A technical discussion on Quality on Time
November, 2003

Rational. software

Quality, on Time

By Aki Fujimura,
Chief Technology Officer,
New Business Incubation,
Cadence Design Systems, Inc.

Table of Contents

How is it that a group of talented, highly motivated, hard working software engineers consistently produce low-quality software, late? It is the author's view that schedule management and quality management go hand in hand. This paper discusses the notion that schedules are probability distributions, and presents several, practical quality and schedule management techniques.

Background

Software is everywhere these days. Cellular phones, automobiles, dishwashers, telephone switches, credit card transactions, stock trading, power plants, medical equipment, packaged software—the world relies on software. And the quality of everything around us depends upon software quality. The time-to-market for everything around us depends on timely delivery of software components.

Yet somehow, delivering quality software on time seems to be an impossible task. Some managers have given up on the notion entirely. They just do "the best they can." Other managers make a choice between delivering quality and delivering on time.

For many, the decision is clear: "Quality is important, but I have to deliver something to the customer ASAP." The customer is willing to tolerate poor quality and to "work with the vendor" on quality issues.

For some managers, it is the other way around: "Quality cannot be sacrificed at any cost. The cost of failure is so high that we must take the time to get it right." These managers care about the right issues, but they soon find that it is difficult to achieve software quality efficiently. Time-to-market suffers. The competition releases poor quality software, with greater functionality, earlier. Projected revenue shortfalls create pressure to release the new product. It is only the most disciplined organizations that can withstand that pressure and maintain the quality-first attitude and behavior.

Deliver Quality, on Time

The only answer to the software quality dilemma is delivering a fit-for-use product, on time. This involves three steps:

- *Quality first: value it, act on it, measure it, and improve it*
- *Implement the Must-Should-Could technique for schedule management*
- *Automate software quality to improve phase containment*

The first and third steps are discussed widely in many forums; this paper concentrates on the second step.

Schedule Is a Probability Distribution

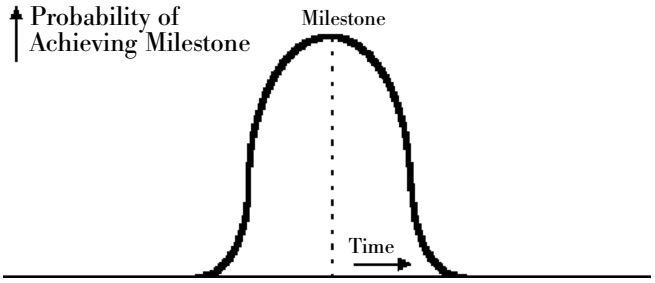
The IBM Rational engineering initiative, Quality by Design, referred to in the first step, enhances the abilities of architects, analysts, developers, testers and project leads to improve the quality of their work.

Properly managing a project's schedule further improves the ability of the team to deliver fit-for-use products, on time. First, we must understand the complexity of the software schedule management problem. The odds are against us!

Most people think of schedule milestones as static dates. Achieving the goal early would be interpreted as bad scheduling; being late is seen as poor execution. We must understand that a project schedule milestone is a probability distribution.

Figure 1 depicts a normal distribution for a schedule milestone. There is no single date of which one can say, "I am 100% sure that the project will be done by this date."

Figure 1. Normal distribution of a schedule milestone



The notion that it is possible to name a date that can be met with certainty is an approximation of reality. This approximation leads to great misunderstanding and mismanagement, especially in larger projects.

Figure 2. Most people's view of a deadline

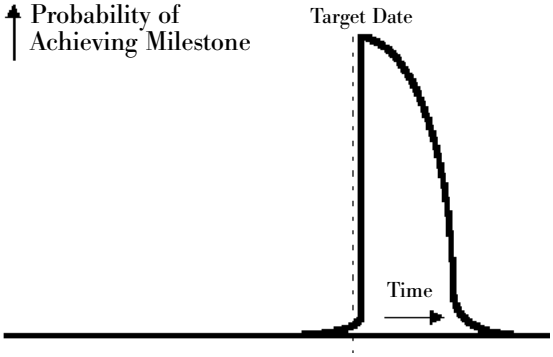
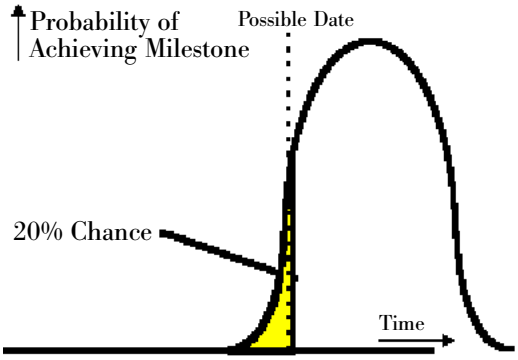


Figure 2 depicts the probability distribution as described by a typical project engineer for any arbitrary deadline. In an astonishingly high percentage of software projects, the response is: "There's zero percent chance that we'll be done a day early, but I think I can make the deadline."

