From Waterfall to Iterative Development—A Challenging Transition for Project Managers

Philippe Kruchten

Rational Software White Paper

TP 173, 5/00

Rational®
the software development company

Table of Contents

Introduction

The Rational Unified Process (RUP) advocates an iterative or spiral approach to the software development lifecycle, as this approach has again and again proven to be superior to the waterfall approach in many respects. But do not believe for one second that the many benefits an iterative lifecycle provides come for free. Iterative development is not a magical wand that when waved solves all possible problems or difficulties in software development. Projects are not easier to set up, to plan, or to control just because they are iterative. The project manager will actually have a more challenging task, especially during his or her first iterative project, and most certainly during the early iterations of that project, when risks are high and early failure possible. In this article, I describe some of the challenges of iterative development from the perspective of the project manager. I also describe some of the common "traps" or pitfalls that we, at Rational, have seen project managers fall into through our consulting experience, or from reports and war stories from our Rational colleagues.

Iterative Development

Classic software development processes follow the waterfall lifecycle, as illustrated in the following figure. In this approach, shown in Figure 1, development proceeds linearly from requirements analysis through design, code and unit testing, subsystem testing, and system testing, with limited feedback on the results of the previous phases.

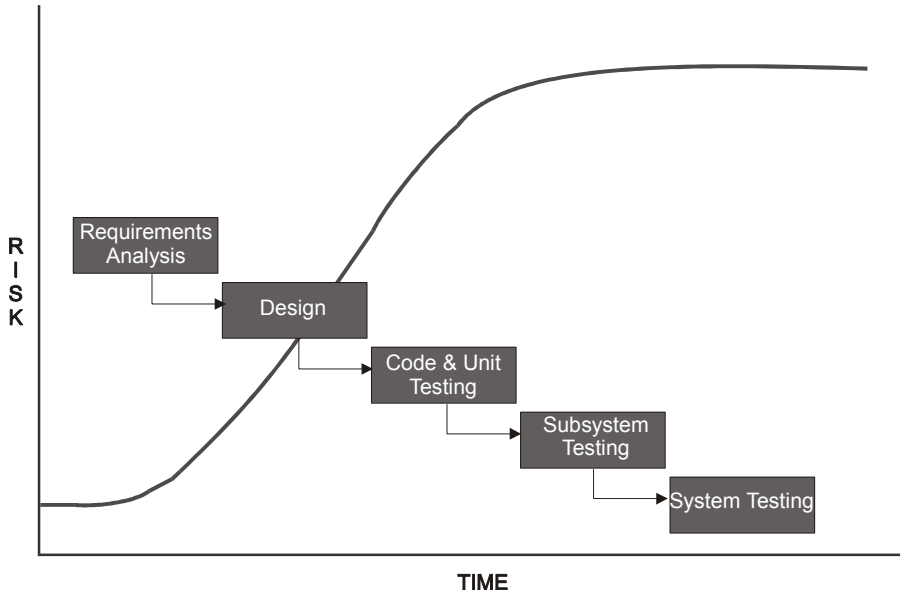

Figure 1: The Waterfall Development Process

The fundamental problem of this approach is that it pushes risk forward in time, where it's costly to undo mistakes from earlier phases. An initial design will likely be flawed with respect to its key requirements, and furthermore, the late discovery of design defects tends to result in costly overruns and/or project cancellation. The waterfall approach tends to mask the real risks to a project until it is too late to do anything meaningful about them.

An alternative to the waterfall approach is the iterative and incremental process, as shown in Figure 2.

