COVER STORY

The Seeds of Rational Thought
- an interview with Dave Bernstein

october 2001
Editor's Notes

Twenty years ago, Rational was a small, startup computing company founded on a big idea: to create tools that would make it easier for software developers to use modern software engineering practices. Relatively few programmers back then thought of system "design" as a first step, for instance. Rational set out to change that, believing that a better software development environment would, ultimately, reduce the chaos associated with writing, deploying, and maintaining code.

After two decades of slow and steady growth, Rational Software is still committed to its original mission. And many of the company's earliest employees are still here, too. For this month's cover story we interviewed one of them: Dave Bernstein, Rational's Senior Vice President and General Manager of Products. He gives us a look at some of the early challenges and breakthroughs that led to the "R1000" and the first iteration of "the Rational Environment." (If these terms are a bit unfamiliar, the interview with Dave will explain.)

In a sense, this month's Features section presents a compact view of Rational Software's past, present, and future. After reading about Rational's early history, you can turn to Philippe Kruchten's discussion of the current challenges inherent in software "engineering": What makes this discipline unique from an engineering perspective? (Hint: Consider the "soft" in "software.") And Paul Dreyfus provides an overview of the new Rational Developer Network, which you'll certainly be hearing more about in the coming months. This online resource center is one of the latest tools we're offering to deliver more information into your hands faster and to help you connect with peers in testing, modeling, requirements and configuration management, and other aspects of the software lifecycle. For a taste of what lies in store for new members, we've given you two sample articles from Rational Developer Network, too.

In our other sections, Mike Taylor offers a very creative approach to project tracking, Ben Lieberman is back with a technique for capturing user navigation requirements, and Peter Eeles explores strategies for system layering.
If you think that software development is simply about programming, then you should check out the book review and sample chapter from Murray Cantor's new title, *Software Leadership: A Guide to Successful Software Development*. And finally, Joe Marasco offers fresh insight into the role and practice of politics in software development organizations.

Welcome to our eleventh monthly issue of *The Rational Edge*.

Mike Perrow
Editor-in-Chief
The Seeds of Rational Thought

an interview with
Dave Bernstein
Senior Vice President, Products
Rational Software

Have you heard? Rational Software turns twenty this year! Of course, Rational is proud to have been in business that long. With two decades under its belt, Rational has survived, adapted, and pioneered changes in the computing industry that would make Jules Verne's head spin. How many of you, reading this, remember what the computing industry was like in 1981? That was before shrink-wrapped software, before the mass market for PCs and peripherals, before any widespread use of GUIs and event-driven programming, a long time before Microsoft Windows, and a long, long time before what we know today as the Internet.

"But from the very beginning," notes Dave Bernstein, Rational's senior vice president of products, "our software development technology was created to help engineers design and deliver sophisticated software systems. Rational provided an environment that automated many of the processes that were critical to a development project's success -- including change management, iterative development, and a modular approach to structuring applications." Bernstein began his tenure with Rational Software in the winter of 1982, nine months after the company was founded. He came to Rational by way of Data General, where he designed computer hardware and applied skills learned at MIT. His mission at Rational? To help the new company "build tools that encourage the use of modern software engineering practices."

Although the tools Rational builds today have changed dramatically with the needs of the industry, Rational's own mission -- ensuring the success of customers who depend on developing software -- has remained essentially unchanged since Bernstein joined the company. In the following interview with The Rational Edge, Bernstein traces Rational's
earliest objectives, its core values, and its founders' commitment to improving the processes associated with software engineering.

**TRE:** What were some of the founding principles on which Rational was built?

**DB:** We started with a coherent vision for changing the way software would be developed. Our two founders, Paul Levy and Mike Devlin, had graduated from the United States Air Force Academy. Mike was involved in Ada programming as part of his work in the Air Force, and both Mike and Paul had supervised significant national defense programs. They had personally witnessed the difficulties that arise when software is developed without sound engineering practices. They discovered that these difficulties were created not by an unwillingness to apply software engineering techniques, but rather by a lack of commercial tools that supported such practices. They reached the conclusion that a better development environment was needed to facilitate dramatic improvement.

In those days, few people built software by designing it first. The software developer's primary tools were a text editor, a compiler, and a debugger. Most programmers simply wrote code with an editor, compiled it, and with the help of a debugger tried to make it work. Paul and Mike saw an opportunity to build a big business around improved methods for developing software. We referred to these improved methods as "modern software engineering practices." Those practices included organizing software into well-defined modules ("modularity"), and minimizing the complexity of each module's external interface ("abstraction") -- techniques that are basic to what is now called component-based development. Following these methods results in software that is easier to maintain in the face of change, and enables teams of developers to work without interfering with each other. To be effective, both modularity and abstraction require programming language support.

We also identified iterative development as an important element of modern software engineering practice. This approach recognizes the likelihood of major mistakes -- particularly design errors -- being made while building a complex application. Given this likelihood, it makes sense to seek out such problems early and resolve them, rather than complete the design, build the system, and then discover in the course of testing that the application is fundamentally flawed. By making design problems visible early on, an iterative approach can prevent the need for vast amounts of rework that would otherwise consume both schedule and budget.

In our view, the primary impediment to using modern software engineering practice was that the standard development tools -- text editors, compilers, and debuggers -- did not provide the required support. So Rational's founding vision was: "Let's build a development environment that encourages people to use modern software engineering practices."
**TRE:** How did Ada come into the picture?

**DB:** We needed a programming language that supported the construction of modular programs with abstract interfaces. We also wanted a declarative language -- one that could aid in the detection of programmer errors by capturing the meaning of a program as well as its logical flow. In the early 80s, most of the languages with these characteristics were research or teaching languages like CLU or Modula. Pascal met most of our requirements, but lacked the rigorous definition that would have enabled multiple vendors to create compatible, platform-independent compilers. Only Ada, developed specifically to support industrial-strength use of modern software engineering practices, met all of our requirements.

Ada was developed under the auspices of the US Department of Defense, just as Cobol had been developed two decades before. Mike and Paul were familiar with Ada from their Air Force experience, and saw the aerospace industry as a receptive initial market for an Ada-based set of development tools supporting modern software engineering practices.

**TRE:** For those of us who are not so language-savvy, can you explain why a declarative language is so important?

**DB:** First, let me give you a simple situation where "meaning" can get lost quickly as a program gets compiled.

Suppose I'm creating an application in which I need to keep track of the current day of the week. Further suppose that I decide to represent the current day of the week in a variable named "I" -- where the value "1" means Sunday, "2" means Monday, ... and "7" means Saturday. While entering my program, a typo results in the following statement:

```
I = 8
```

Given the above mapping of "days of the week" to integers, this statement is clearly erroneous. But the compiler that translates my program into machine language is unaware of this mapping -- and processes my code without complaint. Hopefully this problem will show up during later testing, but that's a relatively expensive way to find problems.

In a declarative language, one can "declare" such mappings, enabling the compiler to detect errors such as the one above. We're expressing more than simple logical flow; we're describing the meaning of program elements in a way that allows the compiler to report inconsistencies as part of the compilation process. Languages like Ada (and today Java, C++, and C#) let me declare a variable of type "day of the week," and define its seven valid values. Any violation of the mapping will result in a compilation error; this is much more efficient than hunting down defects during testing or, even worse, in the field. This also makes for programs that are easier to understand:

```
Payday = Friday
```
is a lot more meaningful to a human reader than the earlier code fragment.

Its declarative nature does make Ada somewhat verbose. On the other hand, we typically write an application once but read it many, many times over its lifecycle. So languages optimized for readability -- even though they may require a few more keystrokes -- are preferable. That's very different from the approach taken by the designers of the C language, where optimizing for minimum programmer keystrokes produces applications that are quite terse.

**TRE:** But that also creates the need for lots of comment lines, so that later, other programmers maintaining the source code can better understand what's going on.

**DB:** Right. As software has become more pervasive, it has also become increasingly long-lived. This makes maintainability even more critical, since the original developers may no longer be available to correct defects or add features. The verbosity or terseness of the language is one of many factors, but a more readable language certainly helps improve the maintainability of one's application.

We were determined to provide a set of development tools that not only supported but also encouraged modern software engineering practices -- iterative development, modularity, abstraction, and declarative programming language. We believed that this would dramatically improve the success rate, particularly for complex, long-lived applications. Our mission -- enabling our customers to succeed in software development -- hasn't changed since Rational was founded.

**TRE:** You've spoken about development tools, but I've heard Rational's first product characterized as an "integrated programming environment." Is this just a fancy name for "development tools"?

**DB:** It's more than just a fancy name. We saw an opportunity to add value beyond providing an advanced, but loosely coupled set of tools. Several research efforts had produced "programming environments" tailored to specific languages like Basic or Smalltalk. Such programming environments combined development tool capabilities -- editing, compilation, and debugging -- in a single tool that was optimized for development in that language. So unlike a generic text editor, a Smalltalk programming environment could detect and report grammatical errors as you were entering the code. What had been separate mechanisms were connected -- integrated -- to improve developer efficiency. When the compilation mechanism detected an error, for example, it could display the offending code in the editor mechanism where the developer could easily correct it. These programming environments demonstrated an important point: Reducing turnaround time -- the interval that starts when a
developer completes a source code change and ends when a version of the application containing that change is ready for testing -- dramatically increases productivity, particularly if the developer perceives the process as interactive.

The programming environments produced by these research efforts were typically focused on individual developers. We decided to apply these concepts, but expand them to support teams building complex applications in Ada. That meant adding support for parallel development -- multiple developers working together on the same project. This support took the form of a configuration management mechanism deeply integrated with the editing, compilation, and debugging mechanisms. Besides the usual check-in/check-out functionality that enables developers to avoid over-writing each others' changes, this configuration management mechanism made it possible to answer questions like "If I change this public interface in version 1.2 of module A, what will be the impact on version 2.5 of module B and version 0.8 of module C?"

TRE: What projects were under way when you joined the company in 1982?

DB: Just one, but it was more than enough. When I joined Rational, the overriding goal was to create and release a team-oriented, integrated, interactive development environment for Ada. The development team had already overcome one significant problem -- a conflict among these objectives -- via a clever inversion of the traditional compilation process.

In order to provide accelerated error detection, compilers for declarative languages like Ada would assemble a comprehensive data structure, referred to as the intermediate representation, containing all of the declarative information gleaned from the source code. If no errors or inconsistencies were detected, the compiler would use the information in the intermediate representation to generate machine-executable code. The intermediate representation was large and took time to assemble; as a result, the compilation process was far slower than that of non-declarative languages. And unlike the situation with non-declarative languages, making a change in one part of an application written in a declarative language would often require re-compilation of the entire application to ensure that the change did not introduce an inconsistency. While this was highly beneficial from the perspective of improved quality, it made compilation decidedly non-interative, which decreased developer productivity.

To eliminate this conflict, our team organized Rational's development environment around a persistent, intermediate representation. Rather than assemble the intermediate representation after each change and then discard it when code generation was complete, the Rational Environment would treat the intermediate representation as a sort of permanent database, updating it as necessary to reflect changes in the source code; in fact, the process of updating the intermediate representation was accomplished transparently as the developer edited his or her source code.
This made error detection quite interactive: A developer could make modifications or add new code and immediately identify consistency errors without waiting for a compiler to re-analyze the entire application. Because it contained so much information about the application, the persistent, intermediate representation made it easy to provide convenient facilities for navigating and generally understanding the application under development. The result was an environment that enables interactive development with a declarative language.

**TRE:** That sounds in some ways like one of Rational's slogans: Resolving the speed vs. quality development paradox.

**DB:** It was the inception of the "speed-quality" duality that is still one of the things Rational hangs its hat on. And that was 1981. We called it the "software development crisis" back then.

We also understood that we were supporting teams of people operating in parallel, which meant that we needed support for configuration management -- version control -- to allow developers to work on parts in parallel without endangering each others' work. They had to manage multiple parallel versions: the currently deployed release, the next release, and maybe some long-lead development on the release following the next release. One important insight was the power of integrating configuration management and version control with the development environment. This seemed like a simple idea at the outset, but it helped expose another conflict in our objectives -- this time involving the use of declarative programming languages by teams practicing parallel development. Overcoming this problem required our team to extend the concept of modularity to include software subsystems, and provide direct support for these subsystems in our development environment. This experience base is one of several contributors to today's Unified Change Management best practice.

**TRE:** Then you had all the right ingredients, so your first ship date was...?

**DB:** Well, it wasn't quite that simple. This development environment had to run ON something, like the DEC [Digital Equipment Corporation] VMS operating system ran on the DEC Vax minicomputer. Unfortunately, our proposed development environment was not well matched to the capabilities of such hardware. We'd taken what was fundamentally a batch compilation process and made it interactive, demanding crisp response times. Worse, we'd introduced a very large data structure -- the persistent intermediate representation -- to which access was fundamentally unpredictable. Machines like the Vax and operating systems like VMS relied on caches and other accelerators for their performance, which in turn depended upon predictable access patterns; an application like the Rational environment would convert such accelerators into decelerators: They would rarely contain the desired information, but expend precious CPU cycles pre-loading and managing it.
Our solution was to design and build our own processor architecture and operating system. The operating system and development environment were built as one layered application that we ultimately named "the Rational Environment"; the machine was called "the Rational R1000," and was designed to provide direct support for both Ada and the manipulation of very large data structures. When I joined Rational in January of 1982, the basic concepts behind the Rational Environment were in place, and the R1000 architecture was largely complete. We just had to design and implement the Environment, and design and implement the hardware and microcode required to run it. The original beta schedule was something like nine months; it required closer to thirty. We shipped the beta version in mid-1985 and didn't deliver a production version that included configuration management until 1986.

**TRE:** It's hard to imagine that any company today would attempt to build new hardware, a new operating system, a compiler for a new language, and this new environment all at the same time. A venture capitalist would probably laugh and walk away.

**DB:** As many have said, we would never have embarked on this effort had we fully understood its difficulty; naïveté is occasionally an asset. But we were -- and still are -- willing to do *whatever it takes* to provide our customers with the means to succeed. Great people rise to extreme challenges. How do you avoid serialization by developing the software in parallel with the hardware? You write the software in Ada, and you create an executable environment based on Data General minicomputers that lets the software team develop iteratively before the hardware exists. Early on, we learned the benefits of using our own concepts and products — the *eat your own dog food* principle. How does a small hardware/microcode development team correctly implement an architecture so semantically intense that inter-task communication (Ada's Rendezvous) was implemented in one instruction? You create a multilevel simulation environment that verifies the design's ability to run critical parts of the nascent Rational Environment before anything is built, and you construct that simulation environment in Ada so that the hardware team will first-hand understand the software concepts. How do you prevent the Ada language from being stillborn due to the lack of commercially available compilers? You sell the Ada compiler used for Rational Environment development to Data General, thereby stimulating DEC to complete and release its Ada compiler.

**TRE:** Was it common for hardware engineers to have the programming skills required to build such simulation tools?

**DB:** It was certainly uncommon in those days; but that kind of versatility was -- and still is -- essential. It's critical that each member of the team understands the big picture, not just his or her personal responsibility or his or her team's responsibility. With shared context and trust, both the idea flow and the execution are amazing.
Rational started in 1981, and the large-scale demand for our products and expertise did not materialize until 1995. That's a fourteen-year gap that, for most companies, would not be survivable. Why did Rational succeed? We hired people who were better than we were, we trusted our people to develop solutions to what seemed like impossible problems, and we expected our teams to do whatever it took to succeed. These principles are fundamental to what we do and what we are. They're the reason why we're still here today.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
The Nature of Software: What’s So Special About Software Engineering?

by Philippe Kruchten
Rational Fellow

As engineering organizations across North America struggle with the concept of opening their doors to and registering -- or even licensing -- software engineers, questions naturally arise about what software engineering actually entails. How do we qualify and evaluate software engineers? How do we validate their experiences? A first reaction may be to approach these tasks in the same way that we have done for all other engineering disciplines. However, software engineering differs from structural, mechanical, and electrical engineering in subtle ways. The differences are linked to the soft, but rather unkind, nature of software. In this article, I explore four key differentiating characteristics:

- Absence of a fundamental theory
- Ease of change
- Rapid evolution of technologies
- Very low manufacturing costs

Absence of a Fundamental Software Theory

Despite all the research done by computer scientists, there is no equivalent in software for the fundamental laws of physics. This lack of theory, or at least the lack of practically applicable theories, makes it difficult to do any reasoning about software without actually building it. During design, software can be structured and partitioned into chunks, but the real thing (once it crawls inside a computer) is actually totally unstructured, so that anything that goes wrong somewhere can corrupt something somewhere else. The absence of solid and widely applicable theory also means that the few software engineering standards we do
have rely on good practice alone, whereas building codes in other
disciplines can trace their rules to sound physical principles.