In reality, nothing has a distribution curve like the one in Figure 2. The real distribution is more like the one in Figure 3.

Figure 3. Reality of how people schedule under pressure



Since most well meaning, hard working engineers give the first conceivable deadline as the due date, it is most likely that the probability of meeting that date is less than 20%. What the engineer is really saying is, "It's a 20% schedule, but I'll make it an 80% schedule through sheer determination and hard work." What we ought to strive for is an 80% probability schedule, where the probability of finishing early is greater than the probability of finishing late. Yet getting 80% schedules from every individual is a difficult task.

Successful software organizations grow. As the customer base and the software infrastructure surrounding it matures, products become subproducts of larger entities. The integrated whole becomes the goal rather than the individual tools. At this stage, software organizations, regardless of talent and track record, often stumble. This is because even 80% schedules don't scale in larger projects.

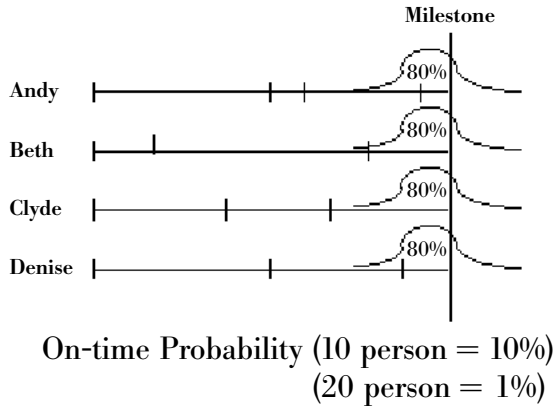
The Multiplying Probabilities

Let us take a look at a ten-person software project with a six month development schedule. The product is entering version 3.0, a significant enhancement over version 2.0 released last year. The motivated and hard working engineers are talented graduates of top schools. The seasoned manager understands the technology and is well aware of the competitive threats. She works well with the marketing department and the customers to define the new version. Simultaneously, she works with the engineers to extend the product architecture, define numerous programming tasks, minimize interdependencies and interfaces, and produces a bottoms-up, 80% schedule that matches customer expectations.

Each person in the group is assigned several tasks over the six months. The manager works hard to fit in every feature she can and still make the scheduled deadline. The schedule takes into consideration vacations, holidays, and other anticipated time off. Every one of the tasks must be completed before the project is considered successful.

In most software projects, team members work on projects largely independently from each other. Managers try to ensure that this is the case. Also, particularly in successful and fast growing organizations, the tasks assigned to each individual require that particular individual to complete the work. Last minute reassignment is typically not possible, either because of lack of expertise or ownership reasons. For the sake of simplicity, we will assume that the probability of Andy making his six-month schedule is independent of the probability of anyone else making their schedules. This situation is illustrated in Figure 4.

Figure 4. Multiplying probabilities in a multi-person project



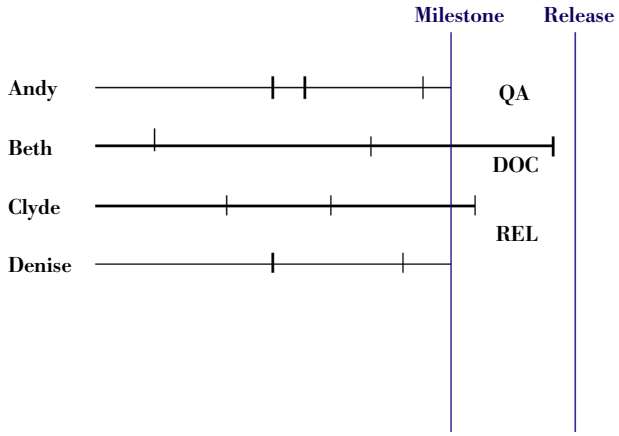
For a ten person team, the probability of this software project being on time is thus 80% multiplied by 80% multiplied by... and so on, 10 times. Before the project even starts, we know the probability that every task on the schedule will be completed before the deadline is only 10.7%!