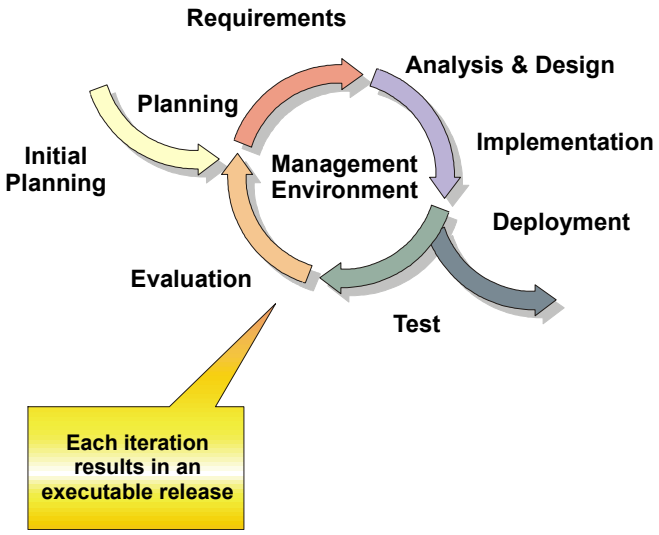

Figure 2: An Iterative Approach to Development

In this approach, built upon the work of Barry Boehm's spiral model (see "Further Reading"), the identification of risks to a project is forced early in the lifecycle, where it's possible to attack and react to them in a timely and efficient manner. This approach is one of continuous discovery, invention, and implementation, with each iteration forcing the development team to drive to closure the project's artifacts in a predictable and repeatable way.

The Good: Benefits of Iterative Development

Compared with the traditional waterfall process, the iterative process has many advantages.

1. Serious misunderstandings are made evident early in the lifecycle, when it's possible to react to them.
2. It enables and encourages user feedback, so as to elicit the system's real requirements.
3. The development team is forced to focus on those issues that are most critical to the project, and team members are shielded from those issues that distract them from the project's real risks.
4. Continuous, iterative testing enables an objective assessment of the project's status.
5. Inconsistencies among requirements, designs, and implementations are detected early.
6. The workload of the team, especially the testing team, is spread out more evenly throughout the lifecycle.
7. This approach enables the team to leverage lessons learned, and therefore to continuously improve the process.
8. Stakeholders in the project can be given concrete evidence of the project's status throughout the lifecycle.

Risk Mitigation

An iterative process lets you mitigate risks earlier because integration is generally the only time that risks are discovered or addressed. As you roll out the early iterations you go through all process components, exercising many aspects of the project, including tools, off-the-shelf software, and people skills. Perceived risks will prove not to be risks, and new, unsuspected risks will be discovered.

If a project must fail for some reason, let it fail as soon as possible, before a lot of time, effort, and money are expended. Do not bury your head in the sand too long; instead, confront the risks. Among other risks, such as building the wrong product, there are two categories of risks that an iterative development process helps to mitigate early:

- Integration risks

- Architectural risks

An iterative process results in a more robust architecture because you correct errors over several iterations. Flaws are detected in early iterations as the product moves beyond inception. Performance bottlenecks are discovered at a time when they can still be addressed instead of being discovered on the eve of delivery.

Integration is not one "big bang" at the end of the life cycle; instead, elements are integrated progressively. Actually, the iterative approach that we recommend involves almost continuous integration. What used to be a lengthy time of uncertainty and pain—taking as much as 40% of the total effort at the end of a project—is now broken into six to nine smaller integrations that begin with far fewer elements to integrate.

Accommodating Changes

You can envisage several categories of changes:

- Changes in Requirements

  An iterative process lets you take into account changing requirements. The truth is that requirements will normally change. Requirements change and "requirements creep" have always been primary sources of project trouble, leading to late delivery, missed schedules, unsatisfied customers, and frustrated developers. But by exposing users (or representatives of users) to an early version of the product, you can ensure a better fit of the product to the task.

- Tactical Changes

  An iterative process provides management with a way to make tactical changes to the product—for example, to compete with existing products. You can decide to release a product early with reduced functionality to counter a move by a competitor, or you can adopt another vendor for a given technology. You can also reorganize the contents of an iteration to alleviate an integration problem that needs to be fixed by a supplier.

- Technological Changes

  To a lesser extent, an iterative approach lets you accommodate technological changes. You can use it during the elaboration phase, but you should avoid this kind of change during construction and transition because it is inherently risky.

Learning as You Go

An advantage of the iterative process is that developers can learn along the way, and the various competencies and specialties are more fully employed during the entire life cycle. For example, testers start testing early, technical writers write early, and so on, whereas in a non-iterative development, the same people would be waiting to begin their work, making plan after plan. Training needs—or the need for additional (perhaps external) help—are spotted early during assessment reviews.