**Ease of Change**

Software is, almost by definition, easy to change, so naturally,
organizations want to take advantage of this characteristic. There is
pressure to change software throughout its entire development and even
after it's delivered. If you're building a bridge, you don't have this kind of
flexibility. You cannot say, "Hmm, now that I see the pilings, I would like
this bridge to be two miles upstream." But it is very, very difficult to
change software in a rigorous fashion, with all ramifications of all changes
fully understood and completely coordinated. Again, because of the
absence of solid theory, it's hard to validate a change set and its impact
without actually doing all the changes. Most of the damage that is done to
software is done through changes.

**Rapid Evolution of Technologies**

Software development techniques, and the environment of software itself,
are changing at an extremely rapid pace that does not allow for
progressively consolidating a body of knowledge. This puts a lot of
pressure on companies to train and re-train their software engineers, and
some do not really understand why they have to spend four times more
per capita on training than do people from other disciplines. This makes
the initial training software engineers have received less important, except
for the most general education, such as math and so on -- especially when
this original training occurred twenty-five or more years ago. This rapid
evolution also means that it is more and more difficult to maintain and
evolve "legacy" systems -- as the recent Y2K scramble has demonstrated --
because the technologies used some ten years ago are not in use any
more, and people who still have mastery over them are rather rare. The
norms and standards also have to evolve rapidly to catch up with
technology evolution. Finally, software engineering, unlike other
disciplines, has not had the benefit of hundreds or thousands of years of
experience.

**Very Low Manufacturing Cost**

First, I would like to note a slight shift in paradigm. Software engineers
speak about design, but by this they mean only a high-level description of
their intent, and then they think of program construction as akin to
manufacturing. Actually it is not -- program implementation is more like
preparing a cast in mechanical engineering. Also, for a software engineer,
a prototype is roughly equivalent to a scale model; it's pretty incomplete.
The real "manufacturing" of software entails almost no cost; a CD-ROM,
for example, costs less than a dollar, and delivery over the Internet only a
few cents. Often it doesn't matter if the design -- that is, the initial
program -- is a bit wrong; we can just fix it and manufacture it again, as
we noted above in the discussion on ease of change. We hear people refer
to this as a "free bug fix release" or a "must-have upgrade." Clearly, this
combination -- ease of change and low manufacturing cost -- has led the
software industry into a pretty big mess. And these practices are
supported by outrageous licensing policies that allow the designer and manufacturer to assume no responsibility other than, in good cases, a promise that they will re-manufacture the product in a few days or months. You can't do that with a bridge or a car engine because the cost would be huge, and that forces engineers involved in building these things to get them right the first time.

**Engineering All the Same?**

For twenty years, refusing to acknowledge the four factors I described, the software industry has tried hard to pretend that software development could follow the same path as other engineering disciplines. We have failed. We also hoped that science would bring us solutions, but they have not been forthcoming. Why? To answer that question, let's look at two important ways that software engineering departs from other disciplines.

**Iterative Development**

The very rational and straightforward "waterfall" development lifecycle -- define and freeze requirements, create and validate the design, implement and test, then deliver -- works very well in many disciplines but has failed many times in software engineering. This project lifecycle does not accommodate changes: It does not allow you to really validate much, so you have to rely on your own warm, fuzzy feeling that "the design is OK." Nor does it allow for tactical changes in technology, or take advantage of the low manufacturing cost -- except for pushing undue costs on to consumers.

Today, software engineers take a more iterative approach to software development, which allows them to integrate changes, to refine and validate the design based on execution and not just examination, and to accommodate evolution in technology. An iterative approach would be impossible in other disciplines; you cannot build a bridge iteratively, for example.

Iterative development allows you to continuously verify the quality of a constructed prototype as opposed to demonstrating correctness a priori, based on fixed laws. Software has no laws that can ensure the ultimate product will perform as expected, but iterative development allows you to confront technical risks earlier in the development cycle.

Also, software engineering puts more emphasis on some techniques that have less importance in other disciplines, such as requirements management and change management, because requirements and other software artifacts may change throughout the development lifecycle and even after that.

**Component-Based Development**

Another dream of software engineers is to mimic with software what has happened in electronics or construction -- to develop families of standardized parts out of which you can build larger and larger sub-assemblies and ultimately complete systems. This sounds straightforward,
but actually very few can do it. Again, this is due to the lack of a strong underlying theory to rigorously define the components and their interface. The situation has not been helped by the rapid changes in technologies: No component family has had time to settle and develop as a self-sustaining industry. Only recently has a new sub-discipline emerged -- software architecture -- which tries to address, despite the lack of fundamental theory, some of the structural aspects involved in constructing large software programs and reusing software assets across product lines or product families.

So, if I agree with David Parnas¹ that software engineering should be treated on an equal footing with other engineering disciplines and not solely as computer science or some kind of enlightened craftsmanship, then I also have to acknowledge fundamental differences that make some of the more traditional approaches to engineering and engineering management inapplicable to software. Software engineering has developed its own approaches to manage its specificities over the last few years.

I believe that the registration process for professional software engineers -- as well as the construction of accredited software engineering programs -- must understand, acknowledge, and address these specificities.

---

**Notes**


**References**


---

For more information on the products or services discussed in this
article, please click here and follow the instructions provided. Thank you!
Introducing Rational Developer Network -- The New Online Learning and Support Channel for the Rational Development Community

Rational Developer Network, a new, password-protected site that is currently accessible to Rational Suite customers only, will soon be available to all Rational customers (watch for announcements via e-mail and on Rational.com). For Rational Edge readers who have not yet explored its array of resources, this month we provide an overview of what the Rational Developer Network is all about (see below) as well as a small sample of the practical, developer-focused materials available to its members.

- **ClearQuest to the Rescue**
  by Christian Buckley

- **Forward and Reverse Engineering of Code With Rational Rose and VisualAge for Java**
  by Shawn Keller

About Rational Developer Network

by **Paul Dreyfus**
Rational Developer Network Managing Editor

The purpose behind Rational Developer Network is simple: to help developers get software development projects done more quickly and easily, saving time and money. We offer a wide range of content -- Rational Unified Process® (RUP®) artifacts, documentation, technical notes, whitepapers, online training courses, how-to articles, and code and project artifacts -- to help acclimate Rational users to the Rational product family and accelerate the completion of their projects. Additionally, we host discussion forums so they can interact with other members of the Rational community, contributing their own experiences and learning from others. Our goal is to provide a single portal for Rational learning and support materials as well as to build an ever-widening community of
The Reasoning Behind Rational Developer Network

Before describing the organization and features of Rational Developer Network, let me provide a bit of the reasoning behind Rational’s decision to build a developer network. Since Rational first offered its best practices and tools, engineers have experienced substantial productivity gains by employing them. As with most complex development tools, however, even the most sophisticated developer had to climb a learning curve to get the most out of them. Users of the RUP, Rational Rose®, RequisitePro®, ClearQuest® ClearCase® and other Rational tools often take a class from Rational University or receive hands-on instruction from Rational Professional Services or other Rational consultants.

As we grew our community and the number of Rational developers began to grow exponentially, it became obvious that we had to expand this educational model. We wanted to make it possible for new users to get to work as quickly as possible. We also wanted the productivity gains made possible with Rational tools to be available to smaller development teams who might not have the resources to pay for consulting or training. And we wanted them to be able to apply the best practices and tools quickly to their specific projects, no matter what the target technology, runtime, or domain.

To do all this required a different approach to teaching practitioners how to use the tools and to supporting their use of them. We needed to give Rational users immediate access to the right set of materials to get started quickly and to answer their questions. We wanted to provide a place where Rational and industry gurus could share their knowledge and experience with a broad audience. We also wanted to make it possible for members to choose from a library of sample code and artifacts -- Rose models, frameworks, components, etc. - that they can copy into their project to get work done more easily and quickly. Overall, we wanted to provide easy access to the right set (preferably a small one) of applicable knowledge, not a huge amount of raw information of dubious value.

The solution: An online destination where developers can train themselves with formal, Web-based training courses and other reusable materials that are available any time, from any location. To be most useful, the destination had to let people interact with other members of the community, share expertise, and learn from others in the network of Rational developers. With such a developer network in place, our members could answer questions on their own -- either by researching its knowledge base or by discussing questions with the community -- without having to call a tech support engineer or bring in a consultant. And, when they decided to spend valuable time and money on instructor-led courses and consultants, they'd already know the right questions to ask, and could make the best use of their own -- and the experts' -- valuable time.

Member Participation Is Vital

During the early days of Rational Developer Network, subscribers will see...
the first steps toward fulfilling that vision. This is just a start; no developer network, including its content and community offerings, can be complete without significant feedback, and even contributions, from its membership. We offer the first set of content and services as a way of sharing with our members a relevant portion of Rational's expertise to offer immediate value, and to begin to get input, ideas, and expertise from Rational developers. An important part of this offering is Rational Developer Network's feedback and participation mechanisms. As they browse Rational Developer Network, we invite all of our members to take advantage of them.

**How Rational Developer Network Is Organized**

Initially, we've provided the following basic set of materials and services, all available through the links on our homepage under Knowledge:

**Knowledge Centers.** These are the core areas of our initial offering. Within each center are best practices, articles, whitepapers, documentation, training materials, and artifacts about topics of interest to a Rational developer, gathered by Rational's team of editors with the assistance of a variety of experts. We've organized our Knowledge Centers around three main categories:

- **Rational Tools.** Many of our members' questions will have to do with the use of a specific Rational tool. They can go to these centers for help in getting started and using the core set of tools within the Rational product family. Over time, we'll offer a Knowledge Center for each Rational tool or group of tools, and provide more materials about how to use the tools together, in an integrated fashion.

- **Software Lifecycle.** The RUP describes the phases of the software lifecycle, from inception to deployment, as well as the disciplines and best practices that are part of the software engineering process. We've sought out the experts to help people understand how to apply these RUP-based concepts to everyday development tasks. We start with the topics our research tells us developers need to know the most about, including Design, Analysis, and QA & Testing; we'll add topics to this area based on feedback and our ability to find useful materials.

- **Web Development.** Given the state of the industry, most of our members are facing one type of Web development project or another, so we focus on the implementation and construction of Web applications. Over time, Rational editors will expand this area with the most applicable original and reprinted articles, artifacts, and code samples they can find, again based on feedback.

We offer other areas, as well: Rational documentation and technical notes; Books Central, with selections from new and significant technical books; and Download Central, providing direct access to all the code and reusable artifacts on Rational Developer Network.

Soon, we'll be offering specialized areas about today's hottest technologies to help developers apply Rational best practices and tools to the cutting
edge of software engineering. We're also working with the major platform vendors and content providers to license best-of-breed technical materials for publication on Rational Developer Network. Throughout the topical areas, we have articles and papers from Grady Booch and other Rational experts, who make regular contributions to Rational Developer Network.

**Simple and Advanced Search.** All Rational Developer Network content has been tagged and organized so that searches result in high-value sets of returns. If our hosted content doesn't provide answers, then members can also use our search engine to search the entire Internet.

**Expert-Led Discussions.** Very often, the answer to a question or the help you need sits inside the head of another developer, someone who has been in your shoes before but hasn't bothered to record his or her experiences. When you use Rational Developer Network, if there's anything you can't find, if there's a burning issue that would normally make you get on the phone to tech support or a colleague, if you find something you disagree with or that you want to comment on, you can visit Discussion Central and contribute to one of the discussion forums. While you're there, you can arrange to have the discussion threads forwarded to your e-mail inbox.

Our discussions are monitored by Rational gurus and industry experts, who will be guiding the threads toward fruitful topics and weighing in with answers to commonly asked questions. This distinguishes our forums from the public newsgroups focusing on Rational tools, which have many virtues but offer no guarantee that developers will get definitive answers to their questions. Our industry experts will also be searching the discussion threads for particularly valuable input that could become an article on Rational Developer Network.

**Web-Based Training.** For the first time, Rational is offering training courses over the Web through Rational Developer Network, making available a growing number of Rational University courses through Training Central. Based on the materials taught by instructors in classroom settings, Rational's Web-based training courses teach the principles and skills developers need to make the best use of Rational tools. The difference is that they can take these courses any time they want, no matter where they are. There is no substitute for the classroom, but these courses can help developers attain a base level of competence quickly, easily, and inexpensively. We currently have a small set of Rational-specific, Web-based courses, surrounded by a wide course offering on tools and technologies from other vendors. Over time, Rational Developer Network Training Central will be the location of a complete Rational tools online curriculum.

**MyWorkspace.** Members can customize their experience by building their own Rational Developer Network Home page with MyWorkspace. This feature of Rational Developer Network lets users fill containers with just the content they want to see, whether about a specific tool, a part of the software lifecycle, or a technology. They can also add their own links to the Bookmark section so they don’t lose track of specific articles or other documents. Once they choose categories to build each container, the containers update themselves when Rational Developer Network editors.
add new materials that fit the categories they've selected.

**The Future of Rational Developer Network**

What's on Rational Developer Network today is just the start. In the near future, we'll be adding additional topical areas based on our ability to gather pertinent content and user feedback. We'll also publish new materials to the existing areas as often as we can. Our artifact and code library will expand with reusable artifacts and reference projects that developers can learn from and use to accelerate their own projects. We'll alert members to new, interesting developments through Development Resources, the Rational Developer Network home page.

Over time, we'll also add more features to Rational Developer Network. We'll continuously expand interactivity to make it easier for our members to participate in the Rational developer community. We'll work to make the material they need as easy to find as possible. And we'll provide new features that lead them through the variety of materials on the site to provide complete learning paths around significant subjects.

Our emphasis in offering new content and features will always be quality. We've published materials that we think will provide immediate value; we've focused on gathering knowledge that members can apply, as immediately as possible, to their work. We've spent time identifying each article and artifact and applying relevancy tags to each piece; using our simple or advanced Search feature will yield a small but useful set of returns. We'll continue to publish only the most useful materials and offer only features that make those materials easier to use.

All these ideas are subject to change and elaboration, based on our members' feedback. As I've said, we're providing an initial set of content and services both to offer immediate value and also to obtain feedback so that what we build in the future is ever more useful. We urge our members to join the discussions, fill out the evaluation forms at the end of each article, and use every opportunity they can to let us know what they need. We promise to respond, and to continue to build Rational Developer Network in a direction that's as useful as possible to the Rational developer community.

-Paul Dreyfus, Managing Editor pdreyfus@rational.com

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

by Christian Buckley
Sr. Technical Product Manager
e2open

[A dramatization -- well, sort of] In the not too distant past, I was sitting at a customer site on the other side of the country, waiting for a software vendor (whose application was integrated into my company’s software solution) to respond to a customer question about a piece of functionality that wasn't working as documented. As I sat there waiting for the phone to ring, my mind wandered a bit, and I found myself daydreaming about creating some kind of technology that would allow me to reach through the telephone lines and smack the next person who told me, “Well, we can't seem to recreate the problem over here. Can you tell me again what the problem is?”

I can't tell you how many times I've heard that response. Well, the guy finally called, and his response was as expected -- "Can you help us understand the problem? What is your user trying to do? We can't recreate your problem, and we need more information." After a red eye flight to the East Coast, followed by six hours of user training and a poorly re-heated fast food lunchtime feast, I'm on the edge, to say the least.

