** interface are contained in the following list:

- **getOutputStream** obtains the handle to a **ServletOutputStream** object for binary data.
- **getWriter** obtains the handle to a PrintWriter object for character data.
- **setBufferSize** can be used to establish the buffer size for the response to enable better performance tuning.
- **flushBuffer** flushes the current contents of the buffer.

1. If you are new to Java or unsure about this reference, see the “Conventions” section in the Preface of this book.

Ahmed02.ps 10/4/01 1:03 PM Page 140

For more information, see **javax.servlet.ResponseObject** and **javax.servlet.ServletOutputStream**.

An HTTP specific response object is also available and provides additional capabilities related to HTTP response header formulation. See **javax.servlet.http.HttpServletResponse** for more information.

Figure 10-4 shows a simple usage scenario involving a response object.

Alternatives for Response Generation

If you take a good look at Figure 10-4, you will see several HTML tags involved in the generation of output from the servlet. This represents only one approach for generation of dynamic output.

Another similar but more structured approach is to use libraries of HTML files to generate common headers and footers for the necessary response Web pages, with the dynamic portion of the page still generated much like what was shown in Figure 10-4.

A third and cleaner approach is to use the power of JSP and JavaBeans whenever possible. In this approach, the servlet simply needs to forward to a JSP page that contains all of the necessary presentation information and use JSP technology and JavaBeans to fill in the dynamic content portions of the page. Other than the forward, the servlet has little else to do with presentation except perhaps coordinating the necessary items for the JSP page to successfully do its work.

We discuss this approach further in Chapter 11.

```
PrintWriter out;
:
// set content type
response.setContentType("text/html");
:
out = response.getWriter();
out.println("<HTML><HEAD><TITLE>");
:
out.println("Login Unsuccessful");
:
out.flush();
out.close();
```

Figure 10-4 Generating the response

Ahmed02.ps 10/4/01 1:03 PM Page 141

HTTP Request Handlers

The **HttpServlet** class extends the **GenericServlet** class and therefore inherits all of the standard servlet capabilities. In addition to the basic servlet life cycle methods and convenience method, the more complex **HttpServlet** class adds methods to aid in the processing of HTTP requests. These commonly used handler methods are

- **doGet:** Handles HTTP GET requests
- **doPost:** Handles HTTP POST requests

In the case of **doGet**, there is an additional method used for conditional HTTP GET support (the different HTTP request types are explained later in this section). The **getLastModified** method is like HTTP GET, but only returns content if it has changed since a specified time. This method can only be used if **doGet** has also been overridden and is intended to be used in cases where you are dealing with content that does not change much from request to request.

Advanced Handler Methods

There are several advanced handler methods that are defined as well:

- **doPut:** Handles HTTP PUT requests
- **doDelete:** Handles HTTP DELETE requests
- **doOptions:** Handles HTTP OPTIONS requests
- **doTrace:** Handles HTTP TRACE requests

Unlike the **GenericServlet** class, servlets based on **HttpServlet** have almost no valid reason to override the **service** method. Instead, you typically override these request handlers, which the base **service** method implementation calls when appropriate. The **doOptions** and **doTrace** methods also have virtually no valid reason to be overridden and are present only for full HTTP support. An **HttpServlet** must override at least one method, which usually means one of the remaining life cycle methods or request handlers.

Ahmed02.ps 10/4/01 1:03 PM Page 142

Quick Guide to HTTP Requests

For the most commonly used request handler methods, the following list provides a quick guide of what the HTTP requests are for:

- **GET:** A call to get information from the server and return it in a response to the client. The method processing this call must not have any side effects, so it can be repeated safely again and again. A GET call is typically used when a servlet URL is accessed directly from a Web browser or via a forward from a form on an HTML or JSP page. A GET call shows the data being passed to the servlet as part of the displayed URL on most Web browsers. In certain cases, this might not be very desirable from a security perspective.
- **POST:** A call to allow the client to send data to the server. The method processing this call is allowed to cause side effects, such as updating of data stored on the server. A POST call can be used instead of a GET when forwarding from a form on an HTML or JSP page. Unlike GET, the use of POST hides from view any data being passed to the servlet. Some developers choose to process GET and POST exactly the same, or simply ignore one or the other if they do not want that particular call to be supported.
- **PUT:** This call is similar to POST, but allows the client to place an actual file on a server instead of just sending data. It is also allowed to cause side effects, just like POST. Although available, the use of a PUT call is not very common.
- **DELETE:** This call is similar to PUT, but allows the client to remove a file or Web page from the server. It is also allowed to cause side effects in the same way as PUT. Although available, the use of a DELETE call is not very common.

There is another request not specifically mentioned in the preceding list called HTTP HEAD. This request, although valid in the context of the **HttpServlet** class itself, is actually handled internally by making a call to the **doGet** method, which you might have overridden. It differs in that it only returns the response headers that result from processing **doGet** and none of the actual response data.

The RequestDispatcher Interface

Given the simplicity of servlets, it makes sense to keep each servlet focused on a specific task, and then set up multiple servlets to collaboratively achieve a more complex task. Servlets can take care of the mechanical aspects of such collaborative efforts easily by implementing the **RequestDispatcher** interface.

Ahmed02.ps 10/4/01 1:03 PM Page 143

The **RequestDispatcher** interface provides two key capabilities:

- **forward:** This method allows a servlet to forward a request to another Web component. The servlet forwarding the request may process the request in some way prior to the forwarding. Forward can effectively be used to achieve servlet chaining where each link in the chain produces some output that can be merged with the original request data, and then be used as the input to the next servlet in the chain. This is essentially similar to the concept of pipes in the UNIX world.

  Note that the term "redirect" is sometimes used interchangeably with "forward," intending the same meaning. However, this should not be confused with the **sendRedirect** method found on the servlet response. A **sendRedirect** call does not guarantee preservation of the request data when it forwards to a new page, so it does not allow for the same servlet chaining capabilities.
- **include:** This method permits the contents of another Web component to be included in the response from the calling servlet. The first servlet simply includes the other servlet at the appropriate point in the output, and the output from the servlet being included is added to the output stream. This is similar in concept to Server Side Includes (SSI).[2]

Modeling Servlets in UML

The **GenericServlet** class is usually modeled as a standard Java class with the **<<Generic_Servlet>>** stereotype applied. The presence of the stereotype allows for the servlet to be represented in a compact form and still be easily distinguished as a generic servlet without the need to show the inheritance tree on the same diagram. A generic servlet can include any of the life cycle methods or the convenience method discussed earlier.

A more expanded view of the servlet class showing the inheritance from the **GenericServlet** class can also be used. In most cases, though, the more compact stereotyped class view is sufficient. The compact and expanded representations of the servlet are shown in Figure 10-5.

If the servlet implements the **SingleThreadModel** interface, which controls serialization of the **service** method, the servlet can be shown with the interface

2. SSI allows embedding of special tags into an HTML document. The tags are understood by the Web server and are translated dynamically as the HTML document is served to the browser. JSPs build on this idea.

Ahmed02.ps 10/4/01 1:03 PM Page 144

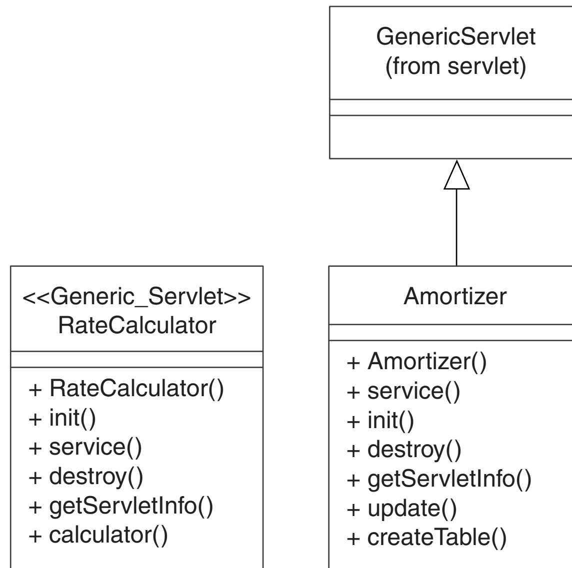


Figure 10-5 Compact and full representation of a generic servlet

to highlight this aspect. Optionally, the servlet can be tagged with *{SingleThreadServlet=True}* instead to clearly identify this on the diagram in a somewhat more compact format.

An example of a servlet that implements the **SingleThreadModel** is shown in Figure 10-6.

The **HttpServlet** class is modeled similarly to **GenericServlet**, but with the **<<Http_Servlet>>** stereotype applied. It can also include the life cycle methods, the convenience method, and any of the HTTP request handlers previously discussed.

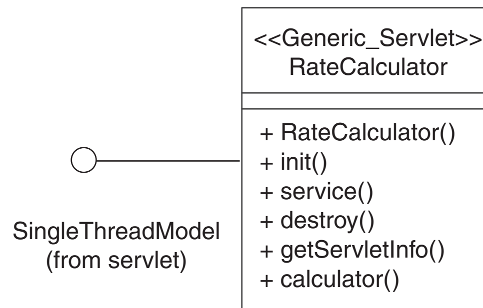


Figure 10-6 Servlet supporting the SingleThreadModel

Ahmed02.ps 10/4/01 1:03 PM Page 145

The **SingleThreadModel** details as well as the tagged value for **SingleThread-Servlet** apply in the **HttpServlet** class exactly the same way as they did for **Generic-Servlet**. As stated earlier, you should not attempt to serialize any of the servlet methods other than by implementing this interface. This interface does not introduce any new methods.

Modeling Other Servlet Aspects

Other aspects of servlets that warrant modeling are servlet forward, servlet include, ServletContext, and Servlet Session Management. The following sections discuss these aspects in more detail.

Servlet Forward

Servlet **forward** is a special kind of relationship, and modeling it explicitly can help clarify the overall application logic. For example, it can shed light on the flow of the processing logic. In complicated **forward** chains, the relationship may be indicative of some algorithm being implemented. Two specific approaches help to identify the overall application logic in this regard.

First, on the class diagram, label the relationships between the servlets that invoke **forward** on other Web components with the <<forward>> relationship. An example is shown in Figure 10-7.

For more complicated servlet chaining, an activity diagram can be used to show the overall interaction. If desired, request and response objects with attributes appropriately updated at specific points can be shown to demonstrate the overall algorithm. See Figure 10-8.

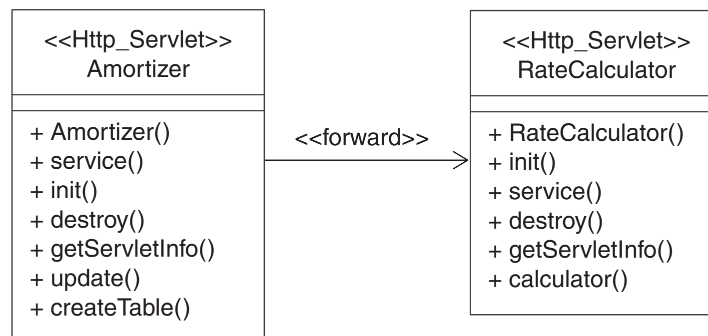


Figure 10-7 Modeling servlet forwarding on a class diagram

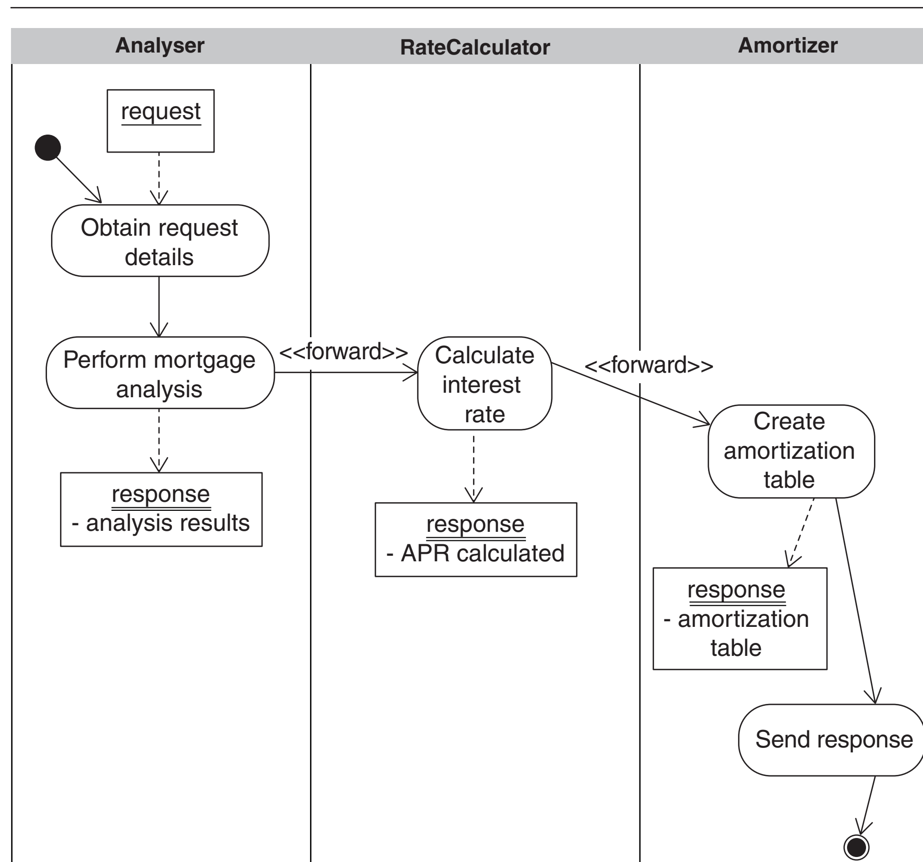


Figure 10-8 Modeling servlet forwarding with activity diagram

In this case, we have labeled the transition with the <<forward>> stereotype to emphasize that it represents a forward relationship between the elements involved. The comments shown for each occurrence of the response object identify what happens as the request and response objects pass through the chain.

Servlet Include

Include is another significant and special relationship as it affects the results produced by a servlet. In fact, **include** may be used as a means to structure and organize the overall output in a modular fashion. Servlet **include** relationships are modeled in the same fashion as the **forward** relationship, that is, as a unidirectional association stereotyped <<include>>. The direction of the association is

Ahmed02.ps 10/4/01 1:03 PM Page 147

from the including servlet to the resource being included. An example is shown in Figure 10-9. In the example, a servlet responsible for creating a mortgage amortization table includes header and footer servlets whose sole purpose is to generate the page header and footer, respectively.

ServletContext

Each servlet runs in some environment. The **ServletContext** provides information about the environment the servlet is running in. A servlet can belong to only one **ServletContext** as determined by the administrator. Typically, one **ServletContext** is associated with each Web application deployed in a container. In the case of distributed containers, one **ServletContext** is associated with one Web application per virtual machine.

The **ServletContext** interface can be used by servlets to store and retrieve information and share information among servlets. A servlet obtains the **ServletContext** it is running in by using the **getServletContext** method.