The process itself can also be improved and refined along the way. The assessment at the end of an iteration looks at the status of the project from a product/schedule perspective and analyzes what should be changed in the organization and in the process so that it can perform better in the next iteration.

Increased Opportunity for Reuse

An iterative process facilitates reuse of project elements because it is easier to identify common parts as they are partially designed or implemented instead of identifying all commonality in the beginning. Identifying and developing reusable parts is difficult. Design reviews in early iterations allow architects to identify unsuspected potential reuse and to develop and mature common code in subsequent iterations. It is during the iterations in the elaboration phase that common solutions for common problems are found and patterns and architectural mechanisms that apply across the system are identified.

Better Overall Quality

The product that results from an iterative process will be of better overall quality than are products that result from a conventional sequential process. The system will have been tested several times, improving the quality of testing. The requirements will have been refined and are more closely related to the users' real needs. And at the time of delivery, the system will have been running longer.

The Hard: Unexpected Downside and Common Traps

Iterative development does not necessarily mean less work and shorter schedules. Its main advantage is to bring more predictability to the outcome and the schedule. It will bring higher quality products, which will satisfy the real needs of end-users, because you will have had time to evolve requirements as well as a design and an implementation.

Iterative development actually involves much more planning and is therefore likely to put more burden on the project manager: an overall plan has to be developed, and detailed plans will in turn be developed for each iteration. It also involves continuous negotiation of tradeoffs between the problem, the solution and the plan. More architectural planning will also take place earlier. Artifacts (plans, documents, models and code) will have to be modified, reviewed and approved repeatedly at each revision. Tactical changes or scope changes will force some continuous replanning. Thus, team structure will have to be modified slightly at each iteration.

Trap: Overly Detailed Planning Up to the End

It is typically wasteful to construct a detailed plan end-to-end, except as an exercise in evaluating the global envelope of schedule and resources. This plan will be obsolete before reaching the end of the first iteration. Before you have an architecture in place and a firm grip on the requirements—which occurs roughly at the LCA milestone—you cannot build a realistic plan.

So, incorporate precision in planning commensurate with your knowledge of the activity, the artifact or the iteration being planned. Near-term plans are more detailed and fine grained. Long-term plans are maintained in coarse-grained format.

Resist the pressure that unknowledgeable or ill-informed management may bring to bear in an attempt to elicit a "comprehensive overall plan." Educate managers, and explain the notion of iterative planning and the wasted effort of trying to predict details far into the future. An analogy that is useful: a car trip from New York to L.A. You plan the overall route but only need detailed driving instructions to get you out of the city and onto the first leg of the trip. Planning the exact details of driving through Kansas, let alone the arrival in California, is unnecessary, as you may find that the road through Kansas is under repair and you need to find an alternate route, etc.

Acknowledging Rework Up Front

In a waterfall approach, too much rework comes at the very end, as an annoying and often unplanned consequence of finding nasty bugs during final testing and integration. Even worse, you discover that most of the cause of the "breakage" comes from errors in the design, which you attempt to palliate in implementation by building workarounds that lead to more breakage.

In an iterative approach, you simply acknowledge up front that there will be rework, and initially a lot of rework: as you discover problems in the early architectural prototypes, you need to fix them. Also, in order to build executable prototypes, stubs and scaffolding will have to be built, to be replaced later by more mature and robust implementations. In a healthy iterative project, the percentage of scrap or rework should diminish rapidly; the changes should be less widespread as the architecture stabilizes and the hard issues are being resolved.