OK, let's look at this logically -- how can he tell me he can't recreate the problem, and then in his very next breath ask me to explain the problem? If he doesn't understand, what was it that he was trying to recreate? What's most frustrating about the situation is that it is -- for the most part -- preventable.

There seems to be an endless volley between product management and engineering and engagement personnel to clarify exactly what the customer wants and needs, and what the development team can deliver. We assume there is a "process" to handle the information being passed
between organizations, and between our customers and pre-sales teams. Apparently the "process" is to send an e-mail or jump on the phone and call someone down in engineering. What information was captured and to whom the bug or enhancement was sent depends entirely on the person who documents the issue.

**Eye Protection Recommended**

But wait -- it gets even better. Then comes the domino effect:

The customer doesn't think his input was adequately prioritized by the salesperson, so he contacts one or two other people at the company. The salesperson enters the request into the bug/enhancement tool, as does one of the other customer contacts. The QA team sees the new items, and flags the development team for an update to each -- which look similar, but are identified as two different bugs. Meanwhile, the salesperson and one of the contacts have discussed the issue with a member of the product team, and the second contact has discussed the issue with a second member of the product team (are you following this?).

Oh, did I mention that all three had a different understanding of what the problem was?

One of the product managers adds the issue to his next review meeting agenda, while the second product manager goes straight down to engineering to talk about short-term solutions. Based on that conversation, engineering adds a small patch to the upcoming build schedule. Three days later, the salesperson logs into the bug/enhancement tool for an update. He doesn't see an update, so he contacts someone on the QA team, who refers him to engineering, where he learns that a patch is being applied in the next build. He then contacts the customer to give him the great news. (I probably should have provided a diagram for all of this!) A week later the build is completed, two weeks after that QA approves the release, and the customer logs on to the new build -- only to find that the problem he reported is still there.

Ouch.

OK, while I stand by my claim that the above account is largely a dramatization, this problem is all too real. Some of the finest, most well-organized teams have fallen into this communication trap. To overcome it, you need to automate, plain and simple.

**It's the Tools, Stupid**

Water flows the path of least resistance. In my experience, so do communications between customer-facing groups and product development teams. We all hate to admit it, but when it comes to solving complex software problems or responding to detailed customer functionality requests, development teams usually do what it takes to get the product out the door -- even if it means bypassing specific customer requests with the knowledge that they'll have to revisit the same problems later. This can be a financially risky decision. Keeping development on
schedule and providing continual feedback to your customers should not be mutually exclusive.

The key to resolving this issue is knowledge. Knowledge comes from understanding the root of the problem. To understand the root of the problem, you need to understand the business use case under which the problem was identified, and for that you need good documentation and a solid feedback loop between your customer, your customer-facing team, and your product development organization. Documentation is the key. I think I said it best when I wrote, "The aspects of good documentation are simple: They must be clear, concise, consistent, and most importantly, they must deliver the expected results. This point deserves emphasis: It is one thing to create documentation that fulfills the technical requirements, but an entirely separate issue to create documentation that anticipates questions and provides solutions."

But who has time for clear and concise documentation? Everyone, if you have the right tools and a pattern to follow. Put the tools in front of the people, and make them as simple as possible (the tools, not the people). How do you get your globally dispersed sales team to input customer bugs and requirements, you ask? I suggest rolling out Rational ClearQuest®.

No, I am not a Rational Software employee -- I'm someone who had a problem and found a solid solution. There are a number of free tools available on the Internet; however, I suggest using something a little more robust than the shareware versions -- mainly for security reasons (access rights). The model I designed for my team was something like the following diagram:

![Figure 1: Model for Handling Customer Input](Click to enlarge)

During any implementation, you must make the interactions clear to everyone within the feedback loop. Here is how our process works:

- The customer contacts the appropriate customer representative with an issue or a suggestion. We trained everyone to point any user bug or enhancement request to someone on the team with ClearQuest access, so that we had one group of people entering all issues in one format. Believe me, people don't seem to mind passing this responsibility over to someone else.

- The customer rep documents the issue/suggestion, enters the information into the repository, and receives a tracking number. At any point, the rep can access the system and get a status on this item. Each time the issue is edited (feedback provided), they can
receive an e-mail notice.

- Bugs are sent to QA, while enhancements are sent to product management. You don't have to set your system up like this, but we wanted to make sure the product management team (our R&D shop) was responsible for all enhancement-related requests, so they could stay close to the customer usage patterns.

- The bugs/enhancements are reviewed [Does the request make sense?], qualified [Is this a valid request?], prioritized [Where does it fit into our current activities?], and consolidated [Do I have any similar requests already in the system?]. ClearQuest is then updated with the appropriate feedback, which, in our case, automatically notifies the customer rep who entered the information. Again, you do not have to set your system up to notify the rep every time a change is made, but we felt we could better serve the customer by staying on top of any changes to the issue. You could just as easily log onto the system a dozen times a day -- but that's what we chose to do.

- Based on the update, the appropriate development organization receives the request and fits it into their development schedule. These teams do not receive the bug/enhancement until QA or product management flag it. I found this to be a great feature in itself. On previous systems, anyone who logged into the system, including the development team or an outside vendor, would have access to every single bug or enhancement that was logged into the tool. Not anymore. They now only see what we want them to see.

- The development organization works with QA to test any new code. Build, test. Build, test.

- Once approved, QA updates ClearQuest. Another great feature -- the issue isn't closed until QA -- and QA alone -- decides it is closed.

- The customer rep is notified that the item has been closed. E-mail notifications are great.

- The rep contacts the customer and notifies him of the closure.

As with any tool, it's not going to solve every problem, but if you don't have something similar in place, it's a great place to start. Based on my experience, here's the roadmap to success:

1. **Train your customer-facing folks to properly capture the information.**
   
   Make yourself the documentation evangelist for your team or company. Advocate capturing requirements via entire sentences -- versus chopped-up fragments of what is generally regarded as written language. Persuade your team to break each issue down into logical chunks, and to carefully capture the steps of the problem AND the expected results. If documenting an enhancement request, include detailed use cases that explain how the customer would use the new functionality and why the request is important. This goes a long way when the product team is prioritizing issues. A well-detailed description will move items to the top of the list,
believe me.

2. **Automate the mechanisms for entering information.**
   I’ve used both homegrown solutions and Internet freebies (Bugzilla), but I prefer ClearQuest. Implement this tool in a way that allows key personnel to easily access and share information. This should be a no-brainer -- log in, define the appropriate component, prioritize your information, and then leave your data. The easier you make it, the likelier they are to use it.

3. **Set up a process for deciphering this information.**
   Once you have the information, what will you do with it? It's one thing to set up a vehicle for capturing and documenting customer information, it's another thing to get your product development team to respond -- so that your sales team can provide timely feedback. Make the tool part of your daily/weekly/monthly process for reviewing bugs and enhancements. Assign responsibility to some lucky product manager, and keep the repository updated.

4. **Assign to necessary technical leads.**
   The product team has responded, and the appropriate technical lead -- whether part of your internal engineering team or part of a third-party development organization -- is automatically notified. Actually, this is a big incentive for my own product management team. They are acutely aware that the sooner they respond, the sooner the issue gets assigned to an engineer and moves off their own plate. How's that for motivation? Instead of the engineering team lead receiving e-mail notification for every single problem that comes across the product team, the system can organize and delegate based on components -- and even based on severity.

5. **Update the repository at each step.**
   At each step of the process, the repository is updated, the customer advocate has access to that information, and notifications are sent automatically. This is the key to cleaning up this entire process. Instead of pushing report after report across the organization, simplify simplify simplify. One version of the truth.

6. **Train customer-facing folks to log in and report back to customers.**
   Every Tuesday morning, I know that I can log into the system and get the latest update on any issue or enhancement that was entered into the tool the previous week. How powerful is that?

And in case you were wondering what happened to me and the customer out on the East Coast, things worked out fine. I bought the customer an expensive dinner, and sat and waited for a couple hours while a team of 5 or 6 engineers back in California tried to fix the software issue. In fact, the problem was resolved by the time we finished our lobster dinner. Sometimes the job has its perks.

*NOTE: This article was originally published on Rational Developer Network, the learning and support channel for the Rational customer community, currently available only to Rational Suite customers. If you are a Rational Suite customer and have not already registered for your free membership, please go to [www.rational.net](http://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!
Tracking Action Items and Projects with Rational RequisitePro

by Mike Taylor
Software Engineering Specialist
Rational Software, Australia

Rational RequisitePro® is a great tool for tracking requirements through projects, but did you know that it has the flexibility to track other project and organizational data as well? It's great for capturing and maintaining minutes and action items from project meetings. It also provides an easy way to generate and update status reports. This article tells you how to get started.

Why use Rational RequisitePro for meeting minutes instead of plain old Microsoft Word? Well, let's suppose you use Word. You start by taking minutes, noting all the action items you want to assign to someone. So far so good. At the end of the meeting, you write up the minutes and distribute them. Fine.

At the next meeting, however, you need to discuss all those action items you assigned. Simply reporting "done" or "not done" is not really sufficient; you need to record what progress has been made and assign any additional action items that grew out of the original list. Further, people working on these items may want to record discussions about particular activities. As you hold more meetings and open and close other actions, your text editing features become really tedious to work with. Plus, over time, your action lists and discussion notes grow to pages and pages, obscuring the real "meat" of your meetings.

Does this sound familiar? If so, then you can improve the situation. If you're already using Rational RequisitePro for requirements management (or would like to), then you can also use this great tool to capture and manage those minutes and action items.

Tracking Action Items from Meeting Minutes
With Rational RequisitePro, you can set up a specific requirement type (e.g., ACT) to capture action items. The attributes for this type might include the following: assignee, priority, status, date opened/date closed, etc. Then, you can create a list of action items like the one on the left side of the screen in Figure 1.

![List of Action Items in Rational RequisitePro](image)

**Figure 1: List of Action Items in Rational RequisitePro**

On the right side of the screen, you can set priorities, assign the action items, and then maintain a running report on the status of each item. This is a great benefit for all your team members. Instead of waiting until the next meeting to find out whether something is being worked on and what has been done, they can open Rational RequisitePro any time and find out.

To print out the latest minutes in your company standard format for distribution, you can use Rational SoDA. SoDA generates complete documentation by automatically extracting data from project databases -- in this case, extracting the minutes and actions from Rational RequisitePro.

**Tracking Projects and Customer Accounts**

You can also use Rational RequisitePro to track the status of projects and customer accounts. Suppose you have a weekly meeting at which you discuss various customers and projects. Given that the discussion points are relatively static (i.e., the projects won't change from week to week), you can use a second Rational RequisitePro requirement type (CUS) for
projects and customers, as shown in Figure 2.

![Figure 2: List of Customers and Projects in Rational RequisitePro](image)

Then, to record progress on a project (customer), you can create a Discussion in RequisitePro (see Figure 3); the Discussion title should reflect the meeting date.

![Figure 3: Creating a Discussion in Rational RequisitePro](image)

If your Discussion notes include any new action items, you can add them to your ACT list and use the "Traceability" function to link them to the
correct customer or project (see Figure 4).

Figure 4: Using the Traceability Function in Rational RequisitePro

Creating a Status Report with Rational SoDA

Once you have finished your weekly meetings, you can use the power of Rational SoDA templates to create a standard status report to send to team members and managers. The SoDA report template will prompt you for the date of the meeting and then do the rest, extracting the relevant (CUS) listings and their Discussions, as well as open action items from your (ACT) list (see Figure 5). Anyone who has ever labored over a status report should recognize what a blessing this automated tool can be for busy managers and project leaders.
Figure 5: Status Report Created with Rational SoDA Template

SoDA can provide different views of the data, too. To see (and print) all the action items for a particular project or customer, for example, you can use the "Trace Out-Of" view, as shown in Figure 6.
Conclusion

It's no accident that Rational RequisitePro and Rational SoDA serve as great tools for tracking and reporting status. The built-in capabilities for requirements analysis already include powerful ways to understand, manage, and continually view the status of what needs to be done in your software development projects. RequisitePro and SoDA give you easy ways to:

1. Report much more than "done" or "not done" in your status tracking. You can record what progress has been made and assign any additional action items that grew out of the original list.

2. Record discussions about particular activities, so that people working on these action items see a dynamic, up-to-date context for the changes that occur.


As I mentioned at the beginning of this article, this is simply a way to get started. The more familiar you become with Rational RequisitePro, the more you can explore your own techniques for status tracking and reporting.

Download a sample Rational RequisitePro Minutes database.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
Getting started: Instructions for Downloading and Customizing the Sample Minutes Database

Instructions from Tracking Action Items and Projects with Rational RequisitePro

Before you follow the instructions below, check to ensure that you have Rational RequisitePro v2001A and a compatible version of Rational SoDA loaded on your computer.

Downloading and Browsing the Sample Database

1. Download and unzip the Rational RequisitePro project archive of the Minutes database from ftp://exchange.rational.com/exchange/outgoing/.../database with Data.zip to your local machine.

2. Open your local copy of the Minutes Project in Rational RequisitePro.

3. Open the ACT:actions attribute view and browse the actions. Double click an ACTION to open it and view the text field in the General tab. Note how status is recorded over successive meetings in the ACTION description field while the action item is still open.

4. Close the actions view

5. Next, open the CUS:Customer attribute view and browse the entries. You can store any status item within this requirement type (e.g., Projects, Customers -- any regularly reported item). Double click any red triangle in the top right-hand corner of the STATUS item title to view discussions about various status items. For any Customers without a red triangle, create a new discussion, as depicted in Figure 3.

Generating the Initial Rational SoDA Report

1. Open your local copy of the SoDA report template (Minutes SoDA Template.doc).
2. Run the SoDA report (SoDA->Generate Report). When prompted by SoDA, enter the string 7/7. This tells SoDA the date string in the discussion title to use. See Figure A.

![Figure A: Generating Minutes from Rational SoDA](image)

Enter the date string for your discussion where indicated -- e.g., "7/7."

3. View the report. It should contain items extracted from the downloaded project database and look similar to Figure 5.

**Customizing the Database for Your Environment**

To start using the database for your own project or organization, you first need to create your own Minutes database.


2. Customize the SoDA template (Minutes SoDA Template.doc) with your company or project MS Word styles, headings, and so, on as you see fit.

3. When you generate your report, the file will (by default) be called Minutes SoDA TemplateRpt.doc. You should change the name to reflect the current minutes: e.g., Minutes 10/25/01.doc. This is important, because each time you generate a new report, it will overwrite the previous version if it has the same name.

Once you follow these steps, you're ready to begin using the database. Just follow the suggestions in my [article](article), and don't be afraid to do a little experimenting on your own.
For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!
The first two chapters were about the software product; this chapter turns our attention to the software project. It introduces you to some powerful mental models for approaching software projects. By understanding and internalizing these models, you can develop good instincts for leading development efforts.

The first two chapters described the software product, its quality attributes and its specification. We now turn our attention to the way in which software is developed. This chapter is an investigation of the software project. All software development projects have certain essential characteristics. Software development can be seen in various perspectives:

- As an exercise in collaborative problem solving
- As a kind of product development
- As nonlinear, dominated by the interactions of the participants

What follows is an explanation of each of these perspectives on software development, along with their implications for leadership. By understanding and appreciating these characteristics, you will have a mental model of software development that will help you make the decisions as if by second nature.
3.1 THE DEVELOPMENT PROBLEM

The problem facing the development leader is the creation of quality software within a specified time and budget. We will call this the development problem. Recall, in Chapter 1, we defined quality as meeting the stakeholders’ needs.

One traditional approach to solving the development problem is to create a detailed project plan. This plan has fixed content, schedule, and budget. Once the plan is developed, all efforts are directed towards meeting the plan. Much of the literature restricts its attention to this solution. In fact, at least one well-known consulting firm defines project success exactly in these terms.

However, this connect-the-dots approach to development is unrealistic. Generally, one does not have enough detailed information to solve the development problem using a fixed, detailed project plan. The requirements are not sufficiently enumerated or even understood, the design is yet to be discovered, the estimates are imprecise. Moreover, as we will see, projects are inherently unpredictable; attempting to hold to a highly detailed plan is futile. A more realistic approach to software development must be found.