Some of the basic services provided by the **ServletContext** interface are

- **setAttribute:** Stores information in the context
- **getAttribute:** Retrieves information stored in the **ServletContext**

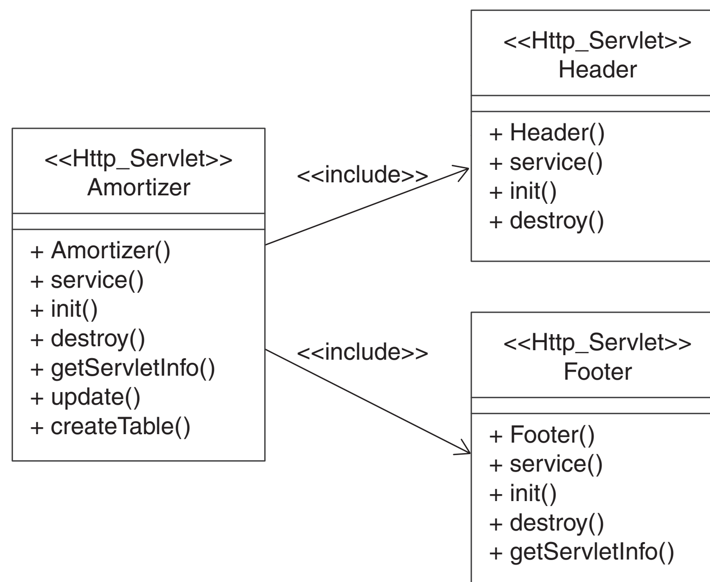


Figure 10-9 Servlet include relationship

Ahmed02.ps 10/4/01 1:03 PM Page 148

- **getAttributeNames:** Obtains the names of attributes in the context
- **removeAttribute:** Removes an attribute in the context

An approach similar to the one discussed for servlet forwarding and shown in Figure 10-8 can be employed to model servlet interactions with the **ServletContext**.

Servlet Session Management

Given the stateless nature of the HTTP protocol, managing repeat interaction and dialog with the same client (such as that required for an ongoing shopping session) poses some serious challenges. There are various means of overcoming these challenges:

- *Hidden fields:* Hidden fields are embedded within the page displayed to the client. These fields are sent back to the client each time a new request is made, thereby permitting client identification each time a client makes a request.
- *Dynamic URL rewriting:* Extra information is added to each URL the client clicks on. This extra information is used to uniquely identify each client for the duration of the client session, for example, adding a "?sessionid=97859" to the end of each URL the client clicks to identify that the request is associated with session id 97859.
- *Cookies:* Stored information can later be passed back to the client repeatedly. The Web server provides the cookie to the browser. Cookies are one of the more popular means of setting up a servlet session.
- *Server-side session object:* Cookies and URL encoding suffer from limitations on how much information can be sent back with each request. In server-side session management, the session information is maintained on the server in a *session* object and can be accessed as required. Server-side session objects are expensive to use, so it is best to use them sparingly.

The Java Servlet Application Programming Interface (API) provides abstractions that directly support some of the session management techniques discussed in the preceding list.

The core abstraction provided by the servlet API is the *HTTP session,* which facilitates handling of multiple requests from the same user.

Figure 10-10 gives an example of servlet session management.

Activity diagrams can be used to model the servlet and session interaction. This is similar to the approach discussed for servlet forwarding and shown in Figure 10-8.

Ahmed02.ps 10/4/01 1:03 PM Page 149


```
import.javax.servlet.http.*;
…
// locate a session object
HttpSession theSession = request.getSession (true);
…
// add data to the session object
theSession.putValue("Session.id", "98579");
…
// get the data for the session object
sessionid =  theSession.getValue("Session.ID");
```

Figure 10-10 Servlet session usage

Servlet Deployment and Web Archives

A descriptor based on XML is used in the deployment of servlets on a Web server. The compiled servlet class, additional supporting Java classes, and the deployment descriptor are packaged together into a Web archive file, also known as a ".war" file.

The deployment descriptor is an XML-based file that contains specific configuration and deployment information for use by the servlet container.

Figure 10-11 shows an example of a vanilla XML deployment descriptor for an **HttpServlet**. Additional required fields in the descriptor are filled in during configuration and deployment on the Web server.