Trap: Project not converging

Iterative development does not mean scrapping everything at each iteration. Scrap and rework has to diminish from iteration to iteration, especially after the architecture is baselined at the LCA milestone. Developers often want to take advantage of iterative development to do gold plating: to introduce yet a better technique, to perform rework, etc. The project manager has to be vigilant so as to not allow rework of elements that are not broken—that are OK or good enough. Also, as the development team grows in size, and as some people are moved around, newcomers are brought in. They tend to have their own ideas about how things should have been done. Similarly, customers (or their representatives in the project: marketing, product management) may want to abuse the latitude offered by iterative development to accommodate changes, and/or to change or add requirements with no end. This effect is sometimes called "Requirements Creep." Again the project manager needs to be ruthless in making tradeoffs and in negotiating priorities. Around the LCA milestone, the requirements are baselined, and unless the schedule and budget are renegotiated, any change has a finite cost: getting something in means pulling something out.

And, remember that "Perfect is the enemy of good." (Or in French: "Le mieux est l'ennemi du bien.")

Trap: Let's Get Started; We'll Decide Where to Go Later

Iterative development does not mean perpetually fuzzy development. You should not simply begin designing and coding just to keep the team busy or with the hope that clear goals will suddenly emerge. You still need to define clear goals, put them in writing, and obtain concurrence from all parties; then refine them, expand them, and obtain concurrence yet again. The bright side is that in iterative development, you need not have all the requirements stated before you start designing, coding, integrating, testing, and validating them.

Trap: Falling Victim to Your Own Success

An interesting risk comes near the end of a project, at the moment the "consumer bit" flips. By this we mean that the users go from believing that nothing will ever be delivered to believing that the team might actually pull it off. The good news is that the external perception of the project has shifted: whereas on Monday the users would have been happy if anything were delivered, on Tuesday, they become concerned that not everything will be delivered. This is the bad news. Somewhere between the first and second beta, you find yourself inundated with features people are concerned about making it into the first release. Suddenly, these become major issues. The project manager goes from worrying about delivering minimal acceptable functionality to a situation in which every last requirement is now "essential" to the first delivery. It is almost as though, when this bit flips, all outstanding items get elevated to an "A" priority status. The reality is that there is still the same number of things to do, and the same amount of time in which to do them. While external perception may have changed, prioritization is still very, very important.

If, at this crucial moment, the project manager loses his nerve and starts to cave in to all requests, he actually puts the project in schedule danger again! It is at this point that he or she must continue to be ruthless and not succumb to new requests. Even trading off something new for something taken out may increase risk at this point. Without vigilance, one can snatch defeat from the jaws of success.

Putting the Software First

In a waterfall approach, there is a lot of emphasis on "the specs" (i.e., the problem-space description) and getting them right, complete, polished and signed-off. In the iterative process, the software you develop comes first. The software architecture (i.e., the solution-space description) needs to drive early lifecycle decisions. Customers do not buy specs; it is the software product that is the main focus of attention throughout, with both specs and software evolving in parallel. This focus on "software first" has some impact on the various teams: testers, for example, may be used to receiving complete, stable specs, with plenty of advance notice to start testing; whereas in an iterative development, they have to begin working at once, with specs and requirements that are still evolving.

Trap: Too Much Focus on Management Artifacts

Some managers say, "I am a project manager, so I should focus on having the best set of management artifacts I can; they are key to everything." Not quite true! Although good management is key, the project manager must ensure in the end that the final product is the best that can be produced. Project management is not an exercise in covering yourself by showing that you have failed despite the best possible management. Similarly, you may focus on developing the best possible spec because you have been hurt by poor requirements management in the past; this will be of no use whatsoever if the corresponding product is buggy, slow, unstable, and brittle.

Hitting Hard Problems Earlier

In a waterfall approach, many of the hard problems, the risky things, and the real unknowns are pushed to the right in the planning process, for resolution during the dreaded system integration activity. This leaves the first half of the project as a relatively comfortable ride, where issues are dealt with on paper, in writing, without involving many stakeholders (testers, etc.), hardware platforms, real users, or the real environment. And then suddenly, the project enters integration Hell and everything breaks loose. In iterative development, planning is mostly based on risks and unknowns, so things are tough right from the onset. Some hard, critical and often low-level technical issues have to be dealt with immediately, rather than pushed out to some later time. In short, as someone once said to me: in an iterative development you cannot lie (to yourself or to the world) very long. A software project destined for failure should meet its destiny earlier in an iterative approach.