We start our exploration of understanding how to solve the development problem by exploring problem solving in general.

Consider how you solve a mathematics word problem. First, you make sure that you understand the problem. You make sure you understand the assumptions and what constitutes a solution. Perhaps you draw a diagram or recast the problem using a formula.

Once you are comfortable that you have a sufficient understanding, you can decide how to approach the problem. You might realize that standard algebra can be applied or that the problem is similar to one you have solved before. As you try to apply your approach, you may realize that you are missing necessary data or that you must go back to enhance your understanding of the problem.

Then, when you are confident that you fully understand the problem and believe your approach will work, you implement the solution. With luck, your approach is viable. However, you may discover that your approach is problematic or may not work at all. In that case, you proceed to refine the approach or find an alternative.

Finally, you verify the implementation by checking to make sure your solution solves the original problem statement. If it does not, you check your work, reconsider your approach, and possibly even rethink your understanding of the problem.
No matter what problem you are solving, from developing a high-energy physics model to installing a home network, you go through the same four phases:

1. Understanding the problem
2. Finding an approach
3. Implementing the approach
4. Verifying the solution

In software development, the entire team needs to solve the development problem collectively. The team as a whole must go through the problem-solving stages. The leader must take his or her team through this process.

3.2 DEVELOPING PRODUCTS

Product development is the discipline of creating and bringing to market consumer or business products. This field also must deal with a problem that is like the development problem. The product development leadership must find a way to design and bring to market, in a timely and affordable way, a product that meets the stakeholders’ needs. Going deeper, we find several similar challenges. The product and software development teams must

- Create an innovative, elegant design that addresses the requirements in some optimal way
- Understand and adopt the latest technology appropriately
- Determine and prioritize user needs while meeting schedule and budget limitations
- Deliver a quality product that will perform well in the field
- Lead a team to achieve a robust, expandable, maintainable design
- Coordinate the activities of multidisciplinary teams
- Integrate technical and marketing strategy
Product development managers are not particularly concerned with generating documents. Their management approach to product development is as a collaborative problem-solving exercise.

In addition, many product developers must worry about manufacturability, generally a small problem for software developers. Therefore, while a few of the details are different, the overall problems are the same. Software leaders can benefit from the lessons learned from the product development community.

3.2.1 Product Development Lifecycle
There is no official product development lifecycle. From the various references given at the end of the chapter, you will find the following product development lifecycle is typical. See, for example, Ulrich and Eppinger [2000].

- **Knowledge acquisition**—gaining an initial understanding of the problem
- **Concept development**—creating a design approach to meeting the market opportunity
- **Product engineering**—implementing the concept design and adjusting it as necessary, adding design details to complete a full, detailed specification
- **Pilot production**—building a functional model to verify the solution and addressing any unresolved manufacturability problems

This development lifecycle ends at acceptance by manufacturing. Final bugs may be ironed out during manufacturing ramp-up.

Note how well these phases of the product development lifecycle align with those of problem solving listed in the previous section. During the knowledge acquisition phase, team members do everything necessary to understand what needs to be developed. They may interview users, read marketing reports, and consult with the sales organization. They may meet with management to understand business needs and the marketing strategy. They may develop a competitive analysis and study the underlying technology. The outcome of this phase may be a business case, a product proposal, a vision document, or a requirements specification. In short, the team understands the problem to be solved. When the team thinks it knows enough about the product, it is ready to proceed to developing the conceptual design.
During the concept development phase, the approach to solving the problem is determined. The development team creates a high-level (low-detail) design of the new product. They may draw pictures, develop a virtual prototype on a computer, or create some sort of mock-up. In the auto industry, they build life-sized clay models. An equipment manufacturer may construct a case or cabinet showing dials and displays (the user interface). This mock-up is tested with marketing representatives or a customer group in the user community. Product features are continually re-evaluated, and tradeoffs are made among the market value, the development risk, and the manufacturing cost. During concept development, the team may also consider several alternative design approaches and the tradeoffs among them. The alternatives may be evaluated in terms of the design quality attributes (for example, robustness and maintainability) and other product issues (for example, manufacturability and the cost of the bill of materials). Each design alternative consists of a list of the major product components and how they interact. This phase ends when the team thinks it understands how it plans to shape the design and agrees on the major components.

During the product engineering phase, the team implements the conceptual design. It fleshes out the design details until a full, detailed specification is completed. They may farm out the component designs to different teams. The development team continues to build a series of prototypes that reflect the increasing detail in the design. Team members evaluate the prototypes with respect to quality and manufacturing concerns. For example, two components may not fit together well enough to permit assembly without frequent breakage, or one component may interfere with the replacement of another in the field. Based on these determinations, the component design may be updated, requiring a trip back to the drawing board. This phase ends when the team is comfortable that any remaining design problems can be uncovered only in pilot production.

Eventually, in pilot production a fully functional version of the product is built and ready for transition to manufacturing. In this phase, the team verifies that they have solved the development problem. During manufacturing ramp-up, the development team addresses the operational issues of construction and assembly in the factory setting. They may discover last minute glitches that require some redesign. This phase ends when the product is ready for full production.

Given that the problems faced by software and product development are so similar, it is reasonable to expect that software development should follow phases similar to those of product development. This similarity will be discussed in detail in Chapter 5.
3.2.2 Phases and Iterations

One of the salient features of product development is that as the team moves through the phases, much of the team’s activity consists of building a series of product iterations. Each iteration is a version of the product. As the development goes from phase to phase, the product iterations become increasingly close to final design. In the concept development phase, the iterations serve to help understand the product requirements. They may be mock-ups of the product for market testing. For example, a medical instrument maker may develop a version of the instrument with all of the buttons and knobs and no internal electronics. This mock-up can be used to test out the industrial design with prospective users. Later iterations can be used as a proof-of-concept to test out the practicality of the conceptual design. The final iterations might be used to address reliability and manufacturability issues.

Product development provides some important lessons:

- The maturity of the overall product design is marked by completion of phases rather than by completion of documents.
- Development progress is attained through product design iterations.
- The overall design is tested on an ongoing basis.
- A continual focus on meeting stakeholder needs is maintained throughout all the phases.
- Throughout the process, attention is paid to adjusting the product features to address development and market risks.

To summarize, in product development, the team goes through the phases of problem solving by developing a series of product iterations. As we will see, this approach provides a good model for software development.

3.3 SOFTWARE PROJECTS ARE NONLINEAR

Common sense flows from a shared view of how the world works. As shared world views change, so does common sense. Examples of how common sense has changed over recent decades include attitudes toward gender and race, as well as how teams work together.
The old common sense about software development was that the process was linear. The common belief was that teams should be managed like machines, each member doing a highly structured task while working as much as possible in isolation. This approach, sometimes called *scientific management* [Accel-Team.Com, 2000], sprang from the implementation of the assembly line in the early twentieth century and peaked with time-and-motion research in the 1950s. In this mechanistic approach, each person in the business process did one job repeatedly and, presumably, expertly. When their task was complete, they passed the item to the next person in the process. No one worker needed the big picture; they just focused on their own task. Interactions were minimal and the result was exactly the sum of the tasks.

Although an assembly line is efficient at producing many identical items that meet rigorous specification, it was found to have important limitations. First, the mechanistic approach did not provide a means for discovering and addressing certain quality issues. Some defects arose from the coupling of the tasks, from how people work together. For example, suppose one person follows instructions in placing a part on an assembly line. The next person sees that to place his or her part, he or she must hammer it in, degrading the product. If the two workers could collaborate, they might find a way to place the first part so that the second part fits easily. Another problem with scientific management was that people did not like being treated like machines. This management approach resulted in an adversarial relationship with management. The history of labor relations in the auto industry illustrates the point.

Some people see a mechanistic approach to business processes as common sense. Each task must be done. Why not analyze each task in detail so it can be carried out in the most precise and repeatable way possible? The *waterfall* software development process is an example of this approach, and early literature on the waterfall process makes such claims. The problem is that this mechanistic approach makes a leap of faith: that the interaction between team members is as simple as handing off a part in an assembly line.

A consequence of this linear thinking is the belief that the process, not the skills of the individuals, is the dominant contributor to success. The leaders of organizations that adopted this approach invested heavily in corporate processes and their enforcement. Their belief was that if they hired staff to fill the process roles with limited skills, the process itself would magically lead the team to a solution to the development problem. They held to this belief even though the research of Boehm [Boehm, 1981, 2000] and others shows that staff skill is the dominant success factor.
In what follows, we will show that process is important but it is not a substitute for skill. The process must provide a means to enable the skilled staff to work better together. To understand the role of process, you first must understand how members of the team need to interact. Today, we understand that development teams act more like societies or ecosystems than assembly lines. Each member plays a role in the community. These roles, not the artifacts, determine the interactions. With the right kind of leadership, the team members can come together in unexpected, but functional, ways to solve the problem. The dynamics of societies is nonlinear.

The old common sense was about control; the new common sense is about leadership.

### 3.3.1 Nonlinear Dynamic Systems

A system is **dynamic** if, for all practical purposes, it changes over time. A bridge may rust or experience metal fatigue, but it is more or less static. The stock market is dynamic; so is the marketplace. So is your software team: Every day, the people and the work are different.

A dynamic system is **nonlinear** if the response is not proportional to the input; that is, if small changes can lead to large reactions. Machines tend to be approximately linear; most real-life processes, including software development, are nonlinear.

To understand how a nonlinear system operates, consider the following thought experiment:

An automobile is placed in an empty, uneven field. You are asked to set the steering and speed, and to add the right amount of fuel so that the car, unoccupied, will arrive and stop at a specified target.

On the first try, you point the car in the right direction, compute the necessary fuel, set the speed control, and release the car. The car is buffeted by the wind, affected by the slope and bumps in the field, thrown off track by a pothole, and misses the mark. You try again and again, making all sorts of adjustments, and the car repeatedly misses the target.

You decide to take a systematic approach to the problem by running a series of experiments. You build a data table of initial conditions for the car, including the starting position, the angle of the steering wheel, the amount of gas, and the initial velocity. You set up the car with each of the initial conditions, let it run, and track where it ends. With enough data, you hope to be able to find the right settings to...
have the car arrive at the target. You plan several runs with each of the settings and average the results. When you run the experiment, you are amazed to find there is no predictability. The car ends up somewhere wildly different each time, with differences so varied that the averages are not reliable.

Eventually, you realize that unless the field is bowl-shaped and the target is at the bottom, the goal is unachievable. There is no repeatability. You observe that just a little steering would make all the difference. You realize that the outcome is affected by nonlinear interactions: interactions internal to the automobile (suspension, tires, steering), as well as by the outside temperature, the shape of the field, the wind, and many other factors. Each time you run the experiment, these nonlinear interactions affect the outcome. Because the car running in the field involves the nonlinear interaction of all these variables, it is a nonlinear dynamic system.

Scientists and mathematicians have come to understand the nature of such complex systems as the car in the field. An area of study, often called chaos theory [Lewin, 1999; Kauffman, 1996], is used to explain systems that involve many interacting entities. Chaos theory has been applied to weather systems, the dynamics of the marketplace, the origin of life, and, for about a decade, business organizations.

Some fundamental principles describe how nonlinear systems work. They provide guidelines for effective management of complex organizations such as development organizations. Most nonlinear systems are in one of three states, each found in business organizations.

**State 1. Chaotic: Unpredictable and unadaptable**

The behavior of chaotic organizations is always changing. The workers’ tasks change frequently, so no one is sure who is doing what or who is responsible for what. People come to work each day trying to find a way to move the project forward. Attempts to get things moving fail. Interactions generate friction, and there is no discernible progress.

In disorderly projects, under chaotic conditions, team members do not communicate. They are continually in each other’s way, unsure of what their job is. They spend more time trying to coordinate their efforts than moving the project forward. This kind of internal friction generates heat and not light.

Staff members of chaotic projects are emotional in interviews. They usually complain that what they do does not meet their job description, that they do not get credit for their efforts, that their time is wasted in unproductive meetings, and that their management does not seem to have a clue about what is really going on.
State 2. Stable equilibrium: Predictable, but unadaptable

Systems in a stable equilibrium behave as the car would if the field were a bowl. No matter where you set the car in the field, the car winds up at the bottom, in a state of equilibrium. The outcome is always the same. These systems require a large amount of external energy to move them from equilibrium. To carry the metaphor further, moving them from their stable state for the long term is like carrying the car out of the bowl. Small changes will not achieve this result.

Some managers believe that a stable equilibrium is good for their organizations. They perform year after year, seemingly impervious to the world around them. Their groups are not only easy to manage; they are hard to damage, even by poor management.

Sounds good, but there is a serious downside to the equilibrium state. Such organizations are very difficult to change, like the car at the bottom of the bowl. When the pressure to change is on, they resist new behavior; when the pressure is off, they revert to old behavior. These organizations do not compete effectively when changes in mission or technology are necessary. There are many examples, such as organizations comfortable with 1960s mainframe-based development methods. Given the amount of change in our field, these stable organizations are doomed. However, organizations in stable equilibrium are the exception.

State 3. Edge of chaos: Unpredictable, but adaptable

For most real systems, small changes in initial states can result in very different results. No amount of precision and detail is enough to set these systems on a course to a predictable end. Unpredictable behavior is intrinsic to these systems and cannot be overcome.

There is a state that is not in equilibrium and not fully chaotic. When systems are in this state, without steering it is impossible to predict their future state from knowledge of the present, like the car in the uneven field. However, these unpredictable systems do respond to external influences, so that they can be influenced to achieve a useful purpose. In the automobile example, a little steering makes it easy to hit the mark.

Edge-of-chaos systems have several remarkable properties that form a basis for understanding how best to organize and lead complex enterprises such as software development organizations. The first property is that small changes have large effects. As with the car experiment, a small change in steering direction makes a big change in the car’s eventual position. This property is a defining char-
actoristic of edge-of-chaos systems. It follows that it is impossible to predict exactly how these systems will respond to a plan or a set of directives. In fact, the greater the detail, the greater the unpredictability.

Fortunately, these systems are manageable, if not controllable, as a result of perhaps their most remarkable property: They spontaneously organize themselves into apparently orderly systems that adapt to changes. However, for this to happen, the system must consist of independent entities with the freedom to interact. With this freedom, the entities create the necessary connections and communications paths. System order does not arise from being controlled, but instead by being managed. In fact, any attempt to apply control may result in the system becoming locked in a rigid equilibrium structure that cannot make the necessary connections to deal with a changing environment.

### 3.4 TEAMS AS DYNAMIC NONLINEAR SYSTEMS

Development teams are made up of interacting individuals with different perspectives who generally do what needs to be done to complete a project. There are designers and testers, people who understand the application technology, and experts in software technology. Each is important, with unique experience, perspective, and priorities; each has a role to play. Throughout the project they need to interact, to communicate, to raise and address issues. As the project proceeds, the nature of the interactions changes. Each individual’s interactions combine to become the team behavior. The theory of dynamic systems applies to this collective behavior of a team performing a development project.

Individuals typically fill one or more roles on a development team.

- The project manager, or lead, develops and maintains the project plan, including cost and budget; prioritizes and schedules the content; staffs the project; and provides day-to-day leadership.
- The project architect looks after the overall design and associated quality issues. Responsibilities include maintaining the integrity of the UML views, especially the top-level logical decomposition, and the problem report database. Brooks [Brooks, 1995] and others (e.g., [Booch, 1995]) consider the architect a key team member.
• The business modeler models the business process that the system under development is intended to support.

• The project requirements analyst analyzes the business model and other user input to develop the requirements database, including use cases and supplemental requirements.

• The developer designs, implements, and tests individual object classes.

• The integrator or configuration management staff defines the project code library structure, instantiates the configuration management environment, designs and implements the method of compiling and integrating the code (the “make” files), and creates the builds for testing and delivery.

• The tester develops test plans, carries out system and component testing, and generates test and problem reports.

• The documentation writer creates the help file and the content of the user guide.

