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:

\[ \text{Payday} = \text{Friday} \]
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 overwriting 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-interactive, 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 decelarators: 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; naiveté 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!

Copyright Rational Software 2001 | Privacy/Legal Information