One analogy is a university course in which the professor spends the first half of the semester on relatively basic concepts, giving the impression that it is an easy class that allows students to receive good marks at the mid-term with minimal effort.

Then suddenly, acceleration occurs as the semester comes to a close. The professor tackles all the challenging topics shortly before the final exam. At this point, the most common scenario is the majority of the class buckles under the pressure, performing lamentably on the final exam. It is amazing that otherwise intelligent professors are taken aback by this repeated disaster, year after year, class after class. A smarter approach would be to front-load the course, tackling 60% of the work prior to the mid-term, including some challenging material. The correlation to managing an iterative project is to not waste precious time in the beginning solving non-problems and accomplishing trivial tasks. The most common reason for technical failure in startups: "They spent all their time doing the easy stuff."

Trap: Putting Your Head in the Sand

It is often tempting to say, "This is a delicate issue, a problem for which we need a lot of time to think. Let us postpone its resolution until later, which will give us more time to think about it." The project then embarks on all the easy tasks, never dedicating much attention to hard problems. When it comes to the point at which a solution is needed, hasty solutions and decisions are taken, or the project derails. You want to do just the opposite: tackle the hard stuff immediately. I sometimes say, "If a project must fail for some reason, let it fail as soon as possible, before we have expended all our time and money."

Trap: Forgetting About New Risks

You performed a risk analysis at the inception and used it for planning, but then forgot about risks that develop later in the project. And they come back to hurt you later. Risks should be re-evaluated constantly, on a weekly, if not monthly, basis. The original list of risks you developed was just tentative. It is only when the team starts doing concrete development (software first) that they will discover many other risks.

Clashes Because of Different Lifecycle Models

The manager of an iterative project will often see clashes between his environment and other groups such as top management, customers, and contractors, who have not adopted—or even understood the nature of—iterative development. They expect completed and frozen artifacts at key milestones; they do not want to review requirements in small installments; they are shocked by rework; and they do not understand the purpose or value of some ugly architectural prototype. They perceive iteration as just fumbling purposelessly, playing around with technology, developing code before specs are firm, and testing throwaway code.

At a minimum, make your intentions and plans clearly visible. If the iterative approach is only in your head and on a few whiteboards shared with your team, you will run into trouble later on.

The project manager must protect the team from external attacks and politics in order to prevent the outside world from disrupting or discouraging the team. He or she must act as a buffer. In order to be "the steady hand on the tiller," the project manager must build trust and credibility with the external community. Therefore, visibility and "tracking to plan" is still important, especially in light of "the plan" being somewhat unconventional in many people's eyes. In fact, it is actually more important.

Trap: Different Groups Operating on Their Own Schedules

It is better and easier to have all groups (or teams, or subcontractors) operating according to the same phase and iteration plan. Often project managers see some time optimization in fine-tuning the schedule of each individual team, each of which ends up having its own iteration schedule. When this happens, all the perceived benefits will be lost later and teams will be forced to synchronize to the slower group. As much as is feasible, put everybody on the same heartbeat.

Trap: Fixed-Price Bidding During Inception

Many projects are pushed into bidding for contractual development far too early, somewhere in the middle of inception. In an iterative development, the best point in time for all parties to do such bidding is at the LCA milestone (end of elaboration). There is no magic recipe here: it takes some negotiation and education of the stakeholders, showing the benefits of an iterative development, and eventually a two-step bidding process.

Accounting for Progress is Different

The traditional earned-value system to account for progress is different, since artifacts are not complete and frozen, but are reworked in several increments. If an artifact has a certain value in the earned value system, and you get credit for it at the first iteration in which you created it, then your assessment of progress is overly optimistic. If you only get credit when it becomes stable two or three iterations later, your measure of progress becomes depressingly pessimistic. So when using such an approach to monitor progress, artifacts must be decomposed in chunks; for example, initial document (40%), first revision (25%), second revision (20%), final document (15%). Each chunk must be allocated a value. You can then use the earned value system without having to complete each element.