We discuss servlet deployment descriptors and Web archive files and their role in the context of modeling in Chapter 15.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "- / / Sun Microsystems, Inc.
/ / DTD Web Application 2.2
/ / EN" "http: / / java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
  <servlet>
    <servlet-name>LoginServlet</servlet-name>
    <servlet-class>LoginServlet</servlet-class>
  </servlet>
</web-app>
```

Figure 10-11 A simple vanilla XML deployment descriptor for a sample HttpServlet

Ahmed02.ps 10/4/01 1:03 PM Page 150

Identifying Servlets in Enterprise Applications

Now that you have become intimately familiar with servlets, it is time to return to building the HomeDirect online banking example.

At the beginning of this chapter, we identified the need to evolve the control object in the Transfer funds use case by splitting it into two, one focused on the external interaction and the other focused on the internal interaction.

Of course, the question remains: How do you actually arrive at this division of responsibilities? The answer is partly based on understanding what a servlet is capable of doing and the rest on judgment and experience. In general, the role of the servlet is that of a coordinator between the boundary objects and the rest of the system. All the interaction between the boundary object and the composite control class belongs in the new servlet. How you split the interaction that is shown between the control object and the entity objects is somewhat less clear. The key factor to remember is that the servlet is primarily a coordinator; and hence, it should only take on lightweight responsibilities, which could include initiating some business logic. However, actual business logic, computations, interaction with entity objects, and so on would all fall outside of these responsibilities.

With this in mind, let's take another look at the interactions involving the control object as shown in Figure 10-12.

If we look at all of the control object responsibilities, we see that the lower half is comprised of several actions that together form a complete transaction. We decide to separate this part and have it be handled by an internally focused control object, leaving the rest to be taken care of by a servlet. Figure 10-13 shows the result of this division of duties.

In this scenario, the servlet is an example of what the RUP calls a *front component.* A front component is typically a servlet or a JSP that is primarily responsible for processing user input but is not itself responsible for presentation. Rather, it acts simply as an entry point to the application and as a coordinator with other components. Note that the term "TransferPage" is used to generically represent a user interface. We might decide to make this a static HTML page or something more dynamic.

We discuss what to do with the other, internal focused control object in the next chapter.

Of the two types of servlets discussed, an **HttpServlet** appears ideally suited to take on the external interaction role due to the Web-based nature of the HomeDirect interface.

Figure 10-14 expands further on this scenario. There are really two customer actions involved in this use case. The first is where the customer decides

Ahmed02.ps 10/4/01 1:03 PM Page 151

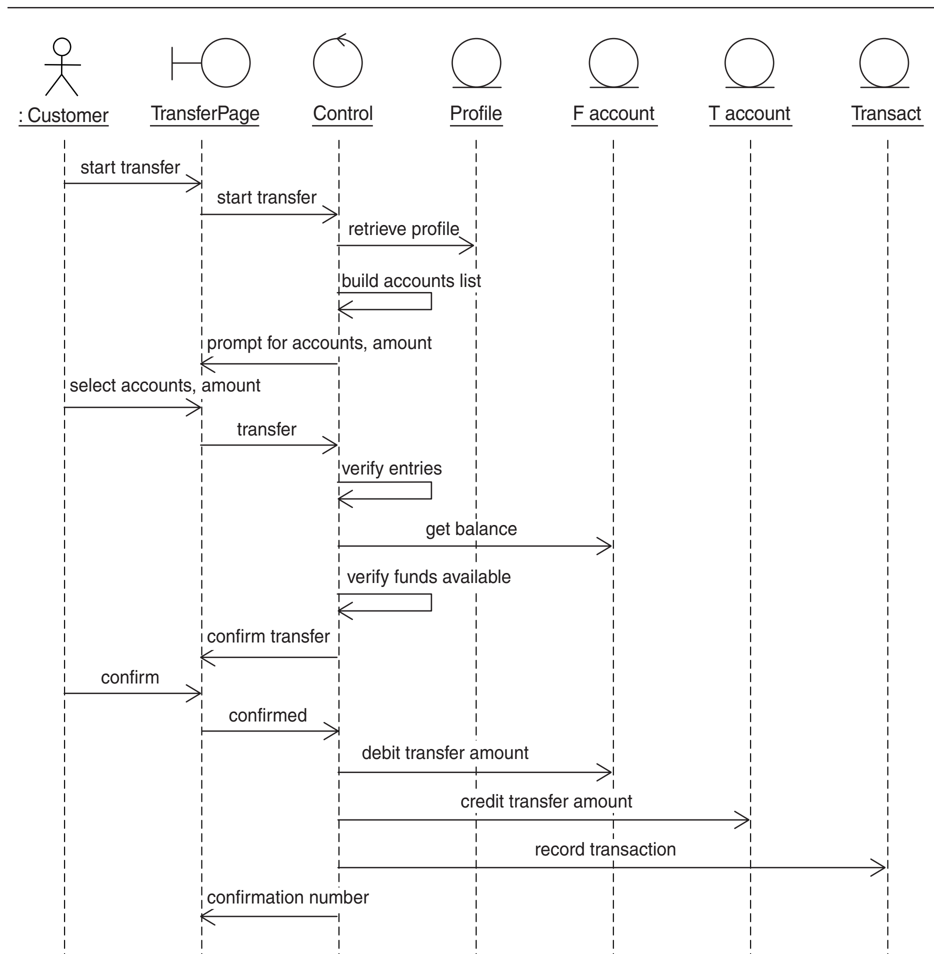


Figure 10-12 Control object interactions

to do a transfer action. This invokes MainServlet, which coordinates the retrieval of the pertinent accounts data and displays this via the TransferPage boundary object. The customer then selects the desired accounts and enters the amount to transfer. Control at this point is forwarded to a secondary TransferServlet, which coordinates the actual transfer action via the internally focused control object.

Figure 10-15 shows the details of the servlets for this example. We purposely have the servlets handling as little processing as possible, offloading most of the

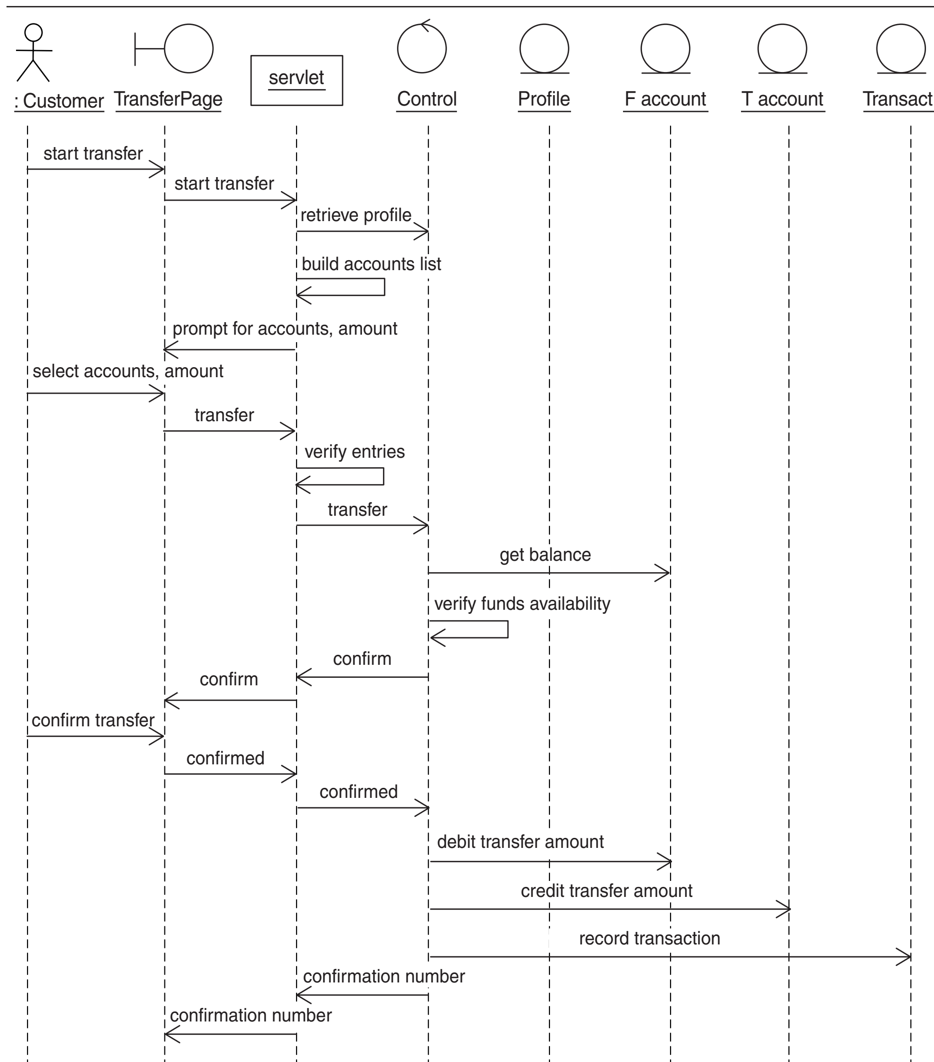


Figure 10-13 Division of responsibilities between the servlet and internal control

work to the other J2EE components, which we discuss in more detail in later chapters covering JSP and EJB technology.

The decision to split up the servlet responsibilities will vary depending on specific needs. In this case, our preference was to minimize the responsibilities of the MainServlet to being a coordinator only. A secondary level of servlets was therefore developed to handle the details of individual use cases.

Ahmed02.ps 10/4/01 1:03 PM Page 153

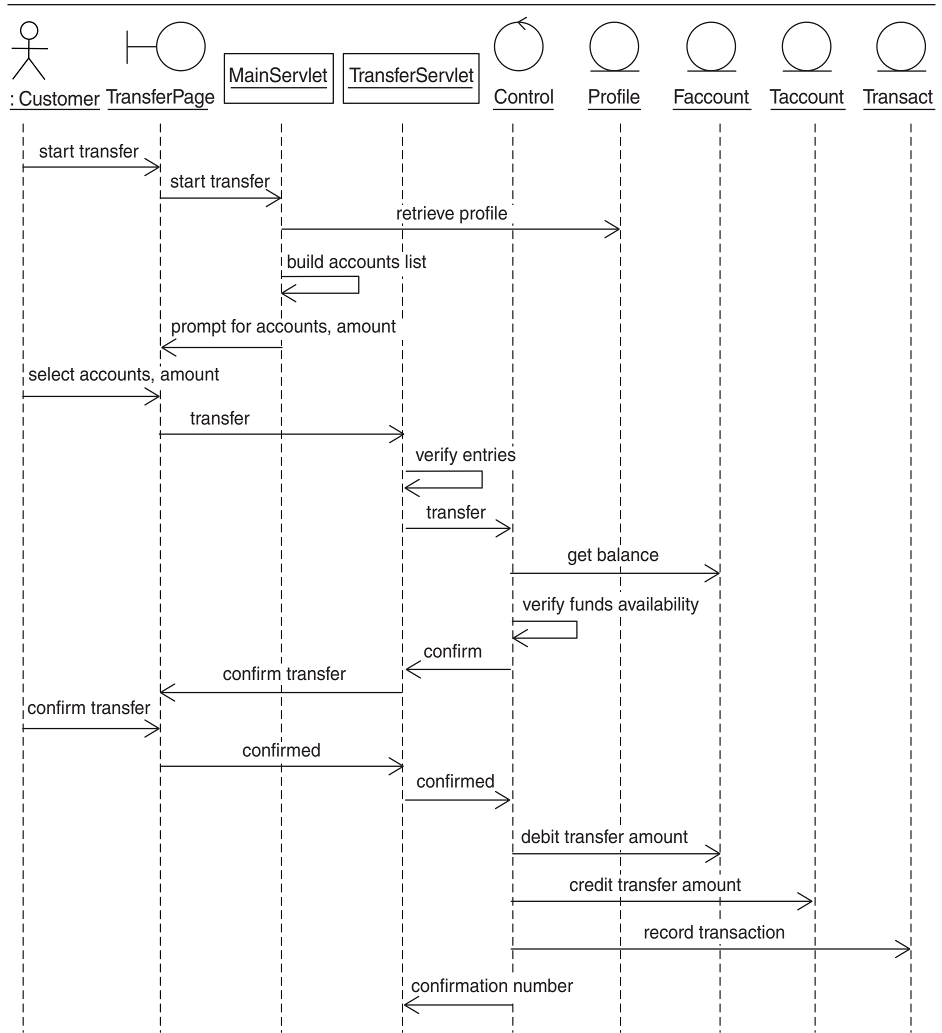


Figure 10-14 MainServlet and TransferServlet division of responsibilities

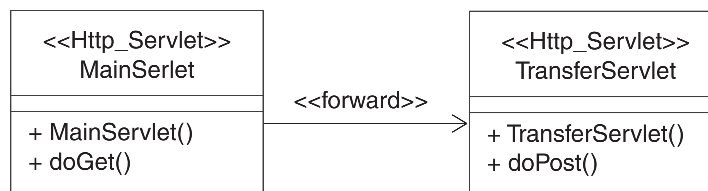


Figure 10-15 MainServlet and TransferServlet details

Ahmed02.ps 10/4/01 1:03 PM Page 154

Summary

Servlets have a lightweight architecture and are ideally suited for request-response paradigms. Servlets are like ordinary Java classes with the exception that specific life cycle methods must exist in the servlet. Specific HTTP request handler methods are used for **HttpServlet**. Two types of servlets, **GenericServlet** and **HttpServlet**, are defined in the J2EE.

Servlets are modeled as stereotyped Java classes. UML modeling techniques can bring special focus on some aspects of servlets, such as forwarding, including, and session management by servlets.

An XML deployment descriptor is required for deploying a servlet.

Dear Dr. Use Case: What About Function Points and Use Cases?

by **Leslee Probasco**
Rational Software Canada

Note: This is a summary of a recent discussion on the chat_rup forum. My thanks to the main discussion contributors: Davyd Norris, Pan-Wei Ng, and John Smith.

Dear Dr. Use Case,

Recently a customer asked me the following question: "If we could estimate the functional complexity of a use case (e.g., hard, medium, or easy), is there a way to then estimate the number of function points those use cases might have?"

Of course, I had a little trouble with this question. My gut reaction was that use cases and function points do not play in the same space (or, at least, they "play the game" differently). Have you ever dealt with this issue? Does Rational have any documentation on using function points versus use cases?

Please point me in the right direction.

Signed,
Pointless About Use Case Estimations

Dear Pointless,

At first glance, estimating use cases (UCs) using Function Points (FPs) might seem like comparing apples with oranges, because we work so hard to avoid functional decomposition with use cases. Function points rely heavily on the physical layout of the system (for example, numbers of tables and fields) and are therefore predominantly data driven. The goals

of the two methods have some obvious similarities; as with UCs, FPs are defined from a user perspective. The International Function Point Users' Group (IFPUG, at http://www.ifpug.org/) defines an FP as "ýmeasured from a functional, or user, point of view. It is independent of the computer language, development methodology, technology or capability of the project team used to develop the application." But that's where the similarities end. To compare the two methods directly, you would have to base your UC grading on the number of tables, functions, and so on, as per the IFPUG standard.

The Rational Unified Process® (RUP®) contains a paper by John Smith called "The Estimation of Effort and Size Based on Use Cases," which looks at some important techniques for estimating development effort and includes an estimation framework based on use cases. This framework considers the idea of *use-case level*, size, and complexity for different categories of systems. Along with other approaches, it discusses a Use-Case Point (UCP) method based on Function Point Analysis (FPA), referencing Gustav Karner's 1993 M.Sc. thesis on this topic (written while Karner worked at Objectory AB, under the supervision of Ivar Jacobson).

Using Karner's Estimation Technique

Several folks at Rational have been using Karner's technique for a number of years now, with good results. Sun and IBM have also publicly posted that they use this technique and have revised the "fudge factors" (described below) based on their experience; in addition, the technique has been documented in several books. *The technique's main benefit is that it can be performed in your head from a use-case model survey* (i.e., *very* early in the lifecycle, with *very* low precision), before the use cases have even been written, provided you have some idea of how many scenarios are contained within each use case (I always include a list of key scenarios in my brief use-case description).

Basically, Karner's technique is similar to FP techniques in that you:

1. Count key aspects of your requirements to form an *unadjusted point count*.
2. Use several sets of questions about your team and their environment to *create a fudge factor.*
3. Multiply your original count by the fudge factor to come up with *an adjusted point count*, which then translates into a person-hour LOE (Level of Effort) estimate.

Karner proposes using a fudge factor set very similar to the FP method factors but with slightly different weightings, and he proposes 20 person-hours/UCP for LOE estimates. By looking at Actors and use cases for input, you can derive a point count as follows:

1. **Rank Actors** as simple (1 point), average (2 points), or complex (3 points):
   - *Simple*: a machine with a programmable API

- *Average*: either a human with a command line interface or a machine via some protocol (no API written)
- *Complex*: a human with a GUI

2. **Rank use cases** as simple (5 points), average (10 points), or complex (15 points):
    - *Simple*: fewer than 4 key scenarios or execution paths in the UC
    - *Average*: 4 or more key scenarios, but fewer than 8
    - *Complex*: 8 or more key scenarios
3. **Calculate unadjusted use-case point (UUCP) count, a fudge factor, and an adjusted use-case point (AUCP) count.**

    For the system under scrutiny, add up all the points to get the *unadjusted* (UUCP) count.

    Then, multiply by the technical and environmental fudge factors to get the *adjusted* (AUCP) count.

    Note: These counts are COCOMO[1]-like; when using this approach with COCOMO, use the *unadjusted* count (based on the assertion that UUCP has the same "weight" as unadjusted function points, which are fed into COCOMO).
4. **Convert the totals from Step 3 to an LOE estimate** based on calibrations of your team/organization. Use 20 person-hours/AUCP as a start. *Note*: The folks at Sun report that in their experience the rate should be closer to 30 person-hours/AUCP; I have found it's somewhere in the middle, but highly organization dependent.

Step 1 above gives straightforward definitions for ranking Actors, but for UCs (Step 2) you need to apply discrimination to determine what constitutes a "key scenario." A key scenario in this case is the major way a use-case instance can be executed. For the most part, this would correspond to a major alternate flow, but not always. It could be that several alternate flows combine into one key scenario, or that a particular exception flow is very complex and so becomes part of a key scenario. The instruction in Step 2 also assumes that your use cases are leveled (with respect to level of detail) in similar ways to other projects. As mentioned in the RUP, a midsize project of about 10 developers over 6-8 months should have about 30 use cases. This fits with the idea that *an average UC has 12 UCP, and each UCP requires 20-30 hours. That means a total of 240-360 person-hours of effort per use case.* So 30 use cases would require approximately 9,000 staff hours (10 developers for 6 months). Note, however, that a very large project with 100 staff for 20 months would NOT start with 1,000 use cases (pro rata), because of the level issue.

It is important to make sure the UCs are not too decomposed, and, equally important, not too high level. Make sure you are dealing with a *system* use case, not a business use case. The test question I ask is: "Can the key scenarios be realized by collaborations of 7ý2 classes? "This

applies to the analysis level; there could be several more classes when you look at a fully elaborated design-level collaboration. If the number of classes starts to explode, and you start to aggregate the classes into subsystems, then your use cases may be at a different level. Vastly different numbers of use cases will skew your results one way or another, but the method can be recalibrated as described above (by reapplying the estimation to similarly leveled use cases) to take this into account for each organization's style.

Estimates for Simple and Complex Systems

Using Karner's technique to estimate effort for simple and complex systems (as shown below) yields a range of values that correlate well with empirically based figures given in the RUP of about 150-350 hours per use case.

- **Example: Simple System**
  The simplest system (UC rank = 5) would be a human initiating a simple use case driven by a command line interface (Actor rank = 2). Based on the formula specified in Step 3 above, this would give a UUCP count of 2 + 5 = 7 points. Using the formula in Step 4, at 20 person-hours per UCP, this yields about **140 person-hours**.
  Note: A typical fudge factor for a new team would add between 10 and 20 percent to the effort estimate.
- **Example: Complex System**
  A complex system (UC rank = 15) would be a human initiating a complex GUI-driven (Actor rank = 3) use case; this would add up to 18 UUCP or about **360 person-hours**.

Do It in Your Head

As you can see, you can apply this technique in your head as you walk into a project. I find it particularly useful when I get thrown into a problem project headfirst. I do a quick mental check of how many people should be on the team and where in the schedule they should be.

Others have calibrated the technique for their teams with very good results (search the Sun and IBM developer sites to see papers on the use of UCPs). The key point, however, is that *with very little effort, you can use this technique to get a very early gross estimate.* And it will be just as accurate (or inaccurate) as any other method you could use at this early stage in the project.

The best way to use the technique is to do a quick calculation and then move on to more effective methods of estimation, such as actually doing some useful work with your team and seeing how long it takes. Your initial estimates can then be calibrated against these findings and refined as you move further along.

Compensating for the Technique's Deficiencies

Most of the problems I have seen in understanding and applying Karner's

UCP technique revolve around evaluating the complexity of use cases, or rather, defining what is a key scenario. For example, should a use case that allows me to do CRUD (Create, Replace, Update, Delete) be considered as 1 UC with 4 key scenarios, or is it actually 1 use case with 1 key scenario, as the other scenarios are so similar? When such questions arise, I turn to Larry Constantine's idea of an "essential use case." In this context, *essential* does not refer to the crucial use cases in your system, but rather to the essence that defines what each use case is about. You should be able to look at a use case and determine which scenarios shape its very essence, as opposed to those that fill out and complete it. (See Larry Constantine and Lucy A. D. Lockwood's book, *Software for Use*[2], for a further discussion of what constitutes an essential use case.)

Another problem I have with the UCP technique is its vagueness about how many use cases you will have and how granular they should be. There are plenty of debates about what constitutes a use case, and how much it should be refined. For an appropriate use-case count, I typically go by "gut feel" and averages I have cultivated over the years, based on my own and others' experiences. The RUP says that an average IT project (business, not technical) of about 6-8 months and 10-15 staff will consist of somewhere around 30 use cases, and what I see in practice confirms this. One of the largest projects I worked on was a 4-year, 300-person project consisting of around 280 use cases (FP estimates of this project put it at around 9,000-14,000 FPs!).

With respect to the issue of use-case granularity, the smallest useful use case I have seen was only a half page in length, and the largest was more than 120 pages! Before you start yelling, this huge use case was actually very simple: It had a main flow that was 2 pages long and 50 alternate flows that started and ended at exactly the same points. Basically, the use case documented the management of company rules and restrictions, and there were 50 or so different types of rules a user could choose from; average reviewers read the 2-page main flow and then picked a couple of rule types that interested them. During development, the rule types were prioritized, and a few new rule types were added in each iteration (in fact, some low-priority rules never did get implemented).

Yet another problem I have with Karner's technique is that estimates change based on the type of project. For example, the GUI for a Web application causes related Actors to be ranked as complex, as would the geographic mapping GUI in a command and control (C2) project. However, it could be argued that the internals of a Web application are based on well-known component infrastructures (such as .NET or J2EE), so its implementation would be trivial compared to the extremely complex inner workings of a C2 system. In this situation several things come into play. Compared to a Web application, a C2 system will have many more use cases, and each of these will tend to have more key scenarios than Web application uses cases; this increases the UCP for C2 systems. In addition, the technical and environmental complexity of a C2 system greatly increases the fudge factors for the project, making them much higher than those for a corresponding Web application.

Beyond Early Estimates

Although the early estimates you get with this technique can give you a good start, the most important thing to remember is that they are only gross estimates. As you proceed with the project, you need to:

- Factor your own experiences into the mix.
- Start refining your figures as soon as you have more information.