Large projects require partitioning into several development teams, with each team responsible for some number of logical subsystems. With the exception of the documentation writer, each role should be instantiated within each team and at the project level. For example, there should be a project architect and a team architect on each team. The project architect looks after the overall system design; a team architect looks after the design of the assigned subsystems. All of the architects should form a joint design team. This sort of structure enables the communications necessary to achieve a coherent design. The same rationale applies to the test staff and the project team managers.

The person in each role has primary responsibility for one or more development items. However, they cannot work in isolation; they must communicate with other team members to be effective.

### 3.4.1 Order and Team Communications

One distinguishing characteristic of a development project, as a dynamic system, is that the entities are people, neither machines nor chemical molecules. The emotions and motivations of the team members are among the conditions and variables that interact to govern their behavior. Applying leadership, you affect these variables and steer the project.
Several scientists have considered models of how systems of interacting discrete entities evolve [Kauffman, 1996; Lewin and Regine, 2000]. They have found that the nature of the system is determined by communications among the entities:

- If communications are too restricted, the system will be in a stable equilibrium.
- If communications among the entities are excessive, the system becomes chaotic.
- A middle range is optimal. With just enough communications, the system is at the edge of chaos and evolves stable, manageable structures.

The following observations about communications are directly applicable to project teams, which consist of independent, interacting entities.

- When a team is too restricted in its communications, it does not have the means to adjust to the challenges of a development project. This can happen if the manager insists on being in all of the communication paths.
- If communications are unstructured, the project is likely to become chaotic. This can happen if the manager is uninvolved and the team must organize itself.
- One of the critical management tasks is to develop an organization that enables and promotes the right amount of communications.

This observation reinforces the wisdom of Brooks [Brooks, 1995] and others:

The key to leading successful projects is the right amount of communications.

This idea is entirely consistent with the software development economics model discussed in Chapter 4.

This new common sense provides a final insight: The global behavior of a nonlinear interacting system emerges from the totality of local interactions. It follows that you cannot mandate how your team will behave, but you can influence its behavior by the nature of your communications with team members. Think of this kind of communications as constructive involvement. Being a part of the team
THE SOFTWARE PROJECT / Chapter 3

allows you to provide the necessary steering to keep the team focused on delivering the right product and to influence overall team performance.

Chapter 6 discusses some mechanisms for enabling the correct amount of communications and participating constructively in team efforts.

3.5 THE PROJECT PLAN

The automobile thought experiment is applicable in several ways to software development. In the thought experiment, the goal is to have the car arrive on a target; in software development, the goal is to have your team deliver a solution to the development problem. The initial conditions in the auto experiment are the setting of the steering wheel and the amount of gas in the tank; in the development problem, the initial conditions are the staff on board and their understanding of the requirements.

Note that a classic project plan pictured in project management books and supported by project planning tools is a linear object. It breaks the work into small pieces and adds them together to determine the effort and schedule. It does not account for the nonlinear nature of the interactions among the team members. As linear objects, these project plans are only approximations of what will take place. The plans then are useful only if they are not taken too seriously. As with many linear objects, a project plan can serve as a useful approximation.

Project plans are also predictions of the future. They contain information on what each of the staff members will be doing when solving the problem. Insights from chaos theory tell us that the information contained in a real-life project plan is unknowable. Insisting that a team hold to a frozen plan ignores the fundamental nonlinearity of the process. In fact, you can be sure that the initial plan is almost certainly not accurate. This does not mean you should not have a plan, but only that you must refine and update the plan continually throughout the development.

This process of refinement and updating enables the plan to serve as a steering mechanism. This requires the plan’s details to be filled in as the project evolves. The team’s experience in a given phase serves as input to the details of the plan for the next phase. This iterative development of the project plan is one of the features of the Rational Unified Process, introduced in Chapter 5.

3.5.1 Less Is More

Some managers and staff process engineers do not understand the nonlinear nature of their projects. They believe that the reason project plans aren’t followed
is that they are not sufficiently detailed. They believe that adding granularity to a project plan creates order. (Adding *granularity* means adding tasks of a shorter duration.) For example, the manager may respond to an identified shortcoming in a previous project by adding more work items. Of course, more detail only makes things worse. Adding details takes the leap of faith that one can predict the future with great fidelity.

I have seen six-month projects with 1,800 work items. Based on 180 working hours in a month, each item, on average, would be less than two hours in duration. Anyone trying to administer such a plan would spend considerable time just updating status. Each developer from hour to hour would have to report which of the work items was being addressed. Fortunately, most organizations wisely ignore managers or process groups that try to impose such plans. The downside is that the projects are left with no plan at all. Ironically, the attempt to impose order results in chaos.

### 3.6 APPROACHING DEVELOPMENT RISK

One way that people think about development projects is an exercise in risk management. Some consultants on risk management recommend that the team identify all of the possible risks, usually through some sort of brainstorming session, and then ensure the risks are mitigated. While understanding and addressing the risks of your project is important, it is best to have a more structured way of thinking about risks in your projects.

People can only think about seven things at a time [Miller, 1956]. Given the complexity of software development, it is important to focus on the right things. A useful way to identify and track the issues in managing a software project is to track a small number of development risks.

The literature on software risk analysis is extensive and typically overly elaborate: lists of hundreds of possible risks; bureaucratic procedures that consist of risk management plans, brainstorming sessions with risk identification, and periodic assessments. Most of this is wasted effort. In the spirit of keeping things simple, you should focus on the three project risks:

- **Schedule risk:** Not delivering the project on time
- **Cost risk:** Exceeding the budget before you deliver the project
- **Quality risk:** Delivering software that fails to meet stakeholder needs
Other so-called risk items such as staffing risk (the inability to staff) or technical risk (uncertainty about how to design the code) put the project at risk only to the extent that they affect schedule, cost, or quality. For example, the inability to staff is only a problem if it will put the schedule at risk.

### 3.6.1 Schedule Risk

The product development perspective addresses schedule risk in three ways:

1. It provides insight into the progress of the software development. By tracking the progress of the actual product and not trying to gauge completion of the activity, the manager can determine the true schedule variances. This accurate and timely insight enables the manager to address schedule risk.

2. It provides ongoing integrations and prototypes. Because this approach supports evolving versions of the software rather than integration at the end, technical risk is distributed throughout the effort rather than being stacked at the end.

3. It provides phases with planned milestones. Planned milestones provide momentum for completion of the effort. The defined phases with specified completion criteria allow the manager to keep the team focused on the dates. The phases help keep the project from spiraling out of control.

### 3.6.2 Cost Risk

The attributes of the product development perspective that help address schedule risk also apply to cost risk. In addition, the manager can prioritize content throughout the development. Iterations are planned so that the most essential features are addressed early. As development progresses, the manager can drop unessential features as necessary to hold to the budget, based on cost/benefit tradeoffs. In contrast, the systems engineering approach does not provide the mechanisms for making such tradeoffs once the specifications are written.

### 3.6.3 Quality Risk

The product development method addresses quality risk in several ways. The underlying approach is to find a solution to the development that satisfies the
product needs. The manager can keep the team focused on the system architecture as the solution evolves. Finally, with good schedule and cost management, the team will not be scrambling to patch together a solution at the end of the project. They will have time to address the quality attributes.

The manager can review the software with stakeholders at each iteration of the evolving software. Necessary changes and tradeoffs can be made throughout the project. The product development approach does not rely on perfect knowledge and understanding at the beginning of the project.

3.7 A WORD OF CAUTION: NO SILVER BULLETS

The product approach may be a good framework for understanding system development, but it is not a silver bullet. Its application takes a significant investment from software managers: You must understand how to apply the approach in a detailed, disciplined way and lead your organization in adopting it. Fortunately, there is a detailed software methodology, the Unified Process, that provides the activities, phase specifications, and milestones needed to apply the product development approach to software. The Unified Process is discussed in Chapter 5.

There is a second type of investment for software managers: The success of the product development approach requires the ongoing involvement of all levels of management. This is a good thing. It is addressed in Chapter 6.

To Learn More

Here are some excellent references on product development. I most highly recommend the first reference.

For over 20 years, Prof. Barry Boehm and his students at the University of Southern California have been studying software development productivity. The results of this effort may be found in


Many who develop software believe that the chaotic nature of the field is due to its immaturity. These texts show the dynamics are much the same for more established fields.


Fred Brooks was the first author to document the nonlinear nature of software projects in this classic:


This is one of the first books to explore the management implications of object-oriented development. It has many good insights.


There have been several books for the layman on the nature of nonlinear systems. Three of my favorites are


One of the best books on the application of nonlinear dynamics to management is this recent text.

This is the classic paper on the limits of people’s ability to process information. You will be able to find it online as well.

- Miller, George A. “The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information,” *Psychological Review*, 63, 81–97, 1956.

At this writing, a summary of the history of scientific management may be found at

Book Review

*Software Leadership: A Guide to Successful Software Development*

by Murray Cantor

Addison-Wesley, 2001
ISBN: 0-201-70044-1
Cover Price: US$29.99
(224 Pages)

The *Dilbert* cartoon strip, by Scott Adams, features an amusing but hopelessly stupid character called the "Pointy-Haired Boss," who manages a department in a high-tech company. This gentleman's main characteristic, besides his hairdo, is that he seems to have absolutely no clue whatsoever as to what it is that his subordinates -- including the introverted Dilbert, high-strung Alice, self-centered Wally, and earnest Asok -- are doing and what motivates them. Funny, outrageous, completely out of proportion? I'm not so sure. Our industry is actually full of Pointy-Haired Bosses, with or without the hairdo; people who have been drawn, sometimes not of their own will, kicking and screaming, into managing IT (Information Technology) projects or software development projects.

What about you? Are you a Pointy-Haired Boss? Think again. Look in a mirror. You run the risk of finding a Dilbert cartoon taped on your door some day if you think that:

- Software development is primarily about programming.
- Managing a software project is no different than managing any other project.
- The best approach is to establish a detailed plan right away and make sure everyone abides by it on a daily basis.
- A safe way to proceed with the product is to freeze all requirements early in order to minimize any risk of schedule slippage and late surprises.

As Murray Cantor explains in this book, to really *manage*, as opposed to simply *administer*, good managers must understand the *nature* of what their team is producing, the dynamics of the team, the tools and processes
they use, and all the risks associated with development -- i.e., anything that can hinder success. They must have some software domain expertise. It happens that the very *nature of software* makes it somewhat different from other industrial endeavors, and simply transposing management techniques that work for, say, an engine or a bridge, will fail, or at least fail to deliver a quality product, in the software domain.

Software differs from other products in several ways.

- It does not follow the "laws of physics." And the equivalent fundamental laws of software have not yet been discovered. This makes it difficult to accurately predict and plan without some form of experimentation or prototyping from which to get feedback.

- Software is essentially "soft," that is, modifiable, and your external environment (customers and users) will expect your software to change, adapt, and evolve -- even as you develop it. Your management approach must accommodate for this.

- The techniques and tools used to design and build software evolve rapidly. These techniques and tools have immense impact on software productivity, costs, risks, and quality. You must integrate them into your planning process to remain competitive.

- Software development costs are almost entirely engineering related. Since a software product can be "mass-produced" at very low cost, the economic model is very different from that of other manufactured goods.

These and other specificities of software make it more challenging for managers getting into software territory.

This book will help you *avoid* becoming a Pointy-Haired Boss. It will give you enough insight into the nature of software, the development process, and the techniques and tools used, so that you can intelligently reason about the development, adjust, plan, re-plan, manage risks, and lead a team to success -- instead of documenting and recording its failure.

I am delighted that Cantor, my colleague at Rational, has chosen to base his new book on the Rational Unified Process®. This will make it the book of choice for managers of teams using, or about to use, the Unified Process in any of its variations, and who might be intimidated by the large amount of very technical material these processes contain. It will give them just enough knowledge about the Unified Process to keep their "Dilberts" motivated, productive, and successful. But whatever process you ultimately decide to follow, you will find valuable guidance here for leading your team to success.

*Philippe Kruchten*
Rational Fellow

---

Read a chapter from Murray Cantor's new book, *Software Leadership: A*
UML Activity Diagrams: Detailing User Interface Navigation

by Ben Lieberman, Ph.D.
Senior Software Architect
Blueprint Technologies

This is the third and final article in my series for The Rational Edge on using Unified Modeling Language (UML) Activity Diagrams. In the first and second articles, I showed how Activity Diagrams could be used effectively to capture use-case flows of execution and also to detail the dynamic aspects of the Process View. In both cases, this involves altering and enhancing the basic semantics of new stereotypes for activities. These stereotypes better capture details of the problem domain for use-case execution flows and the dynamic nature of system processing for the Process View. In this article, which illustrates how to use UML Activity Diagrams to capture and communicate the details of user interface navigation and functionality, I reintroduce three of these stereotypes: presentation, exception, and connector. The focus is on how to apply them to the functionality available to system users via a graphical user interface.

Most modern software is designed to interact, at one time or another, with sapient, bipedal life forms (aka, users). To facilitate communication between computer systems and these creatures, software developers invented graphical user interfaces (GUIs). These interfaces are often based on some form of "windowed" environment populated with presentation elements that represent the current system state, controls for altering the current system state, and navigation to shift the focus of attention from one section of the system to another.

The visual elements may be windows, dialog boxes (modal and modeless), frames, bars, blocks and panes, etc. Controls take the form of input boxes, sliders, dials, buttons, selection lists, grids, check boxes, radio buttons, etc. Navigation, made even more important with the advent of Web-based displays, takes the form of directional buttons (forward, back, sideways), menus, links, hotspots, hyperlinks, and more. This bewildering combination
Capturing Visual, Control, and Navigation Elements

For most system modelers, the set of tools available to accomplish this complex task is somewhat limited, consisting mostly of flowchart semantics and screen shots of the interface. Although flowcharts are effective for capturing the dynamic aspects of an interface, they lack the cohesive capability inherent in UML -- to translate and connect concepts from one model to another (e.g., sequence diagrams and class diagrams). Moreover, static screen shots cannot illustrate the sequence of events that occur during a dynamic user "conversation" with the system. Screen shots show what you can do, but not how you can do it, and they cannot illustrate the effect on system state regarding a specific user action.

UML Activity Diagrams are well suited to capturing a series of actions taken on behalf of a user via a GUI. Although on the surface Activity Diagrams may seem little more than glorified flowcharts (and perhaps not even as powerful because they lack specific symbology), in fact they represent the same highly adaptable, extensible, and specific modeling structure evidenced in the remainder of the UML specification. With the addition of several well-chosen stereotypes (part of UML's extension mechanism, along with constraints and tagged values), Activity Diagrams represent a strong addition to the user interface modeler's tool kit. As will be illustrated in detail below, Activity Diagrams provide the ability to accurately depict the dynamic aspect of virtually all current user interfaces.

Figure 1 shows an overview of Activity Diagram elements and icons. See the Sidebar for a description of the Activity Diagram icons and stereotypes we will use in our model.

Activity Diagram Icons and Stereotypes

In constructing Activity Diagrams, it is helpful to use colored icons (and UML stereotypes) to indicate specific activities and visually differentiate various steps in a flow. (See Figure 1.) This is particularly important for connector icons (pointers to additional diagrams) that link to separate use-case scenarios.*

Action is the primary diagram element. This icon represents activities performed by the System or Actor. Since it is the most common icon, it typically has either a neutral color or no color at all.

Presentation activities are indicated by the &lt;&lt;presentation&gt;&gt; stereotype. This stereotype indicates that there is a conversation between the use-case Actor and the System. It represents a special category of Action activities and is used to abstract user interface details.

he Connector stereotype represents connections to flows diagrammed elsewhere. The use of Activity Diagrams often leads to the creation of
large and complex models, so it is useful to indicate alternate navigational flows. The connector icons for this stereotype can be used to automatically link to another diagram (e.g., to a separate Rational Rose diagram using an embedded link). If desired, the extension to a Rose (UML) diagram can lead to a "subactivity" diagram embedded within the activity itself. One caution, however: This approach can rapidly produce a very "deep" model with multiple, embedded layers. Such a model runs contrary to the ideal of a high-level "roadmap," which shows an overview and leaves the details for the textual description.