An alternative would be to organize the earned value around the iterations themselves, and gauge the value from the evaluation criteria. Then the intermediate tracking points (usually monthly) reported in the Status Assessment would be built around the Iteration Plan. This requires a finer-grained tracking of artifacts than the traditional requirements spec, design spec, etc. because you are tracking the completion of various use cases, test cases, and so on.

As Walker Royce says, "A project manager should be more focused on measuring and monitoring changes: changes in requirements, in the design, in the code, than in counting pages of text and lines of code." (See "Further Reading" below.) And Joe Marasco adds, "Look out not only for change, but also for churn. Things that change multiple times to return to the same starting point are a symptom of deeper problems."

On the positive side, having concrete software that runs early can be used by the wise project manager to obtain some early credibility points. It can show off progress in a more meaningful fashion than completed and reviewed paperwork with hundreds of check boxes ticked off. Also, engineers prefer "demonstrations of how it works" to "documentation of how it should work." Demonstrate first, then document.

Deciding on Number, Duration, and Content of Iterations

What do we do first? The manager who is new to iterative development often has a hard time deciding on the content of iterations. Initially, this planning is driven by risk, technical and programmatic, and by criticality of the functions or features of the system under construction. (The RUP gives guidelines for deciding the number and duration of iterations.) The criteria also evolve throughout the lifecycle. In construction, planning is geared to completing certain features or certain subsystems; in transition, it is geared to fixing problems and increasing robustness and performance.

Trap: Pushing Too Much in the First Iteration

We talked above about not tackling the hard problems first. On the other hand, going too far in the opposite direction is also a recipe for failure. There is a tendency to want to address too many issues and cover too much ground in the first or first few iterations. This fails to acknowledge other factors: a team needs to be formed (trained), new techniques need to be learned, and new tools need to be acquired. And often, the problem domain is new to many of the developers. This often leads to a serious overrun of the first iteration, which may discredit the entire iterative approach. Or, the iteration is aborted— declared done when nothing runs—which is basically declaring "victory" at a point a t which none of the lessons may be drawn, missing most of the benefits of iterative development.

When in doubt, or when confronted with crisis, make it smaller somehow (this applies to the problem, the solution, the team). Remember that completeness is a late lifecycle concern. "Appropriate incompleteness" should be the manager's early lifecycle concern. If the first iteration contains too many goals, split it into two iterations, and then ruthlessly prioritize which objectives to attempt to achieve first.

It is better to shoot for a simpler, more conservative goal early in the project. Note we didn't say easy. Having a solid, acquired result early in the process will help build morale. Many projects that miss the first milestone never recover. Most that miss it by a lot are doomed despite subsequent heroic efforts. Plan to make sure you don't miss an early milestone by a lot.

Trap: Too Many Iterations

First, a project should not confuse the daily or weekly builds with iterations. Since there is fixed overhead in planning, monitoring and assessing an iteration, an organization that is unfamiliar with this approach should not attempt to iterate at a furious rate on its first project. The duration of an iteration should also take into consideration the size of the organization, its degree of geographic distribution, and the number of distinct organizations involved. Revisit our "six plus or minus three" rule of thumb.

Trap: Overlapping Iterations

Another very common trap is to make iterations overlap too much. Starting to plan the next iteration somewhere towards the last fifth of the current iteration, while attempting to have a significant overlap of activities (i.e., starting detailed analysis, designing and coding the next iteration before finishing the current one and learning from it) may look attractive when staring at a GANTT chart, but will lead to problems. Some people will not be committed to following up and completing their own contribution to the current iteration; they may not be very responsive to fixing things; or they will just decide to take any and all feedback into consideration only in the next iteration. Some parts of the software will not be ready to support the work that has been pushed forward, etc. Although it is possible to divert some manpower to perform work unrelated to the current iteration, this should be kept minimal and exceptional. This problem is often triggered by the narrow range of skills of some of the organization's members, or a very rigid organization: Joe is an analyst, and this is the only thing he can or wants to do; he does not want to participate in design, implementation, or test. Another negative example: A large command and control project has its iterations so overlapped that they are basically all running in parallel at some point in time, requiring management to split the entire staff across iterations, with no hope of feeding back lessons learned from the earlier ones to the later ones.