Once you have an Analysis Model available, it is possible to identify boundary, control, and entity classes, or, even better, design subsystems. You can estimate the effort required to implement these by using analogous figures from past projects.[3] (At this level of analysis detail, you can also start employing the techniques presented in a paper entitled "The Estimation of Effort Based on Use Cases"[4] by Rational's John Smith, or use well-known techniques such as those in COCOMO II.[5]

On the other hand, rather than coming up with a better way to nail down costs before you start the project, perhaps you should spend most of your effort on changing the organization's/team's underlying attitude, so you can avoid premature estimation in the first place. Premature estimation is a very nasty condition that leads managers to commit the team to unrealistic budgets, which results in everyone on the project becoming hot under the collar, resources being spent before completion, and requirements that are only half satisfied. It is possible and potentially useful, however, to make estimates at any time, provided you recognize the attendant error bounds on your estimate. Also, the project manager should -- if required to make budgetary projections-- either provide for contingencies or establish a scope management regime that will prune functionality to fit the budget. This is the real world, after all.

Hope this helps.

Usefully yours,
Dr. Use Case

Notes

1 COCOMO is the Constructive Cost Model, originally developed by Dr. Barry Boehm and described in his classic work *Software Engineering Economics*, published in 1981 by Prentice-Hall.

2 Larry L. Constantine and Lucy A. D. Lockwood. *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*. Addison Wesley, 1999.

3 See Joe Marasco's article on "Commitment" in the May 2002 issue of *The Rational Edge*.

4 Available online at
http://www.rational.com/products/whitepapers/finalTP171.jsp

[5] See http://sunset.usc.edu/research/COCOMOII/.

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

Copyright Rational Software 2002 | Privacy/Legal Information

Copyright Rational Software 2002

Integrating Rational Quantify and Rational Purify with the SAS System

Part II: Migrating the SAS Development Platform to NT

by **Claire Cates**
Senior Manager, Advanced Performance Research
SAS

In Part I of this two-part series, published in last month's Rational Edge*, I described all the activities involved in integrating Rational® Quantify® and Rational® Purify® into the SAS UNIX development environment, including product selection, challenges, and resolutions. This part focuses on the process of migrating the implementation to our Windows NT development environment. It describes how SAS made the tools easier for developers and testers to use by taking advantage of published APIs to integrate Purify and Quantify into our nightly build process, debuggers, and runtime environment.*

Several years after we integrated Purify and Quantify into the SAS® System, SAS decided to move the development environment to Microsoft Windows NT. Our developers and testers insisted that Purify and Quantify be available on this new development platform, and we hoped to use a lot of what we had learned by integrating the products into the SAS System on UNIX. Little did we know that we were going to see a new variety of obstacles to overcome.

First, we decided to look into the problems we *thought* we might have. We were happy to find that our tasking/thread issue actually disappeared, because on NT we use the standard OS thread calls. As it turned out, using the OS thread interface was a wise choice, as Rational Purify and

Quantify support only native NT threads on Windows. So, we turned our attention to the memory subsystem and found that the `#ifdefs` that were in the code base to support our alternate memory interface on UNIX worked well with the NT implementation. We simply changed the Purify API calls to the NT version of the Purify routines. Our biggest concern then became our non-standard image loading support. Instead of shared libraries, we now were concerned with our non-standard usage of `dlls`. We talked with Rational technical support to determine a way to keep Purify and Quantify from instrumenting the assembler linkage code, but unfortunately, they could suggest no solutions that kept individual areas of code from being instrumented in the NT version. There are ways to change the type of instrumentation for particular `dlls`, but the linkage code is linked into every `dll`; therefore, we could not use this solution. Yet, as it turns out, the way Purify and Quantify instrument the code is different for NT than it is for UNIX. The major difference is that on UNIX, Purify and Quantify instrument object files, whereas on Windows they instrument already linked executables. The NT version inserts more calls and depends less on individual registers. So when we created a test case allowing Purify and Quantify to instrument the linkage code, we realized that this was not going to be a problem after all. Suddenly it looked like integrating the product with NT was going to be easy. A first test of our initialization worked well and made us even more excited about an easy solution. Unfortunately, it turned out to be more difficult than we had initially expected.

Investigating Random Failures

There are many similarities between our NT and former UNIX development environments. The compiler on the NT development platform is built by SAS and does extra checking on our code in order to detect problems that cannot be detected until the code is ported. The NT development platform also contains the same tracking abstraction that is used on UNIX. Two tracks were added to the NT platform, one for Quantify and one for Purify.

But our initial hopes for an easy solution were soon crushed when random failures started occurring. The first problem we found was the use of the variable *errno* within our code. This symbol confused the Purify engine and caused several instrumentation problems. We changed the code and ran our tests a little further, yet we still had many random failures of the instrumented application. Although we suspected a variety of areas, none of those proved to be the cause of the problems. It took a tremendous amount of debugging to find the critical problem: Some developers had substituted alternate routines for standard C library routines such as `printf`. This overriding of standard C routines is not supported by either Purify or Quantify. The instrumentation engine for Purify and Quantify recognizes this user defined routine as the original system C library function and replaces a portion of the code in the instrumented executable file. Often these substituted routines did not have the same parameters defined as the original routine. So when our C code called into what was supposed to be a local `printf` routine but instead was the code substituted by Purify and Quantify, memory was overwritten. These overwritten memory areas were the cause of many random failures. After discovering this class of problem, we scanned all of the millions of lines of source code

to find any other routines with C library routine names. We then changed each of these routine names as well as all calls to these routines. After these changes were made, Purify ran with the NT version of the SAS System. Our testers were ecstatic.

Making Adjustments to Quantify for NT

Unfortunately, we still had problems with our Quantify introduction. The instrumented application was stable, but when Quantify was computing the results, an error would occur that kept Quantify from saving the data. We guessed that this was another memory overwrite. Eventually, we noticed that if we selected function level instrumentation instead of line level instrumentation, the system produced correct results. Function level instrumentation enables measuring times for functions but doesn't provide the exact distribution of recorded times on the lines of code. This gave us an intermediate solution, but we truly needed the line level instrumentation. Therefore, we started serious debugging to determine the cause of this problem. After some testing, we discovered that we could make it through part of the initialization code and then stop the system, and Quantify would compute the results correctly. Yet we knew that if we executed all of the initialization code, Quantify would not produce any results and would instead terminate the session. Our only choice was to find the exact place in our initialization code that caused Quantify to terminate during the computation phase.

This took a while, because a large amount of code is executed during our initialization phase. Eventually, we did pinpoint the place: a line of generated C source used by one of our subsystem tools. Some of our developers had placed the generated source into a `header` file so that it could be included in one of the C modules. The problem in Quantify was caused by this included `header` file within the C source code because Quantify gets confused about the number of lines in the routine when a `header` file with C source is included in a C file. Quantify's internal line counter array ended up being created too small, and then it was overwritten when this code was executed. That is why Quantify worked fine for function level analysis but not for line level analysis. Once we found this problem, we worked with our local compiler developers to insert full debugging information for each line in the included `header` files, so that Quantify would recognize the exact number of source lines and allocate the correct size for their internal structures. The compiler added a special option named `-quantify` that triggered this fix, so we added a `-quantify` compiler option to the build in the Quantify track.

A series of compiler changes were needed in the NT Purify and Quantify track. Each change was triggered by either the `-quantify` or `-purify` compiler option. Besides the `header` file change, we made additional changes to the compiler to generate a different code sequence for the `switch` statement that was causing problems with Quantify and to memory assignment statements involving structures that were causing false positives within Purify. We also modified the location of the source file in the debug data. Initially, the source pointed to the UNIX version of the source code, because all the compiles are done with a cross compiler, and

the source resides on a Free BSD machine. Unfortunately, pointing to the source code on UNIX kept Purify and Quantify from easily locating the source files. We discovered that the compiler must change the source path for the file in the debug data to use the NT path to the Free BSD machine instead of the UNIX path. Once we changed the location of the source file in the debug data, Purify and Quantify could easily locate the source from the NT machines. This default source location was added to the track variables for the Purify and Quantify tracks. Finally, we turned off the definition of the `DEBUG` symbol for the same reasons we turned it off under UNIX.

Initially the SAS code generator was turned off for NT. This solution was not optimal because the interpreter is very inefficient, and it became the performance hot spot for many tests. The first solution we investigated was to specify a special, undocumented instrumentation level for the `dll` that contains the interpreter by setting the `-quantify exclude must="dllname"` option in the default settings dialog. With this option set, and by filtering the `RuntimeGeneratedTimedFunction` routines produced when this option is used, we were able to get somewhat useful data. Unfortunately, this didn't work well when the code that needed profiling depended heavily on our internal code generator. It was time to start looking at what could be done to make the SAS code generator work with Quantify.

By running the code generator with Quantify, we noticed that Quantify seemed to work fine if the code generator did not call back into instrumented code. We also noticed that the code generator seemed to work in some instances in which the code generator called into instrumented code. The major problem was that the call trees were distorted. For instance, if routine *a* called into generated code, and the generated code called routine *b* and then routine *c*, then the call tree produced by Quantify said routine *a* called routine *b*, and routine *b* called routine *c*. We needed the call tree to say that routine *a* called routine *c*. Finally, we noticed that if the generated code called *b* and *c* in a large loop, then the run with Quantify would eventually crash. We assumed that the NT version was doing something very similar to the UNIX version. After careful debugging and experimentation, we determined that by inserting a call to an undocumented Quantify routine `_x_q_fn_exit` after every call from generated code, Quantify would report correct results. This extra call was inserted into the generated code stream produced by the internal code generator. Of course, this call was only added in the Quantify track.

Learning about the `-quantify-exclude-must` option also gave us the knowledge to fix another problem we were noticing. One of the subsystems within the SAS System has a pseudo code version of an object-oriented system. The problem stemmed from the fact that all method calls go through one routine, `wobmth`, which then calls, via a function pointer, to the correct routine. The call tree produced when several different routines called into `wobmth` became confusing. What we desired to see was routine *a* calling method routine *b*, and routine *c* calling method routine *d*. Instead, what we saw was routine *a* calling `wobmth`, which in turn called both routine

b and routine *d*. We saw the same thing with routine *c*. There was no way to determine that *a* actually was calling *b*, because everything was funneling through `wobmth`. `Wobmth` was located in a library that was being linked with every module, so we created a new `dll` that exported the entry point `wobmth` and contained the source code for the routine. The build scripts were then changed to look for this `dll` to satisfy the calls to `wobmth`. Also, the `dll` that contains `wobmth` was added to the list of modules to be instrumented with the "Exclude" instrumentation level `-quantify-exclude-must` option. With this change, Quantify now reports that routine *a* calls routine *b*, and routine *c* calls routine *d*.

Finally, we noticed inconsistencies between the function detail and source code analysis windows. After careful examination, we discovered that the filters that had been applied to the run did not propagate into the source code view. The time and percentages listed in the source code window do not remove any filters that have been applied to the run. This inconsistency was verified by Rational and documented explicitly in our internal Quantify documentation. It is imperative that developers know about this so that the confusing results do not lead them to mistrust the Quantify data.

Adjusting Purify for NT

The version of SAS that runs with Purify on UNIX used a global suppression file to remove the unwanted information from the report, and we had hoped to have a similar implementation on NT. In order to have consistent reports for all users, we set up one suppression file for all Purify users. However, Purify for NT uses filters,[1] and as we attempted to use the `/FilterFiles` option, it seemed not to work when the file was not local on the disk or did not have the same name as the executable running. We had to copy this filter file to all the workstations to make use of it. To avoid this administrative overhead each time the global filter file changes, we chose to change our SAS invocation tool so that it would copy a filter file from a global location and name it to match the executable. This new filter file is placed in a location that causes Purify to recognize it as the filter file for the SAS application. Unfortunately, this new file overwrites any existing filter file the user may have created for the SAS

Rational Purify and Quantify Benefits at a Glance

Rational Purify...

- Easily finds all types of memory problems.
- Pinpoints exactly where the problem occurs.
- Displays where the memory in question is allocated.
- Supplies API functions to call for customizing your application.
- Works well with a debugger.
- Can store data in text files for custom analysis.
- Has a simple user interface.

Rational Quantify...

- Displays performance data on the source code.
- Leaves results unperturbed by processes on the system.
- Provides detailed data on each system function.
- Has an easy-to-use interface.
- Can store data in text files for

application and makes it awkward for users to add their own custom filters for the SAS application. To get around this, users must create their own global filter group for their workstations in the filter manager that is separate from the sas_exe filter area.

custom analysis.

- Supplies API functions to call for customizing your application.

Further Adjustments

The NT development environment, like the UNIX environment, uses its own internal version of a debugger. The debugger in our NT environment had problems locating the images that were being loaded by Purify and Quantify. In the UNIX environment, the shared library names are appended with a constant string containing the product name and information concerning which version of the product is being used. The NT version appends to the dll name information on the uninstrumented dll location. The dll location changes when local playpens are used, so the prefix is not constant across loaded dlls. Our debugger developer added code to scan the image name for the Quantify and Purify tags, and when the tags are found, any appended information is stripped. This allows developers and testers to set breakpoints just as they would in the regular development track. We also added an option to the debugger to pass access violations and not report them. Purify forces first-chance access violations when running under the SAS internal debugger as part of the standard operations performed by Purify. These first-chance access violations must be ignored in order to make a Purified version of SAS run under the debugger.

The NT version of Quantify does an excellent job with multiple threads; each thread is marked in the Threads tab of the Run Summary window. Unfortunately, many of our threads are created by a common piece of code, so we cannot distinguish the threads from each other. Originally, we had added calls to the QuantifyPrintf routine in this common code so that in the Details tab we could see mappings of the names that Quantify assigned to the threads. Here are examples of the annotations we added for the mappings:

```
Annotation: htthread.554 = IDLETASK
Annotation: htthread.740 = EVENTTSK
Annotation: htthread.714 = sasxkern
Annotation: htthread.778 = Session
Annotation: htthread.820 = TKSRV
Annotation: htthread.718 = Object S
Annotation: htthread.110 = DOSPEIN
Annotation: htthread.7f8 = OLPMVASV
Annotation: htthread.844 = OLPWTSK1
Annotation: htthread.68c = OLPWTSK2
```

The latest release of Quantify for NT has an API routine called QuantifySetThreadName. It makes the thread names appear in the Quantify output windows, which is a welcome improvement.

Like the UNIX version, the NT version of Quantify supports the SAS options statements, naming of Purify threads, and Leak detection calls. We left this code virtually unchanged, merely changing the UNIX style function names to NT style function names.

Worth the Effort

Although integrating Rational Purify and Quantify into the SAS System was not easy, it was worth the effort. Once we accomplished the UNIX integration, it was clear to developers that the tools were needed on the new NT development platform, so we persevered despite the challenges we encountered along the way. SAS is a large, complex system; if we were able to accomplish this integration, then it should be possible to integrate these Rational products with any UNIX or NT software system on the market. Although each system integration will be different, the rich set of options and API calls supplied by both Purify and Quantify should give users enough flexibility to work around any problems they may encounter. Then, once the systems have been integrated, testers and developers will readily see how well the products perform and demand their usage within the product development lifecycle.

Again here are the benefits that both our UNIX and NT teams have realized from our own integration of Rational Purify and Quantify (as detailed in Part I of this series):

- During development, Purify has saved untold time that we used to spend hunting down the source of problems. Our other tools could tell us that memory was being overwritten, but we couldn't pinpoint where. Debugging was a huge binary process that could take days; now it takes minutes or hours. Also, Purify finds problems we didn't know we had; it picks up errors that our internal memory system did not, such as Array Bound Reads (ABRs) and Array Bound Writes (ABWs). So we avoid debugging time later on, too. Ultimately, that means we can get our product to customers much faster.
- As they write code, developers use Quantify to test and comparison test various algorithms before making a permanent change. If they see big differences in performance, then they know to choose the more efficient option.
- Now that testers have ready access to Purify for checking new code, we can avoid passing on problems to our customers. Testers run checks on builds almost nightly to ensure that any new code the developers pushed did not introduce new errors.
- Developers use Quantify to troubleshoot existing implementations; if a customer complains that their SAS application is too slow, then our developers run the job through Quantify to identify code that's consuming excessive resources. They also run it through Purify to check for and correct memory problems. The end result of this work is a faster, more efficient system for our customer, which saves them time and money.
- The integration has allowed us to automate a portion of our testing and our performance analysis. We've created back-end processes

that use the text data files Quantify and Purify produce.

I hope this "story" of our integration will inspire you to consider using these products for your own development efforts.

Notes

[1] Purify for Windows uses two types of filters to hide unwanted messages from the report. The first type of filter is associated to particular executable files. When the main executable is run in Purify, these filter files are applied for every report of that run to which they are associated. When local filters are associated to a shared library or other shared executable module, they are applied to every application that uses the executable shared module with a Purify filter. The second type of filter is a global filter. By default, global filters are associated with every application run on the workstation on which they were created. All filters are saved in a proprietary binary Purify filter file and can be created, managed, engaged, and disengaged from within Filter Manager in Purify.

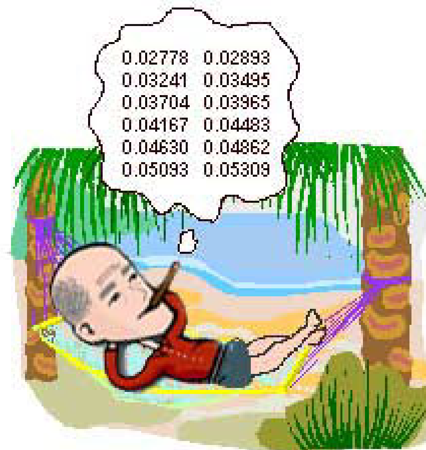
For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!

Copyright Rational Software 2002 | Privacy/Legal Information

Desert Island Math

by **Joe Marasco**
Senior Vice President and General Manager
Rational Software

In the heart of the baseball season, our old friend Roscoe Leroy returns with a tale from his past. Whilst shipwrecked on a desert island, he came across a seemingly simple math problem. He takes us through the solution, makes some observations, and poses a challenge at the end for our readers.



Roscoe Sets the Stage

"Remember the time I got shipwrecked in the South Pacific with my pal named Monday?" Roscoe began. Monday, it should be noted, was Roscoe's answer to Robinson Crusoe's Friday -- his traveling companion and overall helper. Roscoe and Monday had spent many a night under the stars, and, if two guys had to be marooned, those two had a chance of surviving together without killing each other.

"Yeah," I replied. "You guys were really lucky; as I recall, you were the only survivors. The island was tropical; there was plenty of food, and you had shelter from the elements. All you had to do was wait to be rescued."

"Well," snapped back Roscoe, "all we *could* do was wait. There was no way we could accelerate anything. Our biggest problem was boredom; there were no books in the flotsam and jetsam, and we desperately needed a way to pass the time without going bonkers."

It turns out that in going through their salvage inventory, Roscoe and Monday discovered a mint set of baseball cards. After awhile, Monday suggested they use the statistics on the baseball cards to construct two teams, so that they could conduct some "fantasy" baseball games to pass the time. Fortunately, they had an ample supply of pencils and paper. Roscoe had even rescued his pocket slide rule.

"As soon as Monday came up with the idea," said Roscoe, "I was keen on it. It would give us something to do, and it would be a harmless form of competition. So I set about figuring out how to create the game.

"And that is how the thorny problem came about."

Simulating the Batter

"In all these simulations," continued Roscoe, "you need to devise a sort of random number generator."

"Well," I said, "what devices did you have at your disposal?"

"Not much," responded Roscoe. "All we had was three identical dice of the usual variety, with one to six spots on each. Still, I figured it would be easy to simulate probabilities with them, since that was really what we needed to do. We decided that the simplest thing to do was roll the three dice simultaneously, add up the number of spots, and use that total to determine success or failure."

"Probabilities? I'm not sure I understand," I said.

"Sure, probabilities! That's how the fantasy baseball concept works. For example, assume you have a hitter at the plate with a batting average of .250. At the lowest level of sophistication (forget, for a minute, about walks) we need a way of randomly deciding if he gets a hit for this at-bat, so we need to create a random event that has a probability of occurring 250 times out of a thousand." I had never known Roscoe to be that interested in probability and statistics, but I was about to be impressed.

"Now in reality, it's more complicated. In our game, for example, we ignored the quality of the pitcher, and complicated situations like sacrifice flies, and so on."

"Well, assuming you have made these simplifications, how do you actually simulate a batter's appearance at the plate?" I asked.

"We have only two things to think about," Roscoe replied. "Plate appearances and official at-bats. When a player gets a walk, it counts as a plate appearance, but not as an official at-bat. So basically, you need two numbers: the percentage of plate appearances that yield official at-bats for that player, and then his batting average. Suppose, for example, that a player walks in one out of every ten plate appearances.[1] What you would do is then first determine if the player walks, by asking for a successful trial of an event with probability 0.1. With the three dice, you'd need to know what combined number on a given throw has that probability. So imagine that you roll the dice, and you get that total -- then, bingo! The batter walks to first base, and you're done.

"On the other hand, if you don't roll that total, then the player does not walk; instead, he has an official at-bat. Now you take his batting average, say .250, and you roll for that probability. If you're successful, he makes a hit; if not, he is out. Then, of course, you can roll for what kind of hit

(single, double, triple, home run) by using those statistics per at-bat. And so on. I used batting average as the generic example, but basically you can refine the simulation to your heart's content, depending on how many of the "corner cases" you want to include.[2] It will always, however, come down to simulating an event with a certain probability. And, because sometimes we will need probabilities for things other than batting average, we need to be able to cover the entire range from zero (total failure) to one (certain success.)"

"Fair enough. I get the gist of it. So what's the problem?" I replied in turn.

"The problem," said Roscoe, "is this: Can we figure out how to simulate probabilities using a device as simple as three dice, and still have enough granularity to make the simulation reasonable? For example, if we can only simulate .250, .500, and .750, we don't have enough values to do a good simulation."

"Well, if you throw all three dice simultaneously, you get totals that range from 3 to 18; that's sixteen different outcomes, so that's a start," I responded.

"Indeed," said Roscoe, "that is exactly how we proceeded."

First Steps

"Monday got right on the basic combinatorics. With 3 six-sided dice, there were 6 x 6 x 6 possible outcomes, or 216 possibilities. Of course, as there were only 16 different totals, many of these yielded the same answer. So Monday constructed the following table:

Total	Number of Ways
3	1
4	3
5	6
6	10
7	15
8	21
9	25
10	27
11	27
12	25
13	21
14	15
15	10
16	6

17	3
18	1
Sum	216

"Looks good to me," I replied. "There is some obvious symmetry. For example, both 3 and 18 come up exactly once, as you would expect. And 4 and 17 are the same, and so on. The most frequent occurrences are 10 and 11, as there are lots of combinations that will yield those totals. And the number of total ways adds up to 216, so you can't be too far off the mark. But it looks like there are, so far, only 8 distinct probabilities in the offing."

"Appearances can be deceiving," said Roscoe. "But let's add in the probabilities we have so far to double check our work." He then produced the following table:[3]

Total	Number of Ways	Probability
3	1	0.00463
4	3	0.01389
5	6	0.02778
6	10	0.04630
7	15	0.06944
8	21	0.09722
9	25	0.11574
10	27	0.12500
11	27	0.12500
12	25	0.11574
13	21	0.09722
14	15	0.06944
15	10	0.04630
16	6	0.02778
17	3	0.01389
18	1	0.00463
Sum	216	1.00000

"Whoop de do!" I exclaimed. "Roscoe can divide by 216."

"Calm down, Sonny," said Roscoe. "The fun is only beginning."

Second Steps

"Well, of course, it should be obvious that if you want to simulate a probability of 0.00463, all you require to be successful is that the shooter roll a 3. You could ask him to roll an 18, which is the symmetrical result; however, we are looking for distinct probabilities, so we pick one or the other. We pick 3." Roscoe waited a second, and then sprang his first surprise.

"But what happens if you define 'success' as rolling either a 3 or a 4? Now the probability of rolling a 3 is 0.00463, and the probability of rolling a 4 is 0.01389, so the probability of rolling either a 3 or a 4 is simply 0.00463 + 0.01389, or 0.01852. So you can see that we can create some new distinct probabilities by considering multiple totals as 'successful.'"

"Let me see if I get this," I replied. "To simulate 0.00463, I demand that the shooter roll a 3. To simulate 0.01389, I require that he roll a 4. And to simulate 0.01852, I ask that he roll either a 3 or a 4. Is it that simple?"

"Yeah, you've got the hang of it. But how do you start to figure out all the other possibilities?" Roscoe smiled that smile that bordered on a smirk. I started to crank up my "thinking on my feet" engine.

Generating More Probabilities

"Hmm," I said. "Let's add a column to your table. We can generate some new probabilities by considering more 'cumulative' outcomes." So I augmented Roscoe's table, as follows:

Total	Number of Ways	Probability	Probability of "N or Less"
3	1	0.00463	0.00463
4	3	0.01389	0.01852
5	6	0.02778	0.04630
6	10	0.04630	0.09259
7	15	0.06944	0.16204
8	21	0.09722	0.25926
9	25	0.11574	0.37500
10	27	0.12500	0.50000
11	27	0.12500	0.62500
12	25	0.11574	0.74074
13	21	0.09722	0.83796
14	15	0.06944	0.90741
15	10	0.04630	0.95370
16	6	0.02778	0.98148
17	3	0.01389	0.99537

18	1	0.00463	1.00000
Sum	216	1.00000	

"Well, you're on the right track," Roscoe offered. "Your fourth column definitely adds some new probabilities. For example, the probability of shooting either a 3 or a 4 is 0.00463 + 0.01389, which you have calculated to be 0.01852. You express this '3 or 4' as the probability of '4 or less.' By '5 or less' you mean that rolling a total of either 3, 4, or 5 defines a successful outcome.