Exception activity occurs when there is an exceptional flow in the use case, and is indicated by the <<exception>> stereotype. This usually represents an error condition but may also represent unusual or unexpected system behavior. If the exception is an error condition, then it is useful to summarize the error inside the icon. The icon is also useful to indicate system logging and recovery activities.

Scott Ambler also recently authored a book chapter that recommends the creation of Interface-Flow Diagrams to illustrate the movement from one "screen" to another; the transition between one system state and another is indicated as the result of a user action (e.g., "click search button," or "select customer"). As we will see below, this approach can be very neatly applied using the guard condition semantics and activity elements from Activity Diagrams.
The Graphical User Interface

As I explained in my two previous articles for The Rational Edge, Activity Diagrams give you the ability to construct a "roadmap" of user functionality that accurately represents the paths a user can follow, but they do not discuss why or when a particular path should be followed. This information is best captured in the use-case textual description, which allows you to express the required level of detail. Similarly, process flows are paths that the system can take, but the detail on when and why is found only in the Software Architecture Document artifact of the Rational Unified Process®, or in similar system description documentation.

This article presents a technique for modeling three common but distinct types of GUIs: a Web site; a guided interface (e.g., "wizard"); and a standard application interface. For each type of interface, the details of the user experience -- including branching, selection, navigation, and constraints (enabled/disabled menus or controls) -- will be captured in the model.

As with all modeling techniques, the task begins by segmenting the modeling domain into sections that can be organized and managed. For this purpose, we will use the elements we identified above: presentation (manages appearance), controls (manage state), and navigation (manages location).

Modeling a Sample Web Site Interface

As an example of a Web site interface, we will model the Microsoft™ Hotmail® Web site (www.hotmail.com). As shown in Figure 2, the starting point for the Web site is the Entry page, which consists of two frames: a Language Selection frame and a Login Information frame. The term frame indicates that the functionality is logically separated, but not that it is necessarily implemented using HTML-based frame tags (although it may very well be, in which case the designation is exact). As shown in the figure, users can select a different display language (all of which are indicated), and the system will then present them with the same interface branded to the selected language.

Users may also select from a collection of hypertext links that will take them to different pages or sites (note: to save space, I will not include a full description of these sites in the example; a document should accompany the diagrams to indicate the transfer location -- whether it is internal or external to the modeled site).

The user login information consists of the Sign-In name, the location (MSN.com or Hotmail.com), the password, and the security setting (high,
After successfully completing the login, the user is taken to the main page for the Hotmail system (Figure 3). From here, the user is presented with either the Hotmail homepage if no new messages have been received, or the user's InBox if new messages have arrived. The InBox offers numerous options for navigation and state control. The navigation bar, which offers options for composing messages, the address book, calendar, etc., is available from all pages within the site. The InBox frame allows the user to change the state of the display and sort messages according to From, Subject, Date, and Size. Messages may be marked for deletion, blocking, or movement to other designated folders.

By selecting a message, the user can bring up the Display Message page and access additional navigation options for his messages, such as Next, Previous, and Delete. If the user clicks on Reply, Reply All, or Forward, he is taken to the Compose Message page; the requested addresses are already displayed in the message header (shown here as a <<connector>> activity). Finally, the user may chose to add the sender to his Address Book or to block further messages from the sender.

In Figure 3, notice the use of frames to segment functionality and the placement of links into notes. Often a Web site makes many hyperlinks...
available to users (advertisements, off-site links, etc.). Usually, it is only necessary to diagram those links that will take users to another section of the current site (Navigation) or change the state of the display (Controls). The guiding principle here should be to "tell a story" with your diagrams. Too much information may confuse the reader and make the model less useful. Frames are a good way to segment the functionality presented by complex sites (such as the Hotmail site). Separating concerns visually (i.e., Navigation and Control) will facilitate modeling of the site. Note, however, that <<connector>> rather than <<frame>> is used for the off-page links. This is done deliberately to show that these links lead to other diagrams or some other change of user focus. The <<frame>> stereotype, as used in this example, indicates the current focus of user attention: the InBox and associated messages.

Figure 3: Hotmail Inbox and Message Display

As an aside, an interesting extension to this approach might be to use a new generic stereotype called <<navigate>> to indicate a toolbar, drop-down, or other navigation-based page element that had not been fully specified (during site construction).

Modeling a Guided Interface

To illustrate the modeling of guided interfaces (e.g., "wizards") we will use the Print Wizard in Windows 2000,™ which assists users in creating new printer ports and installing print driver files. As shown in Figure 4, the Print Wizard is mostly a series of dialogs that lead the user from one step to the next. The functional flow is as follows: First identify the printer type that
will be used (networked vs. local); then select the printer port (or network location), the printer driver, and printer identifier; finally, generate a test print page. As Figure 4 shows, there are several branches in the flow (e.g., the creation of a new port, shown by the Port Wizard <<connector>> icon) as well as data elements that are either optional or mandatory at each step. If the user does not provide a mandatory data element (such as Share Name), then an exception will occur and an error message will be displayed. When the user follows all the steps correctly, then the printer connection will be established and the Print Wizard will end.

In this model, the note at the beginning indicates that the user has the option at any step to return to the previous step (back) or to cancel the operation (cancel). Also, there is only one exit point from this wizard, although some guided interfaces provide the option to end the sequence early (such as the Internet Connection Wizard). In this case, the guard [finish] condition may occur in earlier steps and lead to a different exit state. Also, this series of steps allows the user at one point to go online to select a driver update from the Microsoft™ printer Web site; this is indicated by a <<connector>> activity and may or may not be modeled elsewhere. In any case, the resulting information (here the updated driver files) is indicated. As would be expected from a directed interface, navigation is limited, and there are very few explicit backward connections (we already noted the ability to back up one step at a time) or other looping connections.
Modeling an Application Interface

Our final example will be to study one aspect of a standard application and capture the functionality and navigation available to the user in a model. For this example we will explore the Rational RequisitePro® user interface. The RequisitePro application opens with a display of the main navigation menu (which is always the top-most window). Figure 5 shows the functionality available from the Main Dialog project menu. The other menu choices (not shown) can be captured using the same mechanism. As shown in the figure, there are five possible selections from the project menu when no project is available: New, Open, Archive (ClearCase and RequisitePro), and Exit. By selecting New, the user opens the "Create Project" dialog. Selecting Open brings up the "Open Projects and Documents" dialog and offers two frames: one for projects, one for documents. With the projects frame, the user can open the properties dialog that will allow validation of the project, setup of the ODBC database, and presentation of account information. Once a project is open, then the Main Dialog is altered to allow for full functionality.

Not shown in this example is the security check dialog, or the remainder of the program functionality. The remaining system functionality, however, can be modeled in much the same way I described above. Buttons, dials, sliders, switches, and other controls can be shown by the changes they produce in system state. Dialogs, windows, frames, boxes, and other presentation mechanisms can be illustrated in the same manner as the initial project set-up. Finally, navigation around the system via menu selections, buttons, or other navigation mechanisms can be captured as transitions between screens and system states.
Activity Diagrams: A Powerful Tool for System Architects

This article has presented a technique for capturing the navigation of various user interfaces, including Web sites, assisted functionality (e.g., wizards), and standard applications. In each case, Activity Diagrams can illustrate possible navigation paths through the interface and connections to other parts of the system functionality. This technique provides a clean way to model and represent possible paths through the interface and key data elements required for state changes. For organizational purposes, it is useful to segment the system functionality by presentation, controls, and navigation elements of the interface.

For each type of interface, presentation elements are the windows, dialogs, and other visual elements of the interface. Controls are shown by their effect on the system (e.g., sorted messages), and navigation is represented as transitions between activities.

Like the other two articles in this series, this one illustrates the power and value of Activity Diagrams for communicating system functionality,
processing, and user interface flows. For the system architect, these Diagrams expand the tools available for the expression and modeling of complex software systems.

Notes


5 Hotmail is a registered trademark of Microsoft, Inc. The functionality modeled here is subject to change, as Microsoft alters its Web site; it should not be taken as a representation of all available services.

*For a Rational Rose® script to automate the application of color to activity model elements, see the Appendix below.

Appendix: Rational Rose Activity Diagram "Colorizer" Script

This Rational Rose® script (for version 2000 and greater) will automatically add diagram element fill color to the Activity Views on each Activity Diagram included in the use-case model (e.g., under the Logical View root package).

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 2001 | Privacy/Legal Information
Layering Strategies

by Peter Eeles
Rational Services Organization, UK

For system architects, all techniques for decomposing software systems address two main concerns:

1. Most systems are too complex to comprehend in their entirety.
2. Different audiences require different perspectives of a system.

Layering is a decomposition technique that has been adopted in numerous software systems and espoused in many texts as well as the Rational Unified Process® (RUP®). It is, however, often misunderstood and incorrectly applied. This article clarifies what is meant by layering, and discusses the impact of applying different layering strategies.

What Is "Layering"?

The term layer refers to the application of an architectural pattern generally known as the Layers Pattern, which is described in a number of texts and also in the RUP. A pattern represents a solution to a common problem that exists in a particular context. Table 1 provides an overview of the Layers Pattern.

Table 1: Overview of the Layers Pattern
<table>
<thead>
<tr>
<th>Layers Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Context</strong></td>
</tr>
<tr>
<td>A system that requires decomposition.</td>
</tr>
<tr>
<td><strong>Problem</strong></td>
</tr>
<tr>
<td>A system that is too complex to comprehend in its entirety.</td>
</tr>
<tr>
<td>A system that is difficult to maintain.</td>
</tr>
<tr>
<td>A system whose least stable elements are not isolated.</td>
</tr>
<tr>
<td>A system whose most reusable elements are difficult to identify.</td>
</tr>
<tr>
<td>A system that is to be built by different teams, possibly with different skills.</td>
</tr>
<tr>
<td><strong>Solution</strong></td>
</tr>
<tr>
<td>Structure the system into layers.</td>
</tr>
</tbody>
</table>

**Layering Basics**

One of the most familiar models for layering is the OSI 7 Layer Model (Figure 1), defined by the International Organization for Standardization (ISO). This model defines a set of networking protocols: Each layer focuses on a specific aspect of communication and builds upon the facilities of the layer below it. The model also uses a responsibility-based layering strategy that associates each layer with a particular responsibility. Layering strategies can be based on other system characteristics as well, such as reuse. We will explore both responsibility-based and reuse-based strategies in more detail later.

![Figure 1: OSI 7 Layer Model (Responsibility-Based Layering)](image)

First, however, let's look at Figure 2, which shows another, simple model for responsibility-based layering. Note that:

- The *presentation logic layer* contains elements responsible for providing some form of communication with a human being, such as an element in the user interface.
- The *business logic layer* contains elements responsible for performing some kind of business processing and the application of business rules.
- The *data access logic layer* contains elements responsible for providing access to an information source, such as a relational database.
Layers can be modeled in a number of ways, as we will see later on. Here, we represent a layer using a UML (Unified Modeling Language) Package with the stereotype <<layer>>.

![Layer Model](image)

Figure 2: A Simple Model for Responsibility-Based Layering

The layers in this kind of model are often called tiers, a familiar concept in distributed systems development, which deals with 2-tier, 3-tier, and n-tier systems.

An important aspect of the model in Figure 2 is the direction of dependencies. The dependencies shown imply a certain rule that is a characteristic of layered systems: An element in a particular layer may only access elements in that same layer or layers below it. In the example shown here, elements in the Business Logic layer may not access elements in the Presentation Logic layer. Also, elements in the Data Access Logic layer may not access elements in the Business Logic layer. This structure is often referred to as a directed acyclic graph (DAG). It is directed in that the dependencies are unidirectional, and acyclic in that the path of dependencies is never circular.

When defining a layering strategy, it is important to be precise about the meaning of each layer so that system architects and designers will correctly place elements in the appropriate layer. Incorrect placement will diminish the value of applying the strategy in the first place. As we discuss layering strategies in more detail below, we will also provide general guidance on what each layer means.

**Modeling Layers**

For an architect, communicating any layering strategy effectively requires a set of specific models (and UML elements). A model represents a complete description of a system from a particular perspective. Figure 3 shows an example of four models that represent different perspectives of the system under consideration:

- *Use-Case Model*: captures system requirements.
- *Analysis Model*: captures system requirements analysis.
- *Design Model*: captures system design.
● Implementation Model: captures system implementation.

![Figure 3: Four Models Representing Gradual System Refinement](image)

Additional models include:

- Deployment Model: captures the distribution aspects of a system.
- Data Model: captures the persistent aspects of a system.

**Layering Strategies**

Now that we have a basic understanding of what layering is and what is required to represent it, let's look at two specific layering strategies in more detail. As we saw above, layering can be based on a number of characteristics, including responsibility and reuse. This section discusses strategies based on these characteristics, as well as ways to depict them.

**Responsibility-Based Layering**

Responsibility-based layering is probably the most commonly used layering strategy. Because it isolates various system responsibilities from one another, this strategy can improve both system development and maintenance. As we saw in Figure 2, a system can be layered based upon the following responsibilities:

- Presentation logic
- Business logic
- Data access logic

Each of these responsibilities can be represented as a layer, as shown in Figure 4, which also shows sample content for each layer. Here we consider three concepts in an order processing system: Customer, Order, and Product. As an example, the Customer concept comprises the:

- CustomerView class: responsible for the presentation logic associated with a customer, such as the rendering of a customer in the user interface.
- Customer class: responsible for the business logic associated with a customer, such as validation of customer details.
- CustomerData class: responsible for the data access logic associated with a customer, such as making the state of a customer
Despite its widespread use, this layering strategy has generated some myths that either discourage people from using it or confuse those who do. Let's look at some of these myths and attempt to clear up the confusion.

**Myth 1: Layers and Tiers Are Different**

This particular "myth" is a frequent source of confusion. In fact, a tier is a layer, albeit a layer associated with a particular strategy: responsibility-based layering. Certainly, the numerous ways in which tiers are organized, as shown in Table 2, may cause some confusion when attempting to understand the relationship between tiers and layers.

<table>
<thead>
<tr>
<th>Application</th>
<th>Layers (Tiers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-tier</td>
<td>combined presentation logic and business logic</td>
</tr>
<tr>
<td></td>
<td>data access logic</td>
</tr>
<tr>
<td>3-tier</td>
<td>presentation logic</td>
</tr>
<tr>
<td></td>
<td>business logic</td>
</tr>
<tr>
<td></td>
<td>data access logic</td>
</tr>
<tr>
<td>n-tier</td>
<td>presentation logic (distributed)</td>
</tr>
<tr>
<td></td>
<td>business logic</td>
</tr>
<tr>
<td></td>
<td>data access logic</td>
</tr>
</tbody>
</table>

**Myth 2: Layers (Tiers) Imply a Physical Distribution**

Another common misconception is that the logical layering implies a physical distribution. Consider a 3-tier layering. Various elements will reside in a particular layer, but each layer can itself be applied in a number of ways, as shown in Table 3. Note that the names in the left column (e.g., "thin client") are those commonly used to characterize a
particular physical distribution.

Table 3: Application of 3-Tier Layering

<table>
<thead>
<tr>
<th>Application</th>
<th>Layers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Client side</td>
<td>Server side</td>
</tr>
<tr>
<td>Single system</td>
<td>presentation logic</td>
<td>business logic</td>
</tr>
<tr>
<td></td>
<td>business logic</td>
<td>data access logic</td>
</tr>
<tr>
<td>Thin client</td>
<td>presentation logic</td>
<td>data access logic</td>
</tr>
<tr>
<td></td>
<td>business logic</td>
<td></td>
</tr>
<tr>
<td>Fat client</td>
<td>presentation logic</td>
<td></td>
</tr>
<tr>
<td></td>
<td>business logic</td>
<td></td>
</tr>
</tbody>
</table>

It is also true that a single system may employ more than one physical distribution strategy; certain elements would characterize a "thin client" distribution, while others would represent a "fat client" distribution. The choice is typically made based upon nonfunctional requirements such as performance.

Modeling Responsibility-Based Layers

Responsibility-based layering typically affects the design model, implementation model, and deployment model. The design model is typically structured using one of two approaches.