See Figure 3 for a few common unproductive iteration patterns.

Teams not synchronized -> chaos

First iteration too loaded -> panic

Too much overlap -> no feedback loop

Figure 3: Some Dangerous Iteration Patterns

A Good Project Manager and a Good Architect

To succeed, a software project needs both a good project manager and a good architect. The best possible management and iterative development will not lead to a successful product without a good architecture. Conversely, a fantastic architecture will fail lamentably if the project is not well managed. It is therefore a matter of balance, and focusing solely on project management will not lead to success. The project manager cannot simply ignore architecture: it takes both architecture expertise and domain expertise to determine the 20% of things that should go into early iterations.

Trap: Use the Same person as the PM and the Architect

Using the same person as project manager *and* architect will work only on small projects (5-10 people). For larger endeavors, having the same person play the role of both project manager and architect will usually end with the project neither properly managed, nor well architected. First, the roles require different skill sets. Secondly, the roles, in and of themselves, are more than a full-time job. Therefore, the project manager and architect must coordinate daily, communicate with one another, and

compromise. The roles are akin to that of a movie director and a movie producer. Both work toward a common goal but are responsible for totally different aspects of the undertaking. When the same person plays two roles, the project rarely succeeds.

Conclusion

At this stage, you may feel discouraged: so many problems ahead, so many traps to fall into. If it is so hard to plan and execute an iterative development, why bother? Rejoice; there are recipes and techniques to systematically address all these issues, and the payoffs are greater than the inconvenience in terms of achieving reliably higher quality software products. Some key themes: "Attack the risks actively or they will attack you." (From Tom Gilb's book, listed under *References and Further Reading*.) Software comes first. Acknowledge scrap and rework. Choose a project manager and an architect to work together. Exploit the benefits of iterative development.

The waterfall model made it easy on the manager and difficult for the engineering team. Iterative development is much more aligned with how software engineers work, but at some cost in management complexity. Given that most teams have a 5-to-1 (or higher) ratio of engineers to managers, this is a great tradeoff.

Although iterative development is harder than traditional approaches the first time you do it, there is a real long-term payoff. Once you get the hang of doing it well, you will find that you have become a much more capable manager, and you will find it easier and easier to manage larger, more complex projects. Once you can get an entire team to understand and think iteratively, the method scales far better than traditional approaches.

Note: John Smith, Dean Leffingwell, Joe Marasco, and Walker Royce helped me write this article by sharing their experiences in iterative project management. Part of this article is included in Chapter 6 of our colleague Gerhard Versteegen's new book on software development (see References and Further Reading below).

About the Author

Philippe Kruchten joined Rational Software in 1987 and is currently a Rational Fellow, based in Vancouver, B.C. Formerly the Director and General Manager of the Process Business Unit, he has led development of the Rational Unified Process. In addition to focusing on software architecture and design, he is keenly interested in software engineering practices and the development process. He holds degrees in mechanical engineering and a doctorate in computer science from French institutions.

References and Further Reading

1. *Rational Unified Process 2000*, Rational Software, Cupertino, Ca., 2000.
2. Barry W. Boehm, "A Spiral Model of Software Development and Enhancement," *Computer*, May 1988, IEEE, pp.61-72.
3. Tom Gilb, *Principles of Software Engineering Management*, Addison-Wesley, 1988.
4. Philippe Kruchten, *The Rational Unified Process—An Introduction*, Addison Wesley Longman, 1999.
5. Walker Royce, *Software Project Management—A Unified Approach*, Addison Wesley Longman, 1999.
6. Gerhard Versteegen, *Projektmanagement mit dem Rational Unified Process*, Springer-Velag, Berlin, 2000.

Rational®
the software development company

Dual Headquarters:

Rational Software
18880 Homestead Road
Cupertino, CA 95014
Tel: (408) 863-9900

Rational Software
20 Maguire Road
Lexington, MA 02421
Tel: (781) 676-2400

Toll-free: (800) 728-1212
E-mail: info@rational.com
Web: www.rational.com
International Locations: www.rational.com/worldwide