"In fact, that's exactly how Monday and I proceeded. Problem is, you haven't gone nearly far enough."

"Before you jump ahead," I said, "let's look at how many probabilities we have so far. I'll shade the non-redundant ones in the table." My table now looked like this:

Total	Number of Ways	Probability	Probability of "N or Less"
3	1	0.00463	0.00463
4	3	0.01389	0.01852
5	6	0.02778	0.04630
6	10	0.04630	0.09259
7	15	0.06944	0.16204
8	21	0.09722	0.25926
9	25	0.11574	0.37500
10	27	0.12500	0.50000
11	27	0.12500	0.62500
12	25	0.11574	0.74074
13	21	0.09722	0.83796
14	15	0.06944	0.90741
15	10	0.04630	0.95370
16	6	0.02778	0.98148
17	3	0.01389	0.99537
18	1	0.00463	1.00000
Sum	216	1.00000	

"So I count 8 in the first column and 14 in the second column, for a total of 22. Is that what you get?" I asked.

"Not exactly," said Roscoe. "We have what might be called an 'accidental degeneracy.' Note that the probability of rolling a 6 is identical to the probability of rolling a '5 or less.' That's because there are 10 ways to make a 6, and (1 + 3 + 6 = 10) ways to make a 3 or 4 or 5, which is 5 or less. So I guess you have 21 distinct probabilities so far.

"But, as I said, you haven't gone far enough. There are lots more combinations."

Of Course, We've Already Left the World of Baseball

Things were getting a little hairy. "What did you guys do next?" I asked Roscoe.

"Well, a couple of things became apparent," replied Roscoe. "In fact, Monday sat me down for a chat, and what he said made a lot of sense.

"First thing he told me was that if we wanted to generate baseball probabilities, this was not the best way to proceed. He came up with a scheme whereby we could roll a single die several times and generate what we needed for baseball in about 3 rolls. I had to agree with him on that," Roscoe continued.

"But he also said that the problem of creating probabilities out of the total of 3 identical dice rolled simultaneously was interesting, in and of itself. He was now more interested in that than in the original problem. So we decided that we would concentrate our focus on seeing if we could figure that one out in all its generality.

"This sometimes happens in the real world, by the way," remarked Roscoe. "We start out trying to solve one problem, only to discover a new, more interesting, problem in the process. I think it is called serendipity, or something like that."

Reality Is Ugly

"The next thing we decided was that figuring out all the combinations was going to be laborious. We started making some tables but quickly gave up. Monday said he wanted to sleep on it a bit." Roscoe lit up a stogie, and I figured the rest of the tale would come out now.

"Wouldn't you know it, that Monday came back with the answer the next day," continued Roscoe.[4]

"Monday concluded that the first thing to do was to decide how many probabilities were possible as an upper limit. Since there were only 216 different ways to roll the dice, that had to be the maximum. So the best we could possibly do was to cover the interval from zero to one in 216 steps. From a granularity point of view, the optimum solution would be to have equal steps of 1/216, or 0.00463."

"Well," I said, "we know we can generate the first one!"

Roscoe grinned.

"Why was it important to try to determine the maximum possible number of probabilities?" I asked.

"Well," said Roscoe, "if you don't do that, you have a problem. Suppose you find that there are 87 distinct probabilities.[5] How do you know if you've got them all, and not missed some? Once you know the maximum possible, you can stop if you attain it. If not, you have to figure out why you couldn't get the others.

"Actually, all we have to do is figure out if we can do 108 possibilities up to a probability of 0.5. Then we can get the other 108 by taking the complementary solution," said Roscoe.

That made sense. This was a common trick in this area. If you know that rolling a 3 has a probability of 0.00463, then rolling anything but a 3 will have a probability of (1 - 0.00463), or 0.99537. So getting to a probability of 0.5 is always good enough.

Monday's Solution

"Monday started out systematically. First, he worked with the number of ways a total could be rolled, knowing that we can always convert to probabilities by dividing the number of ways by 216. That, it turns out, is easier to think about than decimal numbers."

"His first approach," continued Roscoe, "was to see if all of the first 9 'ways' could be constructed. Here is the little table he came up with:

Ways	Need to Roll
1	3
2	3 or 18
3	4
4	3 or 4
5	3 or 4 or 18
6	5
7	3 or 5
8	3 or 5 or 18
9	4 or 5

"I'm beginning to see," I said. "What Monday is going to try to do is see if he can cover the entire set of 'ways' up to 108, using the elements in the 'ways' column of 1, 3, 6, 10, 15, 21, 25, and 27. Very clever."

"Yes," replied Roscoe, "that's the idea. But remember, he can only use

each element twice. The symmetry of the problem is helpful, but notice that 'ways' get used up, so we need to be careful. As an example, suppose we used 3 and 18, and needed one more 'way.' We would be stuck. So there are constraints we have to watch out for."

"Wow," I said, "so the answer is still in doubt."

"Monday was equal to the task, it turns out," said Roscoe. "Here is the rest of his logic: To get 10 ways, you just use a roll of 6. By using 6 and its companion of 15, we have two '10s' to work with, so we can now get from 1 to 29. We can now add 1 to 29 to anything, provided we don't need any additional totals of 3, 4, 5, 6, 15, 16, 17, or 18. We have basically used up the 'ways' elements corresponding to 1, 3, 6, and 10."

"Well, 29 is still a long way from 108," I said.

Roscoe completed Monday's solution. "If you now use one of the '21s,' say 8, you extend the range from 29 to 50. And you still have both '15s', both '25s', and both '27s' to work with. Using the second 21 extends the range from 50 to 71. The pair of '27s' takes us to 125, well past the 108 we needed. So it is possible."

"What you are saying, then," I responded, "is that every way is possible, so that we can uniformly cover the interval in steps of 1/216. That is pretty amazing. Did you actually construct the table?"

"It was easy, once we knew it could be done," said Roscoe. "Here it is."

Ways	Probability	Roll Any of These Totals to Get this Probability					
1	0.00463	3					
2	0.00926	3	18				
3	0.01389	4					
4	0.01852	3	4				
5	0.02315	3	4	18			
6	0.02778	5					
7	0.03241	3	5				
8	0.03704	3	5	18			
9	0.04167	4	5				
10	0.04630	6					
11	0.05093	3	6				
12	0.05556	3	6	18			
13	0.06019	4	6				
14	0.06481	3	4	6			
15	0.06944	7					
16	0.07407	3	7				
17	0.07870	3	7	18			
18	0.08333	4	7				
19	0.08796	3	4	7			
20	0.09259	6	15				
21	0.09722	8					
22	0.10185	3	8				
23	0.10648	3	8	18			
24	0.11111	4	8				
25	0.11574	9					
26	0.12037	3	9				
27	0.12500	10					
28	0.12963	3	10				
29	0.13426	3	10	18			
30	0.13889	4	10				

27	0.12500	10					
28	0.12963	3	10				
29	0.13426	3	10	18			
30	0.13889	4	10				
31	0.14352	6	8				
32	0.14815	3	6	8			
33	0.15278	3	6	8	18		
34	0.15741	4	6	8			
35	0.16204	6	9				
36	0.16667	3	6	9			
37	0.17130	3	6	9	18		
38	0.17593	3	6	10			
39	0.18056	3	6	10	18		
40	0.18519	7	9				
41	0.18981	3	7	9			
42	0.19444	8	13				
43	0.19907	3	8	13			
44	0.20370	3	8	13	18		
45	0.20833	6	9	15			
46	0.21296	8	9				
47	0.21759	3	8	9			
48	0.22222	8	10				
49	0.22685	3	8	10			
50	0.23148	9	12				
51	0.23611	3	9	12			
52	0.24074	9	10				
53	0.24537	3	9	10			
54	0.25000	10	11				
55	0.25463	3	10	11			
56	0.25926	3	10	11	18		
57	0.26389	4	10	11			
58	0.26852	6	8	10			
59	0.27315	3	6	8	10		
60	0.27778	5	10	11			
61	0.28241	3	5	10	11		
62	0.28704	6	9	10			
63	0.29167	3	6	9	10		
64	0.29630	6	10	11			
65	0.30093	3	6	10	11		
66	0.30556	3	7	9	12		
67	0.31019	7	9	10			
68	0.31481	3	7	9	10		
69	0.31944	7	10	11			
70	0.32407	3	7	10	11		
71	0.32870	8	9	12			
72	0.33333	3	8	9	12		
73	0.33796	3	8	9	12	18	
74	0.34259	3	8	9	10		
75	0.34722	8	10	11			
76	0.35185	3	8	10	11		
77	0.35648	9	10	12			
78	0.36111	3	9	10	12		
79	0.36574	9	10	11			
80	0.37037	7	9	12	14		
81	0.37500	3	7	9	12	14	
82	0.37963	3	7	9	12	14	18
83	0.38426	3	4	9	10	11	
84	0.38889	7	10	11	14		
85	0.39352	3	7	10	11	14	
86	0.39815	3	7	10	11	14	18
87	0.40278	4	7	10	11	14	
88	0.40741	3	4	7	10	11	14
89	0.41204	6	9	10	11		
90	0.41667	3	6	9	10	11	
91	0.42130	3	6	9	10	11	18

88	0.40741	3	4	7	10	11	14
89	0.41204	6	9	10	11		
90	0.41667	3	6	9	10	11	
91	0.42130	3	6	9	10	11	18
92	0.42593	8	9	12	13		
93	0.43056	3	8	9	12	13	
94	0.43519	3	8	9	12	13	18
95	0.43981	4	8	9	12	13	
96	0.44444	8	10	11	13		
97	0.44907	3	8	10	11	13	
98	0.45370	3	8	10	11	13	18
99	0.45833	4	8	10	11	13	
100	0.46296	6	7	9	12	14	15
101	0.46759	3	8	9	10	11	
102	0.47222	4	8	10	11	13	17
103	0.47685	3	5	8	10	11	13
104	0.48148	9	10	11	12		
105	0.48611	3	9	10	11	12	
106	0.49074	3	9	10	11	12	18
107	0.49537	4	9	10	11	12	
108	0.50000	3	4	9	10	11	12

"Note," said Roscoe, "that for some of these the answer is not unique. But remember also that it doesn't have to be. We just need to find one set of totals that gives us the right number of 'ways' to get there."

Lessons Learned

Roscoe seemed less than totally happy at the end of the story. "What's bugging you, Roscoe?" I ventured.

"Well, first of all," he replied, "I got suckered once again. I thought I had a baby problem on my hands, and it turned out to be much more complex than I first thought. That's always annoying.

"Second, one of my time-honored techniques washed out on me," he continued. "Usually when a problem starts to grow teeth, I revert to a simpler instance of the problem to gain insight. But in this case, the simpler case of the sum of two dice was totally useless.

"Third, Monday's solution is convincing, especially when you actually have the table in hand. But even there, it seems like he used some heuristics instead of a logical proof. Although I am the last guy in the world to criticize anyone for getting the answer any way he can. Never let it be said that I was an advocate of excess purity, let alone elegance."

Now, I take a much more clement approach. Roscoe, and especially his buddy Monday, had done a great job using pencil, paper, slide rule, and common sense. In fact, had Roscoe lost his slide rule in the shipwreck, Monday still could have solved the problem, as the division by 216 is a purely cosmetic event -- we can state probabilities as 67/216 if we have to.[6] That the problem can be solved on a desert island, with no computers, no Microsoft Excel, and no pivot tables, by using just time, energy, and intellectual curiosity, is wonderful. It also means that Blaise Pascal could have solved the problem in the seventeenth century; he had all the tools he needed back then. And he didn't lack for intelligence or intellectual curiosity, either.

Roscoe and Monday wound up being able to extract a uniform probability distribution to about half a percent by simply rolling three dice and looking at the total, so long as they used the algorithm of specifying the probability first and then deciding whether the roll was successful or not. In the original baseball example, it meant they could get batting averages to within 5 points.[7]

"And here's the last laugh," Roscoe concluded. "That Monday showed he really understood the problem with the following observation. If you do the same experiment with *four* dice, you can prove that it is impossible to cover the interval from zero to one uniformly, as you could with three. That is, you can generate many more probabilities, but you cannot have them spread out equally. Now *that* is an interesting result, and it is true not only for four dice, but for any number of dice greater than three. Put that in your pipe and smoke it!"

I encourage readers to see if they can rediscover Monday's impossibility proof for four or more dice.

Notes

1 We note here that statistics such as official at-bats and walks (from which plate appearances can be calculated) are the kinds of numbers that appear on the baseball cards mentioned above in the salvage inventory.

2 Another example of a "corner case" would be the probability, in the case of an out, that the batter hits into a double play if there is a man on first base. There are actually people who construct these kinds of fantasy simulations for a living; as baseball mavens record almost every kind of statistic imaginable, this is a natural result. It should also be noted that our "scheme" for simulating the batter's plate appearance is certainly not unique; there are many ways to achieve the same result. For example, you could recompute the batting average based on plate appearances, and then compute walks, singles, doubles, triples, and home runs from renormalized statistics. There are many, many alternative formulations, of which we have picked just one as an example. However, regardless of the formulation, one of the big challenges before the widespread availability of computers was figuring out how to do the probability simulation using simple, low-cost devices available to most people.

3 Note that the probabilities have, in some cases, 5 significant figures. On his desert island, Roscoe and his slide rule could do 3 at best. 0.00463 represents 3 significant figures; the others should be appropriately rounded. The display is less jarring this way, but we should not attribute any significance to figures smaller than thousandths.

4 Some of you have come to the conclusion that Roscoe is an avatar for your humble author. If that is true, then Monday is an avatar for my son David, who figured out the solution to this problem. I may be good at discovering interesting problems, but David is even better at solving them. And the name of the avatar is Monday, because that is the day we come

