From Waterfall to Iterative Lifecycle: A tough transition for project managers

Philippe Kruchten, Rational Software

Although the Rational Unified Process advocates an iterative or spiral approach to the software development lifecycle, do not believe for one second that the many benefits it 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 shall discuss some of the challenges of iterative development from the perspective of the project manager. I will also describe some of the common "traps" or pitfalls that I have seen project managers fall into throughout my consulting experience, in addition to reports and war stories from my colleagues at Rational Software.

More Planning Work

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 the end-users because you have had time to evolve requirements, a design, and an implementation.

Iterative development actually involves much more planning and is therefore likely to put more of 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, code) will have to be modified and reviewed and approved again and again at each revision. The changes of tactics or the changes in scope will force some continuous replanning. The 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 good grip on the requirements - which occurs roughly at the LCA milestone - you cannot build any realistic plan.*

So incorporate precision in planning commensurate with your knowledge of the activity, the artifact, or the iteration being planned. Short 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 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 is 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 interstate.

* LCA = LifeCyle Architecture = End of the Elaboration phase = the point in time when you need to baseline your architecture, your requriements, your plans, and where normally all your risks have been mitigated.

Planning the exact details of getting through Kansas - let alone the arrival in California - is unnecessary and silly: you may find that the road through Kansas is under repair and you need to find an alternate route. And so on...

Acknowledging Rework Up-Front

In a waterfall approach, too much rework comes in 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, 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 early. Also, in order to build executable prototypes, stubs and scaffolding will have to be built, to be later replaced 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.

However, 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: introduce yet a better technique, redo this and that, etc. The project manager has to be vigilant so as to not allow the rework of elements that are not broken, elements that are OK, that are 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 other ideas on how things should be 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 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, and we'll decide where to go later.

Iterative development does not mean perpetually fuzzy development. You do not just start designing stuff and coding it, just to keep the troops busy, or hoping that suddenly clear goals will emerge out of it. You still need to define clear goals, put them in writing, and get concurrence from all parties. Then you have to refine them, and expand them, and again get concurrence from all parties. 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 of 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 this team might actually pull it off. The good news is that the external perception of the project has shifted: on Monday, they would have been happy if anything would have been delivered, but on Tuesday they become concerned that everything won't be delivered. And that's the bad news. Somewhere between the first and second beta, all of a sudden you find yourself inundated with features that people are concerned will not make it into the first release. These become major issues. The project manager goes from worrying about delivering minimal acceptable functionality to a situation where every last requirement now becomes "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 are still the same number of things to do, and the same amount of time to do them in. 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 on all these requests, he actually puts the project schedule in danger again! It is at this point that he or she must continue to be ruthless and not succumb to new requests. Even trading something new for something taken out may increase risk at this point. Without vigilance, one can snatch defeat from the jaws of victory.

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 the specs right, complete, polished, and signed-off. In the iterative process, the software we develop comes first. The software architecture (i.e., the solution-space description) needs to drive the early lifecycle decisions.
Customers do not buy the specs; it is the software product that is the main focus of attention throughout, and both the specs and the software evolve in parallel. This focus on "software first" has some impact on the various teams: testers, for example, may have been used to getting complete, stable specs and are

given plenty of advanced notice to start testing, whereas in an iterative development, they have to get to work at once, with specs and requirements that are still evolving.

Trap: Focus on the wrong artifact.

"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 your butt by showing that you have failed despite the best possible management.
Similarly, you may have been bitten in a previous job by poor requirements management, and choose to focus on the best possible spec; it 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, the real unknowns, are pushed to the right in the planning process, for resolution in 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, not involving many stakeholders (testers, etc.), not involving hardware platforms, and not involving the 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 that of a university course in which the professor spends the first half of the semester on relatively basic concepts, giving the impression that this is an easy class and allowing the students to get good marks at the mid-term without too much effort. Then, in the closing weeks, a sudden acceleration takes place, wherein the professor tackles all the hard topics shortly before the final exam. The most common scenario at this point is that the majority of the class buckles under the weight, doing a lamentable job on the final exam. It is amazing that otherwise intelligent professors are surprised at this repeated disaster, year after year, class after class. A better approach is to front-load the course, doing sixty percent of the work before the mid-term, and tackling some of the challenging stuff early. Don't waste precious time in the beginning, solving the non-problems or doing the obvious or trivial stuff. 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, this will give us more time to think about it." The project then embarks on all the easy stuff, never dedicating much attention to the hard problem, and when it comes to the point where a solution is needed, hasty solutions and decisions are taken, or the project just 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 the time and money."

Trap: Forgetting about new risks.