If schedule reliability is the paramount issue, then the manager can insist on 99% schedules from each individual. But this will produce an unacceptably long schedule and will not even be sufficient to overcome the multiplying probabilities for large projects. For example, a 99% schedule produced by a team of 100 engineers makes for only a 37% chance of meeting the overall schedule. We

should recall that most people actually make 20% schedules and hope to make up the rest with sheer determination and hard work.

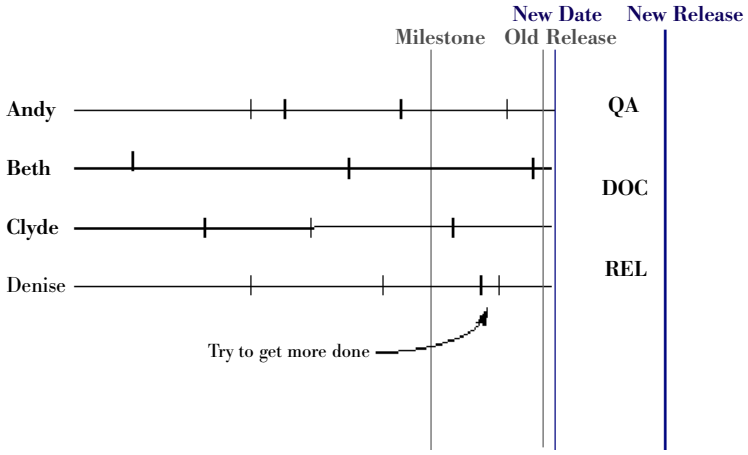
Scheduling to 20% and working more to make up the rest can at best produce 80% schedules, but never a 99% schedule.

Figure 5. Someone is late



Lacking an alternative, our example project proceeds on the 80% plan. Inevitably, someone in the project is late. It is not because someone is inexperienced, incompetent, or lazy. Simply by the law of probability, someone will be late. What happens then, in a typical software project is that the team and the manager recognize an opportunity to turn the delay into a benefit for the customers. Everyone else has to wait for the person who is late anyway. There are plenty of small additional tasks that can enhance this release. The well-intentioned manager reschedules the release and fills everyone's plate to the maximum extent possible. But the manager knows the value of quality. So with the help of the Quality Assurance manager, she slips the final release date line early with the development schedule to ensure ample time for system testing, documentation, and release engineering (Figure 6).

Figure 6. Project is rescheduled to get the most done given the delay



Of course, the multiplying probabilities are still in effect in this newly rescheduled project. Someone is late again. It may be the same person. It may be someone different. Even if everyone were equally competent and equally hard working, the law of probability is against everyone completing on time.

Once again, there is a need to reschedule. But this time, things are a little different. It has been a while since the start of the project when the product was defined. New operating systems have been released. Better hardware has become available. Competition has created new demands on the feature set. Customers have seen what is possible and have become more demanding. There has been a reorganization in the division, and the new general manager wants a different kind of release than was originally planned.

So this time, a fairly massive rescheduling is done. This release is slightly different from the original plan, but it is better and more in line with today's requirements. Of course, the schedule pressure is greater than ever. Everyone is called upon to put in 110%. Everyone's schedule is filled to the last day with new tasks. Everyone is excited about the new charter. The engineers are happy to work on new things rather than working to perfect the last 20% of their earlier tasks. Everyone feels good and charges ahead. But someone is late again. No one is at fault. Everyone tried hard. But the law of probability wins again, and someone is late.

Unlike last time, however, all the good intentions to put quality first cannot win against the need to release version 3.0. There may be revenue pressures. There may be customer organizations questioning "make or buy." There may be the board of directors getting tired of hearing about yet another delay. There may be pressure from the press and the analyst community. In any case, releasing the software becomes more important than giving the back-end processes ample time to assure quality. The "back-end squeeze" occurs, and the quality of the product suffers.