to work.

[5] As Roscoe himself did in an early attempt to solve this problem.

[6] Not to mention the obvious: Roscoe could still do "long division" by hand if he had to.

[7] Note that averages of .400 or greater for a single season have not been achieved in more than 60 years. The recently deceased Ted Williams hit .406 in 1941, and is the last player to have batted over .400 for a season.

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

Copyright Rational Software 2002 | Privacy/Legal Information

Copyright Rational Software 2002

V2002 Rational TestManager Integrations

Rational Software Whitepaper

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

Introduction

This technical paper is designed to familiarize Rational customers with the integration between Rational TestManager and other Rational tools. It is assumed that the reader has a basic knowledge of Rational Suite components and the core testing products, TestManager and Robot. It should be noted that TestManager also provides an open API for integrations with non-Rational tools. Those integrations provide the ability to link custom artifacts to the test plan and to execute test scripts built with other languages and tools. Those API-level integrations are not discussed in depth in this technical paper.

The primary testing tool that provides integration to other Rational tools is TestManager. TestManager is a product that is available in every Rational Suite as well as the point products, TeamTest and Robot. TestManager is also offered as a stand-alone product. TestManager is used to plan, execute and analyze the results of tests.

The areas of integration between TestManager and other Rational tools that are discussed in this paper are depicted in the diagram below.

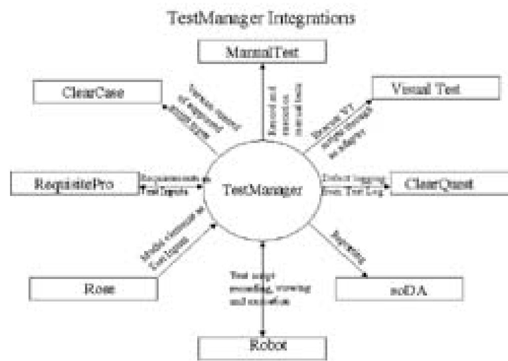


Figure 1: Integration between TestManager and other Rational Tools

(click here to enlarge)

Section I - Understanding Test Inputs

Testers gather a variety of types of information that they will use to determine what needs to be tested, or in other words, what our test cases are meant to cover. Any source of information that the testing team uses to help define and design test cases is called an "input". TestManager provides built-in interfaces for inputs from RequisitePro, Rose and Excel spreadsheets. This provides the testing team an interface to organize the testing effort based on inputs to the test plan. All test inputs are displayed in TestManager through the use of an adapter. An adapter is a mechanism to connect external sources of data to the test plan. In addition to the adapters provided for the three sources listed above, Rational a documented API to create adapters for other input types.

Section II - TestManager - RequisitePro Integrations

RequisitePro is a tool that allows the project team to capture and track all of the requirements for a project. RequisitePro combines the power of using Word to write documents containing requirements, along with a database for sorting, filtering and prioritizing requirements, as well as tracking impact of requirement changes. For the testing team, access to the latest requirements is critical to ensure complete and thorough testing. Testers need to ensure that they are building test cases to validate each requirement (or use case) stored in RequisitePro. In addition, testers need to know when requirements change so that they can understand the impact of the changed requirement on the associated test case(s).

Making Requirements Accessible to Testers

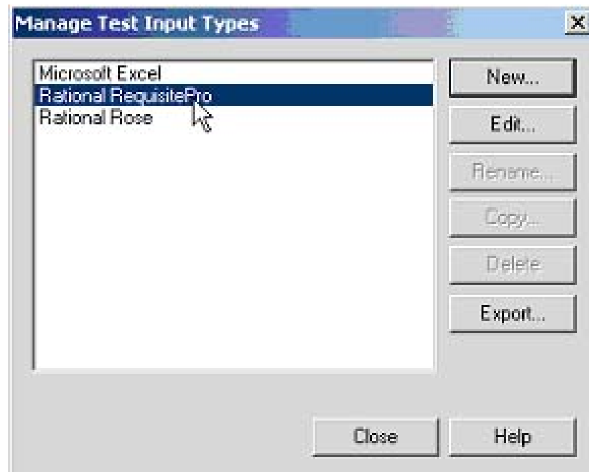
Requirements are one of the most common inputs to the test planning effort. TestManager provides an interface to automatically display requirements stored in RequisitePro as "inputs" to the test plan. This allows you to see any requirement from RequisitePro. You can then set a filter, to see a smaller list of requirements, filtered by the requirement

type.

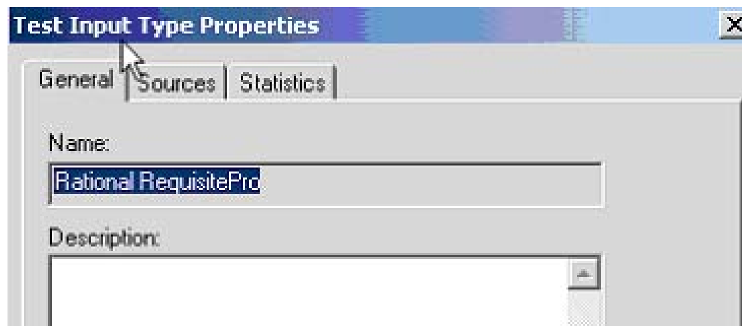
Establishing a RequisitePro Test Input Source

To establish requirements from RequisitePro as an input to a testing project requires only a few simple steps. Assuming the RequisitePro project has been created, follow the steps below to define that project as a test input:

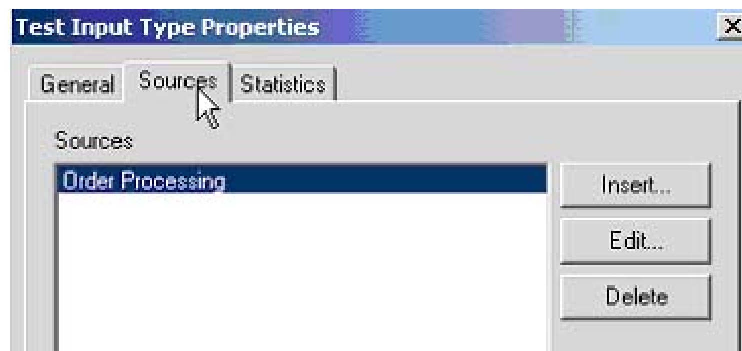
1. From TestManager select **Tools** > **Manage** > **Test Input Types**.



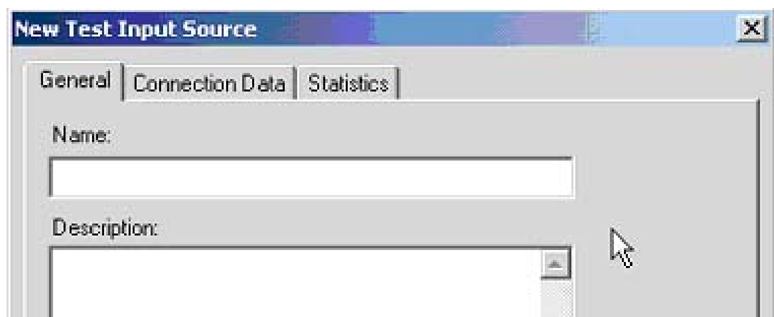
2. Select the type **Rational RequisitePro** and click **Edit**.



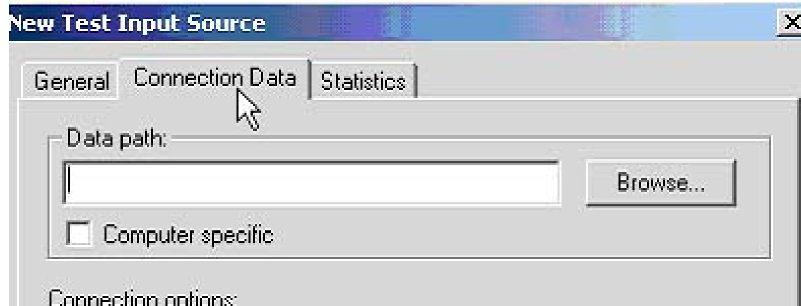
3. Click on the **Sources** Tab. Each RequisitePro project can be a test input source.



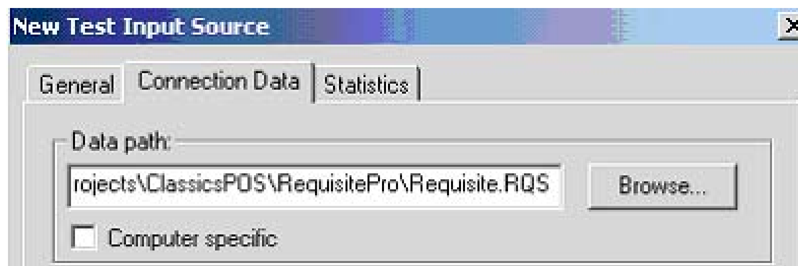
4. If you have existing RequisitePro projects already defined as test input sources, they will display here. To add a new source click on **Insert**.



5. Enter a name for this source then click on **Connection Data**.



6. Click **Browse** and point to the RequisitePro project's .RQS file. Be sure to use UNC naming when working with a source that is not computer specific.



7. Click **OK** to save this new source. It will appear in the list of sources.
8. Close the remaining dialog boxes to return to TestManager.
9. From TestManager select **View** > **Test Inputs**.

   You should now see your RequisitePro project as a new Test Input Source. If you expand the source, you will see all the requirements in that project. By default, all requirements are shown. To filter the requirements to a smaller subset, right click on the Source name and select **Set Filter**. Here you can select the type of requirements that you want to show as inputs. The filters that you set are user specific - not source specific.

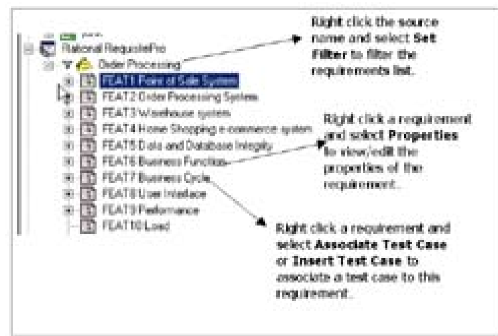


Figure 2: Common Actions on Requirements as Inputs

(click here to enlarge)

Repeat the above steps for each additional RequisitePro source that you want to establish.

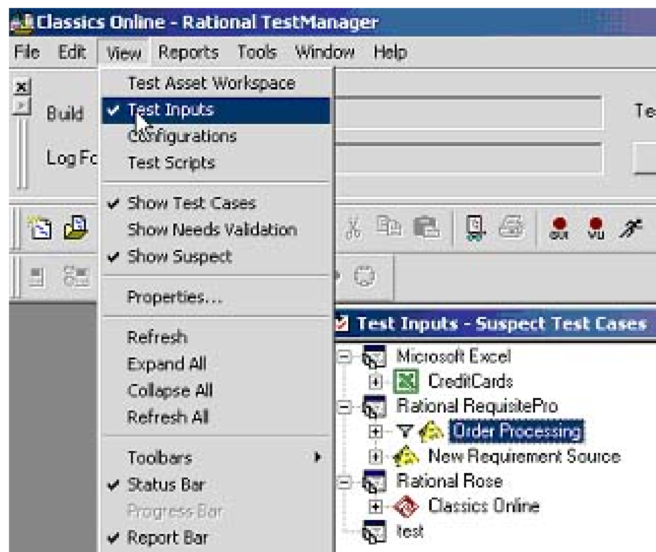
Notes on Using the TestManager-RequisitePro Integration

- If a RequisitePro project has been established as part of a Rational project defined through the Rational Administrator, that RequisitePro project will automatically be defined as a test input source within TestManager.
- Once a source has been established and you begin to associate test cases with requirements, TestManager will keep track of each association.
- Deleting the source will disconnect all links between test cases and the requirements.
- A test case can be associated with more than one requirement as well as more than one input type. So, for example, a test case could be associated with a requirement from RequisitePro and record from an Excel spreadsheet.

Viewing Suspect Test Cases (Impact Analysis)

After RequisitePro requirements have been established as a test input source, you can take advantage of TestManager's ability to track changes to requirements and report on the test cases affected by the change. This is often called Impact Analysis or Suspicious Test Case tracking. Impact Analysis means determining what requirements have changed (since a test case was originally planned for a requirement) and what impact that change has on testing. To use this feature follow the steps below:

1. Select **View - Test Inputs** to open the Test Input View in TestManager.



2. Ensure that **Show Suspect** menu option (shown above) is checked. Once Show Suspect is turned on, you will be able to see suspect test cases in the Test Input view. Suspect test cases will always display in bold in the Test Plan view. In addition, you can run the **Test Plan Suspicion Coverage with Status** report to obtain details on suspicious test cases. Below are examples of the way suspect test cases are marked in each of these two views.

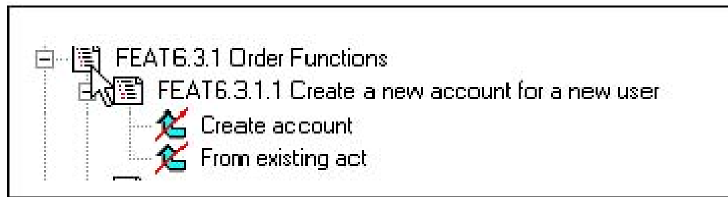


Figure 3: Slash indicates Suspect Test Cases in Test Inputs View

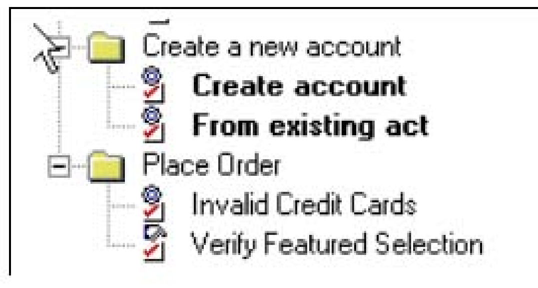
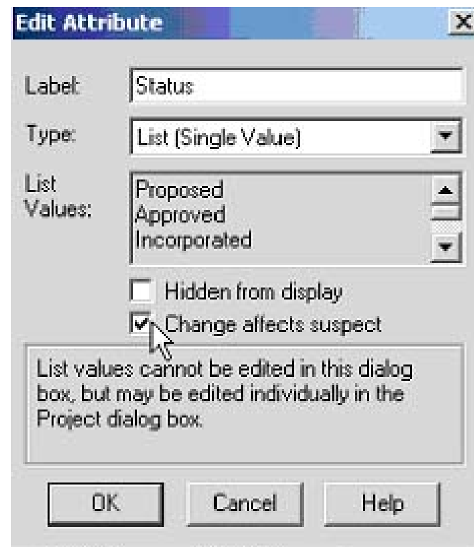


Figure 4: Bold indicates Suspect Test Cases in Test Plan View

Test Cases become suspicious when the requirement they are associated with is changed in some meaningful way. Any change to the text or name of a requirement will automatically trigger suspicion. In addition, you can set a change to an attribute's value to cause suspicion. To establish an attribute value change as a suspicion trigger follow the steps below:

1. Open the RequisitePro project.
2. Select File > Project Administration > Properties.
3. Select the requirement type that defines the attribute, that when changed, should trigger suspicion.
4. Select the Attribute label then click Edit.



5. Click on Change affects suspect. Note that setting this option affects suspicion tracking in both RequisitePro and TestManager.

Notes on Suspect Test Case Tracking

- A test case is marked as suspicious based on a change that has occurred since the test case was associated with the requirement or since the last time the requirement was deemed to have no changes affecting suspicion. This is done by date/time stamping the original connection of the requirement and the test case and monitoring any changes that occurred after that date/time. (The date/time marker is in minutes, not seconds. So, a refresh won't pick up a change that has occurred within one minute.) If the change is one that affects suspicion, that will result in a suspect test case in TestManager.
- If a series of changes occur to a requirement, all of which affect suspicion, the user will need to view the requirement's change description and revision history to find all the changes. It should not be assumed that the latest change listed in the change history is the ONLY change that affected the requirement.
- In order for the user to see suspect test cases, the Test Inputs View in TestManager must be reopened. Any changes made to requirements while the Inputs View is open will not be marked suspect until a refresh of the input source occurs.
- When Microsoft Access is used as the database of a RequisitePro project, a delay (of a few minutes) may be seen when updating the suspicion status. This is due to an MS Access latency issue.

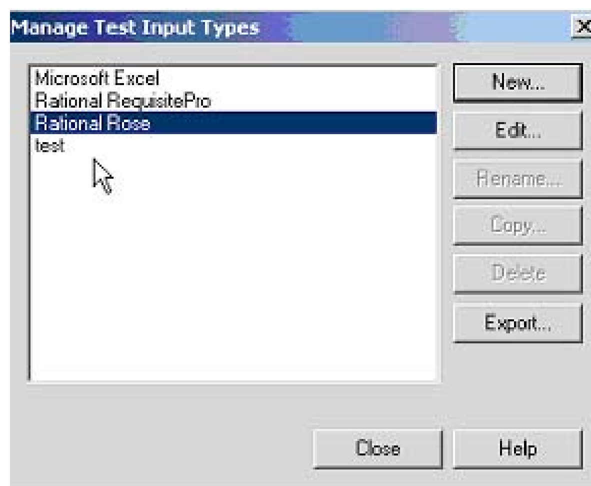
Section III - TestManager - Rational Rose Integrations

Making a Rose Model Accessible to Testers

If the development team on your project is using Rational Rose, you may wish to use Rose model elements and/or use cases as inputs to your test plan. By doing so, you will be able to plan and track testing for use cases and elements of your application. A Rose input adapter is provided to allow you to easily establish Rose elements as inputs to testing. To establish an input source for Rose, follow the instructions below.

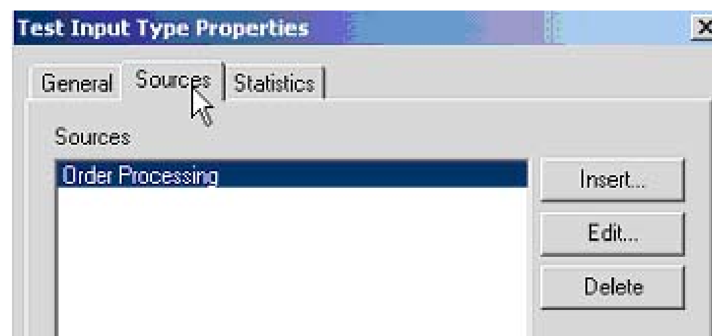
Establishing A Rose Model Test Input Source

1. From TestManager select **Tools** > **Manage** > **Test Input Types**.



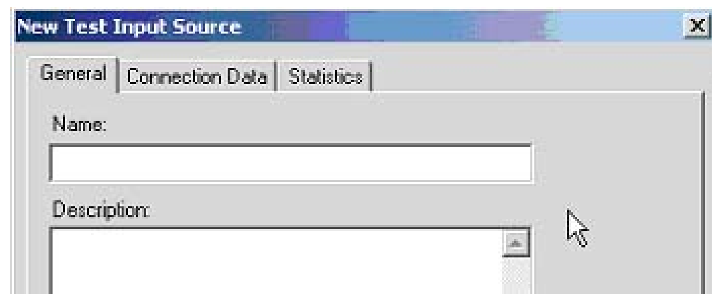
2. Select the type **Rational Rose** and click **Edit**.

3. Click on the **Sources** Tab.

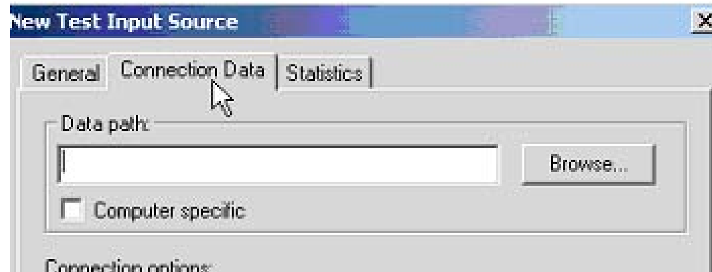


4. If you have existing Rose models as inputs they will display here. To

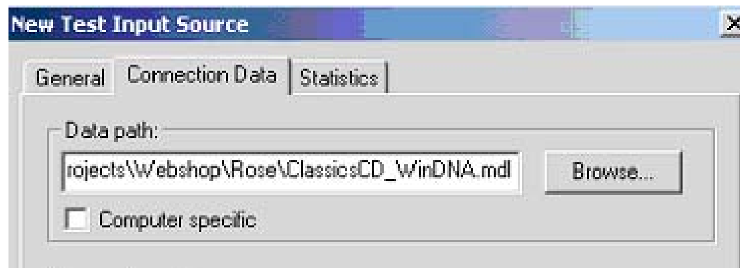
add a new source click **Insert**.



5. Enter a name for this source then click on **Connection Data**.



6. Click **Browse** and point to the Rose model's .mdl file. Be sure to use UNC Naming to ensure all team members will have access to this source.



7. Click **OK** to save this new source. It will appear in the list of sources.

8. Close the remaining dialog boxes and return to TestManager

9. From TestManager select **View** > **Test Inputs**. You should now see your Rose model as a new Test Input. If you expand the source, you will see all the elements in that model. By default, all elements are shown. To filter the model to a smaller list of elements, right click on the Source name and select **Set Filter**. Here you can select the type of elements that you want to show as inputs.

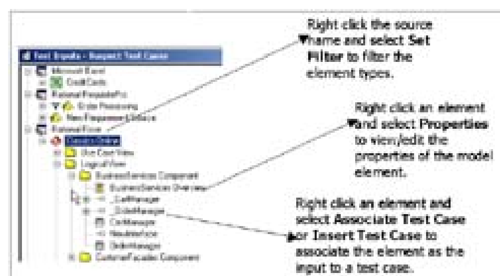


Figure 5: Common actions on
Rose Model Inputs

(click here to enlarge)

Notes on TestManager - Rose Integration

- Unlike RequisitePro projects, a Rose model that is defined as part of a project through the Rational Administrator will not automatically be defined as a Test Input.
- Changes to Rose model elements are not tracked for suspicious test case reporting because individual changes to model elements are not tracked.

Section IV - TestManager - Microsoft Excel Integrations

Making Data in Excel Accessible to Testers

Many testing organizations store test planning information in Excel spreadsheets. Some examples of the kind of information you might have in Excel are, requirements (if your team is not using RequisitePro) or descriptions of tests that need to be executed. If you have information stored in this format, you can use TestManager's built-in Excel Input adapter to allow your test team to see the Excel data through TestManager. In addition to being able to view the Excel data in TestManager, the testing team can associate test cases with each individual record in the spreadsheet and track suspect test cases based on changes in the spreadsheet's data. Below is a figure that depicts the integration between Microsoft Excel and TestManager.

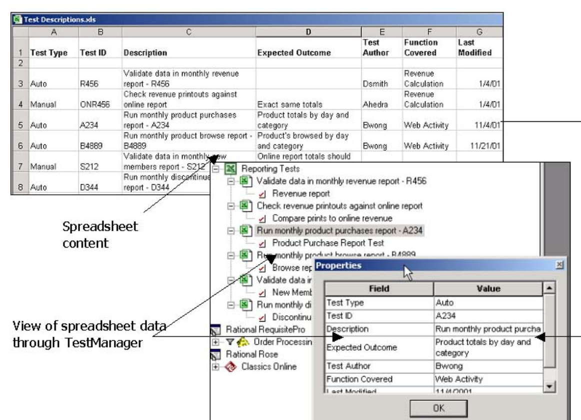
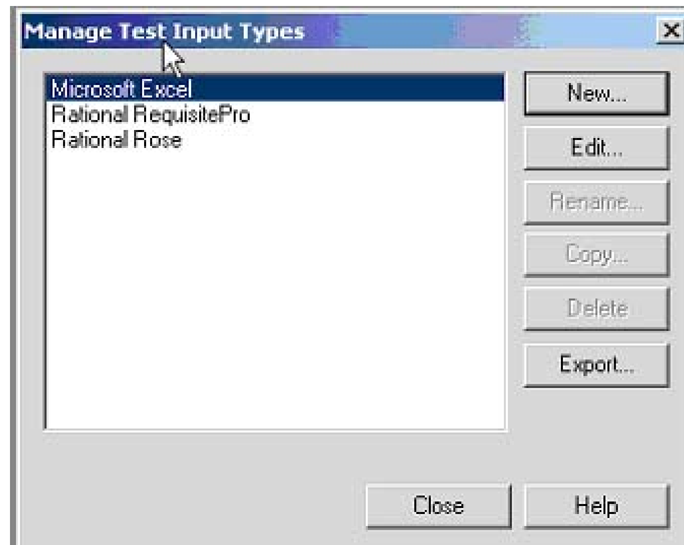


Figure 6: Excel - TestManager
Integration

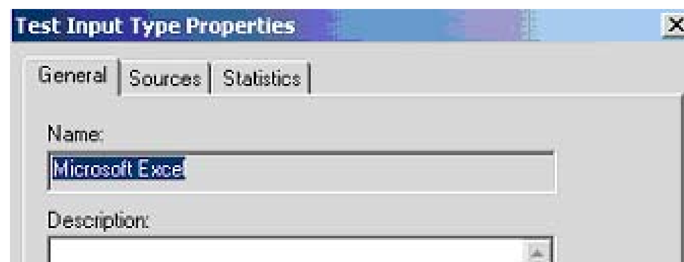
(click here to enlarge)

Establishing an Excel Test Input Source

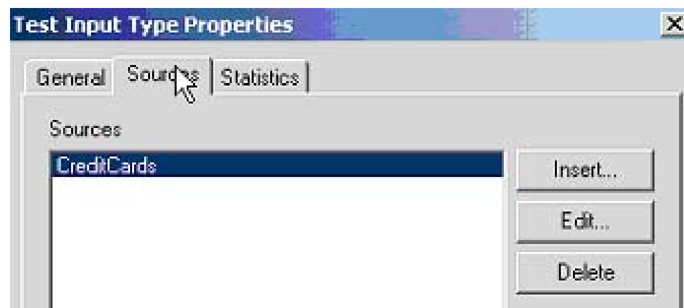
1. From TestManager select **Tools** > **Manage** > **Test Input Types**.



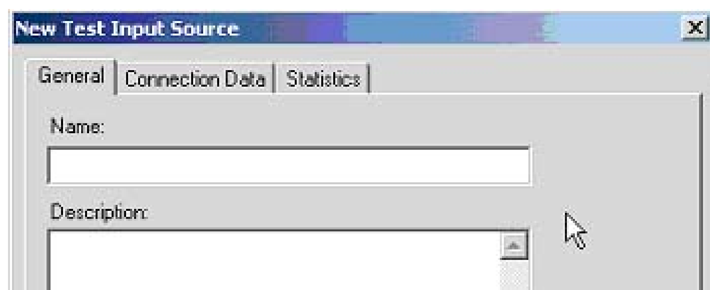