The first approach shows the elements "contained" within the layer. The result is implied in Figure 5, a Rational Rose® browser screenshot, which shows:

- Presentation classes (CustomerView, OrderView, and ProductView) residing within a Presentation Logic package.
- Business logic classes (Customer, Order, and Product) residing within a Business Logic package.
- Data access logic classes (CustomerData, OrderData, and ProductData) residing within a Data Access Logic package.
The second approach focuses on the business component (in this case, Customer, Order, and Product); the primary elements of concern are the domain-related concepts supported by the system. The concept Customer, for example, may have associated elements of presentation logic, business logic, and data access logic. This way of thinking results in the model structure shown in Figure 6. In this example, the layering is implied by the element names. For example, all View classes (such as CustomerView) imply a presentation logic layer, and all Data classes (such as CustomerData) imply a data access logic layer. Unqualified class names (such as Customer) imply a business logic layer.
The layering could also be represented explicitly within each package representing a business component, as shown in Figure 7. This structuring is preferable when a number of elements are involved in each layer of a given business component. Although only the Customer business component package has been expanded in this example, the Order and Product packages would have a similar structure.

![Figure 7: Rational Rose Showing Explicit Layering Within a Business Component Package of a Design Model](image)

A responsibility-based layering strategy typically affects the implementation model as well as the design model, when there is a need to physically partition the elements that implement each responsibility. For example, consider a system with a "thin client" physical distribution: It is useful to identify the implementation units required to support execution on the client and server, respectively. In this example, the elements in the presentation logic layer reside in an application that is deployed on a client, and all elements in the business logic layer and data logic layer reside in another application, which is deployed on a server.

This scenario implies an implementation model like the one shown in Figure 8. Here, a Rational Rose browser image and a component diagram show the elements of the application deployed on the client. There happens to be a one-to-one mapping between a class in the design model and a UML component in the implementation model. Note, however, that mapping typically depends on the implementation technology used.
Figure 8: Rational Rose and Component Diagram Showing Implicit Layering in an Implementation Model

Similarly, a responsibility-based layering strategy affects the deployment model when there is a need to describe the physical distribution of the responsibilities. Referring to Figure 9 and the example above, we can see that six nodes have been defined. Each of the three Client nodes houses a ClientApplication process. The FrontEndServer node houses a LoadBalancer process that is responsible for distributing client requests to one of two Server nodes. Each Server node houses a ServerApplication process.

Figure 9: Rational Rose and Diagram Describing Physical Distribution of Responsibilities in a Deployment Model

Reuse-Based Layering

Another commonly used layering strategy, based on reuse, is particularly relevant to organizations that have an identifiable goal to reuse components. This strategy explicitly groups components according to their level of reuse, and makes their degree of reusability highly visible.

The example in Figure 10 shows three layers:
- The Base layer contains elements (e.g., Math) that might apply across organizations. Such elements will be reused widely.

- The Business-Specific layer contains elements that apply to a particular organization but are application-independent (e.g., Address Book). Such elements will be reused in applications within the same organization.

- The Application-Specific layer contains elements that apply to a particular application, or project (such as Personal Organizer). These elements are the least reusable.

![Figure 10: Example of Reuse-Based Layering](image)

**Modeling Reuse-Based Layers**

A reuse-based strategy primarily influences the design model. The structure of a design model that incorporates reuse-based layering is easy to depict, as shown in Figure 11 (which reflects the example in Figure 10).

![Figure 11: Rational Rose Design Model Incorporating Reuse-Based Layering](image)
Multidimensional Layering

This article has focused on the two most widely used layering strategies, but other related strategies might organize layering by characteristics such as security, ownership, and skill sets. In addition, the strategies we have examined can also be combined to create new layering strategies. Figure 12, for example, includes two reuse-based layers from the previous example (application-specific and business-specific) and three responsibility-based layers or tiers (presentation logic, business logic, and data access logic).

The dependencies in a reuse-based layering strategy typically result from dependencies between elements in business logic layers, as implied in Figure 12, which shows dependency between PersonalOrganizer and AddressBook.

Modeling Multidimensional Layers

In Figure 13 we can see the multidimensional aspects of layering within a two-dimensional design model, which also incorporates the business component concept.
Before you can adopt a multidimensional layering strategy, you must identify a primary strategy; ours is based upon reuse in this example. The design model is first organized based on this strategy, yielding the layers Application-Specific, Business-Specific, and Base. Each layer is then further organized by the elements that reside on each. Figure 13 shows that the Business-Specific layer contains Address Book and Calculator, for example. Each of these elements is then further organized based upon a secondary strategy: responsibility-based layering. For example, the Address Book package contains the three layers Presentation Logic, Business Logic, and Data Access Logic. Each of these layers, in turn, contains any elements that reside on that layer: the Presentation Logic layer contains the AddressBookView class; the Business Logic layer contains the AddressBook class; and the Data Access Logic layer contains the AddressBookData class.

**An Important Choice**

One of the most important decisions an architect must make is choosing an appropriate layering strategy, since that strategy will have a major influence on the structure of the system models. Even more important, the right layering strategy can directly support significant business benefits such as maintainability and reuse. If the architect chooses a responsibility-based layering strategy, for example, and isolates the different responsibilities of the system from one another, then the system the project team develops is likely to be more easily maintainable than it would be otherwise. If the architect adopts a reuse-based layering strategy, then reusable system elements can be clearly identified.

**Acknowledgments**

The author would like to acknowledge the contributions of Kelli Houston, Wojtek Kozaczynski, Philippe Kruchten, Bran Selic, and Catherine Southwood (all of Rational Software) for their insightful comments on early drafts of this paper.

**References**


Notes


2 The abbreviation ISO would suggest a name such as *International Standards Organization*, but, in fact, ISO is not an acronym. The following text is taken from the ISO Web site: "ISO is a word, derived from the Greek *isos*, meaning equal, which is the root of the prefix *iso-* that occurs in a host of terms, such as *isometric* (of equal measure or dimensions) and *isonomy* (equality of laws, or of people before the law). From *equal* to *standard*, the line of thinking that led to the choice of ISO as the name of the organization is easy to follow. In addition, the name ISO is used around the world to denote the organization, thus avoiding the plethora of acronyms resulting from the translation of *International Organization for Standardization* into the different national languages of members, e.g., IOS in English, OIN in French (from *Organisation Internationale de Normalisation*). Whatever the country, the short form of the organization's name is always ISO."

3 Although an event notification may result in a message from an element in one layer being sent to an element in an upper layer, there is no explicit dependency in this direction.


5 Derived from a strategy described in Ivar Jacobson et al., *Software Reuse*. Addison-Wesley, 1997.

For more information on the products or services discussed in this article, please click here and follow the instructions provided.
Thank you!
Forward and Reverse Engineering of Code with Rational Rose and VisualAge for Java

by Shawn Keller
Delivery Leader
WaveBend Solutions

Having the ability to forward engineer, reverse engineer, or roundtrip engineer between design tools and Integrated Development Environments (IDE) offers software engineering teams powerful options in the evolution of component-based development efforts. Rational and IBM have created an effective integration between their design and implementation tools through a connector called the RoseLink Plug-In.

There are three scenarios in which using this integration point is effective:

- **Forward Engineering** -- In an instance where much thought and time has been placed into the design of the model, including modeling detailed subsystems, packaging and class structures, the architect or designer is able to forward engineer the model(s) into Java code having all of the package and class elements and relationships realized. In this case, the model is moved forward to code and the implementer(s) continues on with the generated Java class skeletons, adding business logic and building upon the superstructure designed by the architect.

- **Reverse Engineering** -- Many of us have experienced the situation of being placed on a team or engaged with a new client where the effort has no design, no modeling efforts or no clear understanding of the architecture of an existing system. Perhaps the system was built in haste or a prototype was "band-aided" into a production system. Either way, the role as an architect or implementer can be made easier if an understanding of the underlying architecture is known. By reverse engineering Java packages into Rational Rose®, a visual perspective of the system can be gained.

- **Roundtrip Engineering** -- Having experienced this personally in a rapid development effort, roundtrip engineering allows an architect
to develop the base class structure from a higher level, perhaps not having the time to design all of the properties of the classes. Having built the classes and their relationships to each other, the architect can leave the more detailed element definition to the implementer. Roundtrip engineering can keep both the design model and Java code in synchronization. The architect will forward engineer the model to code, the implementer makes additions or validated changes to the structure and reverse engineers the code back to the same Rose model. The architect can then visualize the implementers' changes in the model. These roundtripping efforts can effectively continue through several iterations until a robust architecture is realized.

**Approach**

**Verify Installation**

Before attempting any type of reverse and/or forward engineering, it is imperative to confirm that the IBM and Rational software is correctly installed. Verify that VisualAge® for Java™ and Rational Rose J/Enterprise are installed.

**Install RoseLink**

The RoseLink Plug-In must be installed in order to create connectivity between the products. It is best to install RoseLink after the Rose and VisualAge for Java products have been installed.

**Enable the RoseLink Plug-In in VisualAge**

- Launch VisualAge for Java.
- Click the File menu and choose "Quick Start."
- Choose "Basic" from the Quick Start menu.
- "RoseLink Plugin Toggle" should be visible in the "Basic" list. If it is not, then the RoseLink installation was not successful.
- Click to highlight "RoseLink Plugin Toggle" and click "OK."
Figure 1: If you have installed the plugin correctly, it will be listed in the right column

- A confirmation window will appear.

Create Project(s) Within VisualAge for Java

It is important to note that the integration that occurs between VisualAge and Rose occurs at a VisualAge project level. As such, you must create a project within VisualAge before creating the connection to VisualAge within Rose. This project will hold the packages and classes that will be generated in a forward engineering effort; likewise, it will contain packages and classes to be modeled in a reverse engineering effort.
Enable RoseLink Connection within Rose

- Launch Rose.
- Click the Tools menu.
- Choose "Java."
- Choose "VisualAge for Java Project."
- Rose may prompt you to reset the Java Virtual Machine to "IBM." By choosing "Yes" to the dialog box, it will do this for you automatically. This is actually a Rose model property that can be set manually.

Connect to a VisualAge for Java Project

- The "VisualAge Link Settings" window appears. Choose the VisualAge for Java project to connect to and click "OK."

![Figure 3: Choose the project to which you'd like to connect](image)

- Rational Rose creates a new Project Model property setting called VAJavaProject. This property is set to the project name you selected. In order to view this and other model settings:
  - Click the Tools menu.
  - Choose "Model Properties."
  - Choose "View" or "Edit."
  - Choose the "Java" tab and change "Type" to "Project."
Forward Engineer

In order to successfully forward engineer from Rose to VisualAge for Java, the correct Java frameworks must be loaded and associated with the Rose model. Having the Java frameworks loaded allows Rose to generate the Java code. (See Building Reusable Architecture using Rational Rose Frameworks -- a whitepaper by Khawar Ahmed about the specifics on implementing frameworks with Rose.) In instances where the final implementation is known, it is best to use frameworks, as this allows the designer to model to the specific implementation using the particular constructs associated with that specific language. By default, Rose is packaged with several frameworks available to the designer. However, other frameworks can be created and added to the design efforts.

- To forward engineer from a class diagram, open a diagram in Rose and click to highlight the classes to be generated. Use CTRL-click to highlight multiple classes.
- Right click to view the shortcut menu.
- Choose "Java."
- Choose "Generate Java." Verification messages will appear if the generation was successful; otherwise the Rose log file will detail errors if generation failure occurs.
- Maximize VisualAge for Java and explore to the target project. Notice that packages (if used in Rose) and classes are created under the target project.
Reverse Engineer

Whether this be reverse engineering or the continuation of a roundtrip engineering effort, any changes made to the connected project within VisualAge for Java can be reflected in Rose as a class diagram. Remember that the integration is based upon a specific VisualAge for Java project. In order to reflect alterations within Rose, all changes to packages, classes, and relationships should be made in that project.

- In Rose, click the Tools menu.
- Choose "Java."
- Choose "Reverse Engineer Java."
Figure 6: Choose "Reverse Engineer"

- The "Java Reverse Engineer" window appears showing the VisualAgeconnected project name and package structure.
- Highlight the files to be reverse engineered in the upper right-hand window and click "Add."
- These files will be added to the lower window.
- Click "Select All."
- Click "Reverse."
- Notice that any new classes, methods, and attributes are reflected in the Rose model.

Figure 7: Add the files you wish to reverse engineer

Assumptions
This effort was based upon the following configuration:
• Windows 2000 Professional (http://microsoft.com/windows/default.asp)
• IBM VisualAgefor Java 3.5 Enterprise with Service Pack 2 applied (http://www-4.ibm.com/software/ad/vajava)
• Rational Rose 2000e (http://www.rational.com/products/rose/index.jsp)

Rose J to VisualAgefor Java link (http://www.rational.com/support/downloadcenter/upgrades/rose.jsp)

*NOTE: This article was originally published on Rational Developer Network, the learning and support channel for the Rational customer community. It is part of a three-part series on the integration between Rational and IBM Technologies that includes "Data Model Implementation with Rational Rose and IBM DB2" and "Integrating Rational ClearCase with VisualAge for Java." Rational Developer Network is currently available only to Rational Suite customers. If you are a Rational Suite 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!
On Politics in Technical Organizations

by Joe Marasco
Senior Vice President
Rational Software

One of the Rational insiders who reviewed my article before it was published in last month's issue took me to task for not sufficiently condemning "politics" as one of the root causes of difficulty in the software construction process. This revealed to me a misunderstanding about the role of politics, not just in software development, but also in the management of technical organizations in general. The purpose of this piece is to explore the many dimensions of politics in technical organizations.

Context

Before launching headlong into the meat of the matter, I need to say a few things about context. Any time we talk about human behavior, such as "politics" and political process, we enter into the minefield of cultural differences. Simply put, different behaviors are more or less acceptable in different cultural contexts. Behavior that could be considered "borderline" in one culture could be considered so offensive in another that it would never happen. When we get into "good politics" and "bad politics" below, let's remember that the norm I am using here is a North American standard, and, even at that, susceptible to many local gradations. Some of the behaviors described might be more or less acceptable in European or Asian contexts. The part of the discussion that I believe transcends national and cultural boundaries is the difference in the way technical people (and the organizations they belong to) view the political process. Here I have observed some common tendencies that are worth noting, independent of geography and culture.

Highly competent technical people often have a distaste for politics and politicians. Although I'm no psychologist, I believe this may stem from their notion that technical matters are "precise" (black and white) whereas politics is "messy" (lots of shades of gray.) By training and education,
scientists and engineers have a very analytic, problem-solving approach, which involves invoking "first principles" and applying data in the course of various inductive and deductive processes. Often, this leads to neat, "closed form" solutions. And because many technical managers come from the ranks of scientists and engineers, they bring along with them not only this very strong toolbox, but also some of the biases that come along with it.

Politics involves a great deal of ambiguity, which makes many technical people nervous. It forces them to play on "someone else's court," as it were, a place where they feel they are at a disadvantage because their technical prowess doesn't count for much.

While this is perhaps an oversimplification, it is fairly safe to say that technical people would prefer issues to be resolved on technical merits alone, or perhaps on some mix of technical and business objectives. What they react to, often violently, is the manipulation they see in various discussions and negotiations -- a manipulation they associate with the darker, uglier side of human behavior -- often characterized by self-aggrandizement and personal agendas. Technical people have a tendency to broad-brush this behavior as "politics," and in so generalizing it, give it a very negative connotation.

On the other hand, regardless of its nature, just wishing "politics" away will not make it go away. There is "the way we would like things to be," and the way things are. Ironically, some people view bridging this gap as itself a political process.

My point is that so long as you have human beings in the loop, you are going to have politics. Human beings tend not to all think alike; in order to resolve differences of opinion, a political process is unavoidable. So, rather than condemn it, it is better to understand politics as an effective means of dealing with the inevitable need to resolve differences of opinion.

**Definition**

Let us note that the word "politics" refers to a category of human activity like "carpentry," "theater," or "surgery." What we really need to define is the term "political process." A political process is one in which you get a person or group of people to do what you want them to do. This is actually a "shorthand" for a much more complicated set of ideas.