In addition, the constant reprioritization and schedule slips have consistently delayed full testing of any of the functions. The proverbial "last 20% that takes 80% of the effort" has accumulated throughout the multiple schedule delays. Without the discipline and the rigor of the release process, the quality of the software continues to decline to the point where recovery takes tremendous energy and effort. Because the tedious "last 20%" work accumulates at the end, engineering morale quickly decays in long hard hours of debugging and rework.

Even when the product is released, the work is not over. If the product is released without adequate testing, it can contain obvious flaws created in the last minute rush and inconsistencies from changing product goals during the development cycle. It doesn't serve any one customer particularly well. It has low reliability. Performance characterization hasn't even begun to be a priority. And it is late.

One Poor Quality Release

The entire software organization is now in trouble. Every single department will slowly decline in productivity, increase in cost, and stop innovating. The customer support organization suffers first. The call rate triples. What used to be "I love the software" now becomes "Did you test it before you gave it to us? Do you know that it took me three tries in four hours to reach you?" Morale of the group plummets, and product knowledge declines as people get burned out.

Trust is fragile. A broken trust caused by late delivery or poor quality is very difficult to mend. The pride in representing the organization declines rapidly.

The development organization feels the pain. Minor releases are required more frequently now. For every bug found by a customer, there is tremendous

overhead. Not only must the bug be fixed, tested, and released, but workarounds must be found and delivered. The workaround and the eventual fix must be documented. Every support person must be educated on each fix. And with every fix, the customer needs hand-holding.

The next release must be built, but the last release still needs attention. Even the parts that are working have temporary "hacks" that the engineers know "have to be rewritten." The code was developed at tremendous speed during the release because it was written in a highly personalized way. It wasn't written for someone else to understand it; that would have taken much more time than the business conditions allowed. The managers have a tough choice: either keep their best people on the maintenance task, or have them spend their time training others on their code. In the pressure of "the customer needs the fix tomorrow," it is rarely possible to train others.

What was once a thriving and innovative organization turns into a typical software maintenance shop that spends 80% of its time fixing rather than creating. Lack of innovation leads to employee turnover, which leads to further decline in productivity. Lack of innovation also causes a decline in new product revenue, which in turn will cause a slow death for the organization.

If no competitor does better in the meanwhile, the organization will have time to recover: put quality back in order, address process and cultural issues, and get back on track to being a successful, growing entity. But if someone else does, a low quality release can be the beginning of the end.

This is how a group of highly motivated, well intentioned, and talented software developers and managers end up producing low quality software, late.

Software Projects Are Special

Delivery quality on time can be hard to achieve. But it is possible with the right tools and attitude. It is just as possible, practical, and necessary to fix this problem in software design as it was in automotive or semiconductor design. However, software engineering does have some differences.

In addition to the generic difficulties of any engineering project, software projects have unique characteristics:

- *Relatively little manufacturing in the back-end*
- *The next version is built on top of the current version*
- *An illusion of ability to change at the last minute*

Manufacturing is a relatively small part of the software process for most organizations. This is deceptively convenient. Releasing software involves more than just completing the code. Support engineers must be trained. Documentation that is consistent with the code must be completed and printed. For Independent Software Vendors, marketing literature must be finished. Sales people must be trained.

If manufacturing took more time, all of these other activities could happen while manufacturing was ramping, well after code development is completed. But this is not the case for most software organizations. Many of these activities that depend on a certain deployment date must proceed in parallel with the final code development and testing. This causes a dependency that often leads to increased pressure to release on time without quality.

In software, release-to-release reuse is very high. Typically, the next version of the software is built on top of the current version. This compounds the quality problem.

It is impossible to create a quality release by adding functionality on top of a release that was not of high quality. Many organizations that get into the quality quandary try to work on enhancements in parallel with fixing defects. But working on enhancements is difficult because the developer can never be sure whether the problem in the code was caused by his or her changes or whether it was already in the code. In retrospect, it would often have been better to rebuild the whole system from scratch. But at each individual decision point, it never seems right to redo the whole thing when a patch can be made in much less time.

The biggest barrier to delivering fit-for-use software on time is the illusion that software changes are easy to make. This is true at the beginning of the development process. But during the late stages of the development process, software changes are just as complex as any hardware changes.

In the beginning, it is important to take advantage of the flexibility of software. Make lots of changes. Try different things. The gray line between prototype and production should be considered an advantage by software engineers and managers.