2. Select the type **Microsoft Excel** and click **Edit**.



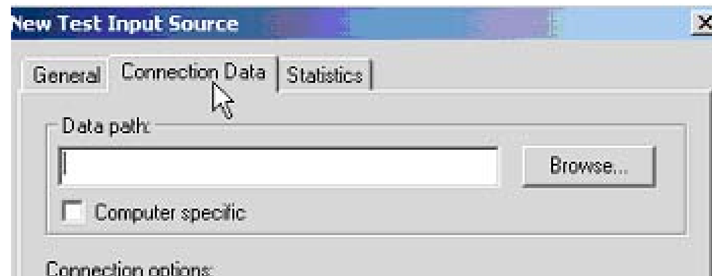
3. Click on the **Sources** Tab.



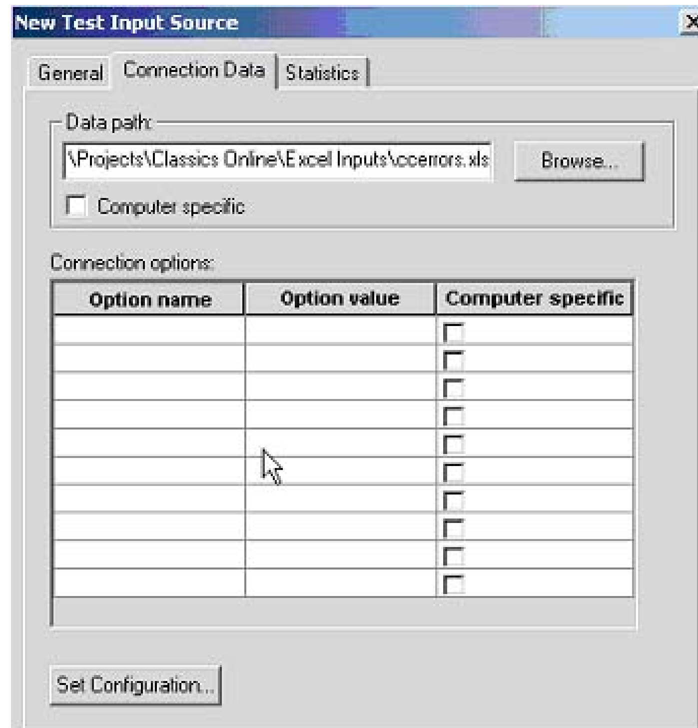
4. If you have existing Excel spreadsheets as inputs they will display here. To add a new source click **Insert**.



5. Enter a name for this source then click on **Connection Data**.



6. Click **Browse** and point to the .xls file. Use UNC naming to ensure that the file is accessible by all team members.



7. Click on **Set Configuration**. Here you must specify the details of the spreadsheet. Below is an example of how this configuration data would match a spreadsheet.

Figure 7: Example Excel Configuration

(click here to enlarge)

- Orientation
  - Select horizontal if a record is a row. Select vertical if a record is a column.
- Row/column containing headers
  - Enter the row number or column letter containing the heading
- Data area
  - Enter the range of cells that contains data that should be available through TestManager. Exclude the row/column with the header as well as other extraneous data that is not

needed for test planning purposes. If the data in the Excel spreadsheet expands, this data area setting must be updated in the adapter settings to make the new data available in TestManager.

- Unique ID
  - Enter the relative column or row number that contains data to be used to uniquely identify a record. This field is *required* and is always a numeric entry relative to the data area.
- Input name
  - Enter the relative column or row number that contains data to be used displayed in the Test Inputs View. This field is *required* and is always a numeric entry relative to the data area.
- Description
  - Enter the relative column or row number that contains data that contains a description of each field.
- Last Modified
  - If you want to be able to track suspect test cases in the event that data in the spreadsheet is changed, you can designate a field that will contain a date. This field must be in a date format. This allows you to track changes at the individual record level, BUT you must manually update this date field whenever you change the content of a record. Updating this field will trigger suspicion for test cases associated with this record.
- Use date/time of file
  - Another option for tracking suspicious test cases is to use any change to the file that causes a change to the date/time of the file as a trigger. If selected, this will automatically trigger suspicion, but all test cases associated with any of the records in the spreadsheet will be marked suspect.

8. Click OK to save this new source. It will appear in the list of sources.
9. Close the remaining dialog boxes and return to TestManager.
10. From TestManager select **View** > **Test Inputs**. You should now see your Excel spreadsheet as a new Test Input. If you expand the source, you will see all the elements in that model. By righting clicking on any items in the source you will receive a properties window that contains all the data from the selected record.

Notes on TestManager - Excel Integration

- Excel must be installed on the system where the Test Input is displaying.

- The date/time stamp used to track changes to an Excel spreadsheet is rounded to minutes - not seconds.
- Supported Excel Versions are version 8 and 9 in Office 97 and Office 2000.

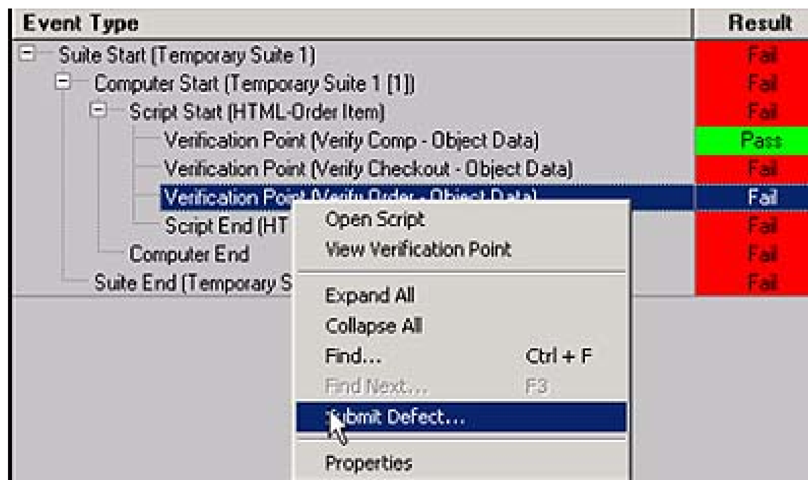
Section V - TestManager - ClearQuest Integrations

For customers using both TestManager and ClearQuest, TestManager provides a built-in integration that allows testers to generate a ClearQuest defect directly from the test log. This allows testers to immediately generate a defect upon detection of a discrepancy reported in the test log.

You can use any ClearQuest schema for the integration, but the TestManager packages must be applied.

Generating a Defect from the Test Log

1. Open the test log in TestManager.
2. Click the **Details** tab.
3. Expand the log so that all events are shown.



4. Right click on an event (line) in the log then select **Submit Defect**.

Depending on the line in the log that was selected when "Submit Defect" was requested, different information will automatically be populated in the defect. Below is a description of the information that is automatically populated and the conditions under which each data item will be included.

- Suite Project
  - Always populated
- Build
  - Always populated
  - Based on the build label selected when the test was executed
- Log Folder

    - Always populated
    - Based on the folder selected when the test was executed

- Log
    - Always populated
    - Based on the log selected when the test was executed

- Test Case
    - Populated only if a test case was executed and the defect was generated from an event in the log resulting from the execution of a test case.

- Test Script
    - Always populated if the defect was generated from an event in the log resulting from the execution of a test script.

- Verification Point
    - Only populated if the tester submitted the defect from a verification point event in the log.

- Test Input
    - Only populated if a test case was executed and the test case has an associated test input(s). This field can contain a list of values.

Notes on TestLog-ClearQuest Integration

- The information that is transferred from the log to the defect is hard coded. You cannot alter the information in this integration.
- Once the defect is saved and the tester is returned to the log, the defect ID is automatically populated in the log field titled "Defects".
- If the tester attempts to generate a second defect from this exact line in the log, he will be prompted as to whether he would like to open the existing defect or generate a new one.
- A tester may submit a defect from any line in the log, regardless of whether the results are listed as a pass or fail.

Section VI - TestManager - Robot Integrations

Rational Robot is the tool used to record GUI test scripts for functional testing and Sessions for performance testing. TestManager is tightly integrated with Rational Robot in a number of areas. Each area is described below.

Recording Scripts from TestManager

From TestManager, you can start recording either a GUI script or a session. This integration is available from the TestManager menu selection **File > Record**. After the type of recording is selected, Robot will

automatically be started and the recording process will begin.

Viewing/Opening Scripts from TestManager

TestManager provides a view that allows you to view all scripts (including Robot scripts). To access that view, select **View** > **Test Scripts** from TestManager.

TestManager provides two mechanisms to open test scripts. To open a test script from TestManager, select **File** > **Open Test Script**, then, select the type of script to open or, from the Test Scripts View, right click the script and select **Open**.

Executing Scripts from TestManager

TestManager is the central script execution engine for all types of test scripts, including those built by Robot. There are three primary methods to execute Robot generated scripts from TestManager. They are: 1) execute a suite 2) execute a test case 3) execute a test script. All of these execution methods are available from TestManager. When a suite is executed, all test cases, configured test cases with a configuration that matches the workstation scheduled to run the test, and test scripts that are in the suite will be executed. If a test case is executed the script listed as the automated implementation will execute. If no automated implementation is listed it will execute the manual implementation. Regardless of which method you use to execute, the GUI Playback Options that are set in Robot will be used during playback of Robot GUI scripts from TestManager.

If any of the diagnostics tools, Purify, PureCoverage or Quantify are selected as a Robot GUI Playback option, TestManager will start the tool at the beginning of the execution and the results will automatically be logged into the results log file.

Section VII - TestManager - ClearCase Integrations

ClearCase is Rational's software configuration management system. There are two variations of ClearCase available, ClearCaseLT, included in every Rational Suite product and designed for small project workgroups, and full ClearCase, that contains more robust capabilities.

UCM (Unified Change Management) defines a process for managing and tracking changes to artifacts (e.g. source code, defects, scripts, requirements) that are versioned in ClearCase.

In Rational Suite, baselining of artifacts is supported for the management of all Rational Test Assets. Baseline support will enable test practitioners to work with versions of an entire test project throughout a product's lifecycle. It should be clear that the reader understands that this baselining support DOES NOT include the ability to version control individual test assets (such as version controlling and maintaining variations of a single test script). Rather, baselining supports versioning the entire test datastore's contents, essentially, taking a snapshot in time

of the entire test datastore.

Included in the support for test assets are:

- The ability to baseline Test Scripts, Test Verification Points, Test Schedules, Test Datatypes, Test Datapools and Test Documents
- The ability to create new test projects from existing UCM project baselines.
- Ability to restore a project from an existing UCM project baseline.

The key benefit of the baseline capabilities of the testing products is the ability to create a new Test project from an existing UCM baseline. It is common to want to create a new project from an established baseline of an existing project and included data from that baseline.

The primary use cases follow below:

1. The need to split teams that had shared the same test assets. Suppose a major milestone was met in this instance First Customer Ship. Half of the team may move on to the next major release while the other half of the team needs to work on a patch for the just released product. In this instance, both teams will be able to utilize all the existing test assets without compromising the integrity of the assets by creating a new test project from an existing UCM baseline.
2. The ability to back out changes en masse and return to a known state. Suppose the test team created or changed a large number of test assets since the last baseline. If the changes made were determined to be more harmful than beneficial. The team would have the ability to roll back to a known good baseline where the Test assets weren't compromised.

Rational Test Tool users will not be aware that they are working on versioned elements aside from the appearance of Check In and Check Out dialog boxes when test assets are added. Rather, they will still access test assets from a shared directory and work out of the configured relational database. All of the versioning work occurs only during two administrator-initiated operations; those being project creation and check in all. Both of these operations are performed from the Rational Administrator application. (Note; the Rational Administrator is installed with any Rational Suite and Rational Test tool point product install.)

The detailed steps for setting up use of UCM with Rational test artifacts are described in the Rational Administrators Guide.

Section VIII - TestManager - Visual Test Integrations

VisualTest is an automated functional testing tool for testing professionals whose team is creating Windows applications for Web or e-business.

VisualTest requires manual programming and is intended for more technical users. VisualTest is integrated with Microsoft Visual Studio, a desktop development environment, and has extensive integration with Microsoft Visual C++.

You can execute Visual Test scripts from TestManager using the Visual Test Execution Adapter. This adapter executes Visual Test suites (.vts). The results of the execution, a standard text log file, are imported into TestManager and are displayed in the log-viewer. To obtain the execution adapter for Visual Test scripts, contact your local Rational representative or download it from Rational Developer Network, located in "Development Resources - Rational Tools - TestStudio - Downloads". (It is provided at no charge)

After obtaining this .dll, you will need to create a Test Script Execution adapter for Visual Test scripts. Follow the instructions in the TestManager User Guide for steps to create an execution adapter.

Section IX - TestManager - soDA Integrations

Rational SoDA® is a project reporting tool that generates complete project reports by gathering data from a number of Rational tool databases and reporting it in a single document. SoDA is included in every Rational Suite product. TestManager provides a link to three pre-built soDA templates for reporting against test assets. The templates are available from the TestManager menu option **Report > soDA Reports...**

You may also create your own templates by using soDA. The online soDA documentation provides details about the classes of information stored in the test data store that you can report against. A list of those classes is below:

Figure 8: Test Artifact Classes Available in SoDA

Build	Session
Computer	Suite
ConfiguredTestCase	TestCare
Group	TestCaseFolder
Iteration	TestCaseResult
Log	TestInput
LogEvent	TestPlan
LogFolder	UseCase
Project	User
Requirement	Variant
Script	VerificationPoint

User-Submitted
We'd like the chance to publish your work. Click for details.

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

Copyright Rational Software 2002 | Privacy/Legal Information