A political process is one in which two or more people adopt a single course of action. Typically, each individual involved in a political process maintains an agenda that is different from the group's collective agenda, or from any other individual's personal agenda. It is through the political process that an individual's agenda becomes publicized (if a free press is at work), negotiated and voted on (if a democracy), or implemented without further discussion (if the individual is the king). In every case, the individual engaged in a political process wants the group to adopt his or her agenda.²

By the way, getting other people to do what you want them to do is also
the job of leaders and managers. But you probably have already made that observation yourself.

Note that "what" you are trying to convince them to do can be either "good" or "bad"; that is irrelevant to the discussion. Of course, from your point of view, you are probably trying to get them to "do the right thing," but this certainly depends on your perspective. Even when purely motivated, your objectives may be viewed as "wrong," "bad," "misguided," or even "evil" by some other third party.

**Three Scenarios**

If you are in the category of people who still find politics not their cup of tea, let me give you another way to look at it. There are, in fact, only three distinct cases to consider.

1. **You Are the King**

   You might think that politics is irrelevant in this case. The simplistic view might be that even if you are a benevolent despot, you pretty much get to do what you want, and your vassals obey. If politicians crop up, you suppress them; dissident serfs get thrown in the oubliette³, so you don't have to deal in politics. In a business organization, the equivalent behavior is to fire any employee who shows "political" tendencies.

   But this is an oversimplification. Even kings need to build some support amongst their circle of advisors in order to stay in power. There is a good reason today why we don't have very many absolute, powerful kings left. But, if you are the king and want complete control over your realm, you can keep politics at bay for a while. In today's world, it is just not a really good, viable, long-term strategy.

2. **You Are Not the King**

   Now you have two choices: one, convince the king to do what you want to do, or, two, band together with other vassals to either reason with the king or overthrow him. I will point out that both alternatives imply a political process. If you are unclear on this concept, go back to our earlier definition of a political process above.

3. **There Is No King**

   This often occurs in organizations where there is no strong leadership. You might even call this nascent democracy. In this instance, people have to get together and decide what to do, as there is no king telling them what to do. People will tend to have differing opinions, so we once again get to decision making by consensus building (also a political process).

   Whether or not you consider technical organizations "democracies" (and we all know managers who strongly believe they are not!), it is important to understand that we exist in a culture that encourages participation of all its members. In fact, the strongest technical organizations are the ones that engage the talents of all, in a free marketplace of ideas. Ideally, one
wants to create a high-trust environment, that is, an environment where there is a high degree of trust among employees at every level, and where intellectual honesty reigns and politics is minimized. But we've just shown that politics is inevitable in all organizations, and technical organizations are just another form of organization. How do we deal with this apparent paradox?

Politics Is Inevitable, But...

In almost every organization, there is going to be some degree of politics. In the House of Representatives and the Senate, political context and the bartering that accompanies it are fundamental. But as we have pointed out above, when it comes to technical organizations, most would like to minimize the political context. Why?

The answer is simple. You cannot solve technical problems by political compromise. If there are two sides to an issue, one of which is technically feasible and one of which is technically ridiculous, no political compromise will work. Imagine an absurd example: The "solidistas" want dams to be solid, while the "holeistas" want dams to look like Swiss cheese. The political compromise of having only one or two holes in the dam would make no sense whatsoever.

Most people quickly understand the issues implicit in the last paragraph, but not always. I have seen, on occasion, otherwise intelligent people falling into the trap of trying to mediate between competent and incompetent people, giving equal weight to their ideas. This never works, annoys both parties, and tends to really discourage the competent people. Another way of saying this is that "Everyone is entitled to an opinion," but technical people know that not all opinions are equally well qualified.

Another, more subtle variant of the political pitfall for technical organizations is bartering. In many political processes, two sides get to trade on issues: You concede on this one and I'll concede on that one. We each "win one," and we each "lose one." This is the most frequent way of avoiding deadlocks.

Well, in technical organizations this won't work in many contexts. If there are two dams to be constructed, to reuse the example above, then again it makes no sense to build dam "A" according to the "solidista" philosophy and dam "B" according to the "holeista" philosophy. One will hold water and the other won't. Period.

Nor does it work to give the wrong-headed side free rein in some other area. If the political compromise to achieve solid dams is to let the incompetent "holeistas" go off and design another class of structure -- say fortresses -- then we have once again made a really bad decision through the use of "politics."

Of course, the choices in real life are not always so clear. If it were always a choice between the "solidistas" and the "holeistas," our lives would be easier. Often the two (or more) alternatives are not so obviously differentiable, and the technical issues may be very muddy indeed. In
such cases, it is not always true that both sides are willing to admit that the other side has some merit. Often, technical criteria become submerged and the discussion degenerates into politics. This is generally bad.

**When Things Get Political**

So what is the legitimate place for politics in technical organizations?

Let's agree that *purely* technical issues need to be resolved technically. Once an issue moves out of the purely technical realm, then we admit politics. Examples of legitimate political issues include decisions on marketability, customer satisfaction, business impact, and so on. Since objective, technical criteria are harder to pin down in these areas, debate, including political debate, is necessary.

However, there are "good politics" and "bad politics." Political scientists would, of course, disagree: For them, all methods are more or less efficacious in their own ways, and they are loath to make value judgments. They might consider my categorizations naïve. So be it. Let's explore "good politics" and "bad politics" a little bit further.

**Good Politics**

The following techniques can be considered valid, legitimate parts of a political process:

1. Education
2. Persuasion
3. Consensus building
4. Fact-finding
5. Intellectually honest discussions
6. Identification of common interests
7. Exposure of hidden or subtle facts
8. Seeking compromise
9. Reasoning together

Plato could get behind this kind of politics. In fact, it has been observed that in organizations that do a good job of consensus building, decisions often are implemented more smoothly, because the consensus-building process causes diverse factions to come together and exercise reason, and to participate in making a joint decision. This process in effect "pre-sells" the idea, and the eventual solution has more buy-in from the participants than if there had not been consensus building.

By the way, I'm proud to observe that one of the hallmarks of the early Rational culture was an emphasis on consensus building to arrive at good decisions. A little later we needed to introduce "time bounded consensus seeking," wherein if consensus was not achieved by a certain time, a
decision was made through a somewhat less democratic, more informal process by the relevant manager, in the interest of moving on. Even in this case, the period during which consensus was sought did tend to give a good airing of all the alternative points of view.

**Neutral Politics**

The following behaviors are in the gray zone. Some people admit them as part of a legitimate political process, while others abhor them. I make no value judgment here other than to put them in the "neutral" category:

1. Cajoling
2. Ridiculing
3. Lobbying
4. Delaying
5. Defocusing issues
6. Positioning
7. Not telling all the truth all the time
8. You do this for me and I'll do that for you

Number eight is basically yielding on one point to gain another, what we called "bartering" above. One man's "horse trade" is another man's "compromise." This is why I place number eight in the gray zone.

**Bad Politics**

Without belaboring the neutral zone too long, we pass to those aspects of the political process that most people find unpleasant and "over the line":

1. Lying or deliberately misleading
2. Bribing
3. Intimidating, threatening, bullying
4. Undermining, conspiring, plotting
5. Personal attacks, abusive behavior
6. Filibustering
7. Hidden agendas
8. Committing to do something you have no intention of doing
9. Committing to not do something you have every intention of doing
10. Appeal to authority to subvert the process
11. "The end justifies the means," or "All's fair in love and war."

Sometimes this brand of politics is labeled "Machiavellian." This does a great disservice to Machiavelli, who had a lot more to say than "The ends
What happens when you get to bad politics is that you now are dealing with people who don't view politics as an adjunct to getting their jobs done. You are dealing with people for whom the political process itself is the primary preoccupation. Winning, by political or other means, is more important to these people than getting the job done. They have their priorities reversed, and, as such, are damaging to the organization. Certainly when you get to number eleven, which basically says, "There are no rules in a knife fight," you are beyond the pale.

**The Engineering Mapping**

The real problem with the gray zone is that many technical people have a bias toward the very high end of the integrity spectrum. They may put gray zone behaviors in their category of "bad politics." For example, not telling the whole truth all the time is technically "lying," yet some people admit a spectrum here, while others view it as black or white. It is difficult to have a "high trust" environment if there is a lot of gray zone behavior.

So don't be surprised, when you talk about "politics" with engineers and technical managers, if they lump "neutral (gray) zone" and "bad" politics under the general, derogatory heading of "politics" and characterize the "good" politics category as "leadership" or "good management." This turns things into a black and white world where all "politics" is evil. As this was my starting point, I now want to make it clear that we can sort things out, and what remains is really a linguistic mapping issue. The following table may prove helpful:

<table>
<thead>
<tr>
<th>Political Scientist’s View</th>
<th>Mainstream Usage</th>
<th>Engineers and Technical Managers</th>
</tr>
</thead>
<tbody>
<tr>
<td>My &quot;Good Politics&quot;</td>
<td>Legitimate</td>
<td>&quot;Leadership&quot;</td>
</tr>
<tr>
<td>My &quot;Neutral Politics&quot;</td>
<td>Doubtful</td>
<td>Not Acceptable</td>
</tr>
<tr>
<td>My &quot;Bad Politics&quot;</td>
<td>Not Acceptable</td>
<td>Not Acceptable</td>
</tr>
</tbody>
</table>

One could say that the entire thrust of this article is reconciling the "mainstream usage" mapping to the "engineering" mapping. These are two different ways of looking at the world, but without understanding the linguistic overloading we can get into a lot of trouble.

Especially when those engineers think we are "defending politics."  

**High Trust Environments**

We've mentioned "high trust environment" in the context of an organization's culture. Perhaps we should be explicit about what it is, and why it is so desirable.

In a high trust environment, the following is taken as the norm:
We can trust each other to tell the truth;
We can depend on each other to do whatever it takes;
We can depend on each other to put the organization's objectives ahead of our own personal or group objectives;
We can assume intellectual honesty in all discussions;
We don't take commitments lightly -- we view missed commitments as a violation of trust. This implies both volition and competency.

The thing that is so wonderful about high trust environments is that they are extremely efficient. When you have trust, you have less need for verification. This means you can get more done with smaller teams. It is the equivalent of reducing friction in a machine: More of the energy goes to producing work (good stuff) and less to producing heat (bad stuff). In general, it is easier to have high trust in small, relatively homogeneous groups. As teams get larger, more dispersed, and more diverse, it becomes harder to maintain a high trust environment. Ironically, the need doesn't diminish as the difficulty increases.

A high trust environment is desirable in all organizations. And, while it is as important or more important in technical organizations than in non-technical ones, it is harder to achieve. If we go back to the engineers' view of the world in the table above, we can see how even "neutral" politics poisons the high trust environment. If having a high trust environment is important to your organization, then you will need to move the bar up, and tolerate less "gray zone" behavior than you might otherwise accept.

Note: Once you have worked in a high trust environment, you will find it difficult to work in lower trust environments. All your instincts can turn out to be wrong, and a trusting person in a low trust environment is an easy target for political manipulators and can be made to suffer. On the other hand, it is crucial for people who have come from low trust environments to learn how to trust others once they are in a high trust environment; a lifetime of paranoia is difficult to overcome. Recent émigrés from the old Soviet Union have told me that assimilating into the less "on guard" American culture constitutes a difficult adjustment.

Other Variants of Bad Politics

In addition to the eleven items I noted under the "Bad Politics" section above, there are a few other negative behaviors that can crop up. We collect them here so that they don't escape unnoticed.

One particularly noxious form of bad politics is failing to "sign up." In every process, there is a period of discussion, followed by a decision. At this point, "the polls are closed." Once the decision is made, everyone must commit to it. Continued whining and "politicking" for rejected solutions after the fact is bad politics. Instead of allowing the organization to move on and implement the solution, it continues to mire down the issue in debate. Worse, failure to commit tends to undermine the leadership structure by calling decisions into question over and over again.
This form of bad politics must be eradicated whenever it crops up. Managers need to send a strong signal that "If you can't sign up, get off the team." If you have a healthy political process, this is a reasonable condition.

Another symptom of bad political behavior is obvious empire building. There are still many people out there who are caught up in the trappings and symbols of "success": rank in the organization, number of direct and indirect reports, size of budget, and so on. Organizations that overemphasize these statistical symbols reap what they sow; behaviors tend to follow the reward system. If, on the other hand, your organization rewards originality, contribution, "doing whatever it takes," and so on, independent of the individual's place in the hierarchy, then you are on the right track. In such organizations, the former behaviors tend to stick out like a sore thumb, and can be culturally discouraged. When individuals persist in them, they are mapping to different cultures that are more politicized, in which one's opinions have value according to one's place in the hierarchy as opposed to the opinion's intrinsic value. High trust organizations will ultimately reject such behavior, and if the behavior doesn't change, then the individuals who practice it must themselves be rejected. The sooner, the better.

Finally, there's a stylistic issue that is often characterized as political behavior, usually in a negative way. Some people turn every interaction into a negotiation; they are always "working the room." If nothing else, such individuals become tiresome, and tend to drain the energy of the group after awhile; people develop the habit of knowing that they have to "play defense" at all times with such individuals. I clearly run the risk of a libel suit if I try to give you an example here. But you all know the behavior, because there are certain individuals in every organization who are "annoying" in just this way. You tend to just not want to have to deal with them. Try not to be one, and coach others to avoid the practice.

**Summing Up**

Political process is a part of human and organizational behavior. To disengage from it is to cede authority to those who choose to continue to play. It is also morally untenable to disengage out of abhorrence to "politics" and then later claim that you were disenfranchised.

As leaders, however, we do have an obligation to avoid the trap of letting politics intrude on purely technical decisions. Even at the national level, we have sometimes made the mistake of allowing political concerns to govern decisions that were untenable on technical merit. When facts are clear, we cannot ignore them, regardless of the political consequences. Try not to violate laws of physics.

When we engage in "politics," we need to encourage "good politics" as indicated above. We need to become wary when we see "neutral zone" behavior starting to occur. And we need to stamp out and crush, preferably through peer pressure, "bad politics." Basically bad politics corrupts the entire process, by admitting as legitimate those behaviors that are clearly unethical and of low integrity.
Note that even "neutral" or "gray zone" practices will undermine a high trust environment. The advantages of a high trust environment are so great that one should seriously consider whether the allowance of "gray zone" practices is worth the risk.

We need to ensure that in all political processes or negotiations, the "polls close," that decisions are made, that the team "signs up," and that we move forward as one to implement the decision, without continuing non-productive debate.

Finally, for those of you who fall back on the "all politicians are corrupt scoundrels" argument, I have only two words for you: Harry Truman. Truman was no technologist, but he was a superb politician and leader. But that is the subject of yet another article.

Notes

1 http://www.therationaledge.com/.../k_buildItTheyWillCome_im.html
2 I must break with tradition here and explicitly credit and thank the editor, Mike Perrow, for this excellent formulation of my more direct but less clear articulation.
3 A pretty awful place in the nether regions of the castle. The word derives from the French oublier, to forget.
4 This is a very important concept, and we will return to it later in the article. High trust environments are valuable to all organizations, and we have found that they are especially important in technical organizations.
5 Let us note once again for emphasis the North American bias. Some or all of the gray zone behaviors might be considered OK, acceptable, dubious, or very bad by Europeans or Asians. This is where things get very complex. But in order to frame the discussion at all, I needed to define three categories. What I put into them reflects the North American point of view, which itself is an approximation.
6 We apologize also to Sun Tzu, who we could not work into this article at all.
7 We often find some curious and contradictory manifestations of engineers' value systems. For example, it is typical for engineers to feel uncomfortable in discussions with customers if they think you are not being "100% honest" in all your utterances. Here they apply what can sometimes be an overly rigorous standard. On the other hand, they are puzzled when their integrity seems to be impugned because they missed a schedule milestone. For them, this was just a technical problem, not an issue of a missed commitment and violation of trust. Understanding some of these characteristics is important in dealing with technical organizations, because the obvious mismatches with other parts of the organization can lead to communication problems.
8 But here is a sports analogy: In tennis, this behavior is akin to always playing at the net.

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