But applying this same approach near the end of the project is a disaster in the making for both quality and time-to-market. That the code itself is easy to change creates an unfortunate illusion that the product as a whole is easy to change. It is not. Code is only a small portion of what is required to deliver a software product to a customer. In order for the customers to take advantage of the change, it must be documented, incorporated in training material, taught to the support engineers, and presented to all representatives of the product. As the product release date approaches, these other aspects of releasing a product to the customer must take over as the dominant considerations.

In addition, the laws of probability apply in making changes. No engineer is more than 99% certain of any given change. The compiler, the static analyzers, regression tests, code reviews, Purify, and other techniques are fully deployed, especially towards the end of the release cycle, to minimize mistakes. But still, no change is more than 99% certain. The probability that one of the changes turns out to be wrong is calculated by raising 99% to the power of the number of changes made. With one hundred changes, the overall probability of something going wrong is 63%. For this reason, it is critical that engineers gradually take their hands off the code as the release date approaches.

Releasing Quality Software, on Time

It is possible. The first step is having the entire organization believing in it and understanding its tremendous benefits. Everyone, starting with top management, must agree that, must agree that meeting the release date with a highly fit-for-use product is the goal, and that a late quality release is second best. Organizations must stand firm and not release products that fail to meet the

internal quality standards. This point must be emphasized and re-emphasized by management through training and, most importantly, through action.

Put Quality First

A common practice among software engineering management is to have incentives based on software release dates. Theoretically, the software product doesn't reach the release milestone until it has achieved its quality goals. This type of incentive can create a conflict with the need to release only quality code. Smaller organizations can manage this conflict by carefully framing the incentive milestone statements to include their quality goals.

For larger organizations, more long-term objective setting is realistic, and infrastructural support for more elaborate measurements are practical. For them, the incentive goals should be based on achieving the break-even time by measuring "time to money" instead of measuring time to market, the incentive incorporates the effects of the quality of the release. Low quality releases will cost the organization more after the release than before. High quality releases will be accepted faster, and benefits of the release will be realized more quickly.

Adhere to the Schedule

Once quality becomes the top priority of the management team, the next step is to realize that quality releases are only possible if the projects are reliably on schedule. In order to maintain software quality, it is critical for management to consistently demonstrate the virtues of schedule adherence.

Management and other influencers of software projects often comment on the specifications of the product after they see the nearly completed product. The late changes that result are always well intentioned, but are often harmful to both the schedule and the quality of the product. This management behavior encourages "feature creep." It promotes the hero-oriented culture that works hard to create the first 80% that can be done by an individual working overnight, but never has time to complete the last 20% that must be done by the whole team by working diligently over months of elapsed time. The right solution is usually to get this release out on time, and then to make the change or addition in the next release.

One way for the management to reinforce the value of schedules is to implement the "Must-Should-Could" (M-S-C) method of Dr. Robert Fulks then Chief Technologist of Cadence Design and currently a board member and technical advisor to Pittsburgh Simulation. Noticing that building slack time into schedules only causes productivity to decrease, the M-S-C system balances the desire to drive productivity with the need to create realistic schedules.

In the M-S-C system, when a project is first scheduled, the tasks in the project are broken up into Must items, Should items, and Could items. Only the Must items are committed to the customers. Must items are scheduled first. The schedule commitment is made so that the last scheduled item for each person is a Should. In this way, even though the probability of completing every Should item on the list is low, the probability of completing every Must item is high.

Could items are scheduled for each engineer after the scheduled completion milestone such as "functionality freeze." Since properly scheduled projects will have 80% of the tasks end ahead of time, it should theoretically be possible to complete many of the smaller Could items prior to the milestone. Every Should and Could item must be selected in such a way that each item can be in or out of the release in the last weeks approaching the milestone date. For example, multiple items that depend on each other to be completed are either all Musts or all in the next release. As another example, an item that has a large impact on the overall look and feel of the product is likely to have too many documentation and other impacts to be a Should or a Could item.

In this system, the traditional "slack" is replaced by real tasks whose benefit to the customer are clear. Everyone involved understands that building quality on schedule is more important than getting every single feature into the release. The project teams strive for an early completion of the Must items so that a few Should items can be in the release. The usual caution is taken to ensure that the entire production required for the item, not just the code, can be completed before Should items are developed. Having the list of potential Should items available from the beginning of the project allows for better planning on the part of the documentation team and other staff.