So you did a risk analysis in the project's inception, and used it for planning. But then you forgot about risks that develop later in the project. They will come back to bite you later. Risks should be re-evaluated constantly - probably on a weekly basis, but at least monthly. The original list of risks you came with was just tentative. It is only when the team starts doing concrete development (software first) that they will find out many other risks.

Clashes of Lifecycle Models

The manager of an iterative project will often see clashes with his environment and other organizations: 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 a lot of trouble later on.
The project manager must protect the team from external attacks and politics, not letting the outside world disrupt or discourage the team. He or she must act as a buffer. However, in order to be "the steady hand on the tiller," the project manager must have built trust and credibility with the external community. So visibility and "tracking to plan" is still important; in light of "the plan" being somewhat unconventional in many people's eyes, it is actually more important.

Trap: Different groups operating on their own schedule.

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

Trap: Fixed-price bidding at inception.

Many projects are pushed into bidding for a 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 with 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 they are reworked in several increments. If an artifact has a certain value in the earned-value system, and if you get credit for it at the first iteration where you create it, then your 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%), and 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-grain tracking of artifacts than the traditional requirements spec, design spec, etc. because you're 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 counting pages of text and lines of code. And Joe Marasco adds: look out not only for change, but for churn. Things that change multiple times to return to the same starting point are symptoms of deeper problems.
On the positive side, having concrete software that runs early can be used by the wise project manager to get some early credibility points, to show off progress in a more meaningful fashion than just completed and reviewed paperwork and hundreds of check-boxes ticked off. Also, engineers prefer "demonstrations of how it works" rather than "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 about the content of iterations. Initially, this planning is driven by risk, technicality and programmability, and by the level of importance of the functions or features of the system under construction. (The Rational Unified Process gives guidelines for deciding number and duration of iteration.) The criteria also evolve throughout the lifecycle: in construction, planning is geared to completing certain features or certain subsystems in transition, fixing problems, and increasing robustness and performance.

Trap: Pushing too much in the first iteration.

We talked earlier 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 few iterations. This fails to acknowledge other factors: a team needs to

be formed (trained), new techniques need to be learned, new tools must be acquired. Often, also, the problem domain is new for many of the developers. This can frequently lead 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 where none of the lessons can 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 that I 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. But also, since there is a 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. Yes, indeed, you will need to start planning the next iteration somewhere towards the last one-fifth of the current iteration, but attempting to have a significant overlap of activities, i.e., starting detailed analysis, design, and coding of the next one before you have finished the current one and gotten the lessons out of it may look attractive when only staring at a GANTT chart, but will lead to all kinds of problems. Some people will not be committed to follow up and complete their own contribution to the current iteration; they may not be very responsive to fixing things up or will just decide that they will 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 a small part of the manpower to do some 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 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 testing. Another negative example: a large command and control project had its iterations so overlapped that they were basically all running in parallel at some point in time. This required management to split all the staff across all iterations, thus there was no hope of applying lessons learned from earlier iterations towards later ones. See Figure 1 for a few common "anti-patterns" of iteration planning.

FIGURE 1: Some dangerous iteration patterns.

A Good Product Manager and A Good Architect
To succeed, a software project needs both a good project manager and a good architect. Having the best management possible and using iterative development will not lead to a successful product without a good architecture, and vice versa: 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 twenty percent of the stuff that should go into early iterations.

Trap: Use the same person as the PM and the architect.
This will work only on the small projects, the 5- to 10-person projects. For any larger endeavor, 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, it requires different skill sets. Second, each one of them is more than a full time job. This requires that the two coordinate a lot - daily - and that they listen to each other and compromise. They are a little bit like the director and the producer of a movie, aiming at a common goal but responsible for totally different aspects of the undertaking. We have seen many projects where for various reasons - history, power, ego, distrust - the two roles are played by the same person. These projects rarely succeed.

Conclusion

At this stage, I know you may be discouraged: so many problems ahead, so many traps into which to fall. 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" (Tom Gilb), software comes first, acknowledge scrap and rework, choose a project manager and an architect to work together, and 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 team have a five-to-one (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 and more complex projects. Once you can get an entire team to understand and think iteratively, the method scales much better than traditional approaches.

Acknowledgments

Thank you to my Rational colleagues John Smith, Dean Leffingwell, Joe Marasco, and Walker Royce for sharing their experience in iterative project management and helping me with this article, which is now a chapter in a new book on software project management by our colleague Gerhard Versteegen, in Munich.

Further Reading

Gilb, Tom. *Principles of Software Engineering Management,* Addison-Wesley, 1988.
Rational Unified Process 5.5, Rational Software, 1999.
Royce, Walker. *Software Project Management - A Unified Approach,* Addison-Wesley-Longman, 1999.