Over time, statistics for actual completion of planned Must, Should, and Could items are collected. The organization strives for 100% completion of Must items by the deadline, and approximately 80% completion of Should items by the

deadline. Pure Software deploys the M-S-C system. Projects that achieved Quality on Time had the goal of completing only the Must items, but actually completed 20-50% of the Should items. Projects that were not on time completed many of the Should items and even some of the Could items before the function freeze.

Only the Beginning

Making quality a top priority and believing in schedule adherence is only the beginning. Continuous improvement principles must be patiently and persistently applied. Metrics must be implemented and constantly updated. Flexible processes must be installed and documented. Templates of standard documents and checklists for standard milestones must be created and maintained. A frequent process review meeting must be used to remind everyone to think about improving the organizational competence, not just the products.

Quality Assurance must be empowered both to check for quality prior to the release and to improve the organization's ability to build in quality. At every project milestone, a specific set of quality goals should be established and measured.

Phase containment should be measured and continuously improved. Phase containment is the degree to which a defect created is discovered and fixed in the same phase. It is always more costly to fix defects after a project milestone than it is to fix them before that milestone. If a defect can be detected and fixed in unit test by the individual developer, it should happen before the code is integrated and made available to other developers. If a defect can be detected and fixed during system test, it should happen before code is released for test deployment. If a defect can be detected and fixed during test deployment, it should happen before general deployment. Maximizing phase containment is the essence of Automated Software Quality tools. (This is the subject of an upcoming paper.)

Conclusion

Achieving an on-time release of excellent software is hard work, and it is even harder to maintain it in a successful and growing organization. But it is possible and it must be done. Although engineering software products is different from engineering automobiles or semiconductor devices, the lessons learned from the quality panic of the 1970's are directly applicable to the software industry.

Specifically, the discipline of quality systems applied to other industries should directly be applied to software towards the last third of the product development cycle. In the first two-thirds, the flexibility allowed in software should be maximized while applying special schedule management techniques such as the M-S-C system.

The ability to consistently produce software that is stable and of high quality, by the ordained release date is fundamental to all software organizations. Quality is the foundation that will start the upward cycle of positive change for every software organization.

About the Author

Aki Fujimura serves as a Cadence Chief Technology Officer and is responsible for Cadence's innovation cycle, bringing advanced new technologies into new products and new businesses.

Before his appointment as CTO, Fujimura served as general manager of Design for Manufacturing, IC Solutions. Fujimura came to Cadence for the second time in June 2002, with the acquisition of Simplex Solutions, where he was president, chief operating officer and a director of the board. Before joining Simplex, he was vice president and a director on the board of Pure Software, which was acquired by Rational Software. Fujimura holds a B.S. and M.S. in electrical engineering and computer science from the Massachusetts Institute of Technology.

Cadence® Design has a deep-rooted relationship with IBM. Cadence is an Advanced IBM Business Partner and has collaborated with IBM to support IBM foundry technologies with EDA tools and solutions, design services and IP development. In September 2002, IBM and Cadence established a new series of agreements. These agreements extend the licensing of Cadence EDA tools by IBM for both internal and external design projects.

IBM software integrated solutions

IBM Rational supports a wealth of other offerings from IBM software. IBM software solutions can give you the power to achieve your priority business and IT goals.

- *DB2® software helps you leverage information with solutions for data enablement, data management, and data distribution.*
- *Lotus® software helps your staff be productive with solutions for authoring, managing, communicating, and sharing knowledge.*
- *Tivoli® software helps you manage the technology that runs your e-business infrastructure.*
- *WebSphere® software helps you extend your existing business-critical processes to the Web.*
- *Rational® software helps you improve your software development capability with tools, services, and best practices.*

Rational software from IBM

Rational software from IBM helps organizations create business value by improving their software development capability. The Rational software development platform integrates software engineering best practices, tools, and services. With it, organizations thrive in an on demand world by being more responsive, resilient, and focused. Rational's standards-based, cross-platform solution helps software development teams create and extend business applications, embedded systems and software products. Ninety-eight of the Fortune 100 rely on Rational tools to build better software, faster. Additional information is available at www.rational.com and www.therationaledge.com, the monthly e-zine for the Rational community.

IBM Corporation
Software Group
Route 100
Somers, NY 10589
U.S.A.

Printed in the United States of America 05-03

The IBM home page on the Internet can be found at **ibm.com**