How to Fail with the Rational Unified Process: Seven Steps to Pain and Suffering

Craig Larman
Chief Scientist, Valtech USA
craig@craiglarman.com

Philippe Kruchten
Rational Fellow, Rational Software Canada
pbk@rational.com

Kurt Bittner
General Manager, Process and Project Management Business Unit, Rational Software
kbittner@rational.com

Abstract: The Rational Unified Process provides a valuable framework for approaching the business of developing software. As a framework, however, it must be adapted to the needs of each project team and their circumstances; it is intended to be applied in a light and agile style, and not adopted as a one-size-fits-all process. This article shares a number of common pitfalls experienced by teams attempting to adapt the Rational Unified Process to their needs, presented with a little tongue-in-cheek.

The Rational Unified Process (RUP®) [1] [2] has emerged as a popular *de facto* standard modern software development process—we feel with good reason. It combines recognized best practices such as adaptive, iterative, and risk-driven development; has been developed by world-class leaders with experience in both small and large systems development; is flexible in its application and extension; and has been coherently documented in both print and the online RUP product.

Yet, there are factors inhibiting the successful adoption of the RUP, leading to far less than optimal results. There are patterns in these failures if you wish to learn from them; to that end, if your goal is spectacular failure with the RUP, we recommend the following steps.

Step 1: Superimpose "Waterfall" Thinking

Does your development process look something like this?

1. Try to define and stabilize most of the requirements; sign-off on them.
2. Do detailed design, based on the requirements.
3. Implement, based on the design.
4. Integration, system testing, and deployment.

This is an example of a linear, sequential “waterfall” lifecycle, and is the first, best, and most common strategy for total RUP failure. If your process feels anything like the above, you know you have successfully *not* adopted the RUP.

It has been argued [3] that the waterfall lifecycle as originally intended would not lead to the degree of failure it has engendered, and it is rather due to misunderstanding that it has been unskillfully applied. Point taken, but we are discussing the waterfall as it is commonly applied or misapplied today.

With respect to failure levels, note that an analysis of project success and failure in 1994 [4], when most development was purely *ad hoc* or based on waterfall process practices (as today, we would argue), estimated that only 70% of software projects were completed, that over 50% cost at least twice their original estimate, and that $81 billion US was spent on cancelled projects in the USA alone. This is an extraordinary problem rate; business as usual is not working.

To their credit, the US Department of Defense, a major contractor of software development services, which originally promoted waterfall processes and attitudes, upon observing so much failure with the approach, has not only dropped this requirement (removed in 1988 in STD-2167A), but in a 1994 report started to actively encourage more modern iterative lifecycles to replace the waterfall [5]. Yet, inertia is a problem and there remains superimposition of “waterfall attitudes” on to projects—“Congratulations, the DoD will use an iterative process for this project. Now, step one: Please submit the complete requirements so we can sign off on them. Then we’ll nail down the design, …”

The waterfall-inspired processes were a reaction to prior 1960s *ad hoc* approaches to developing software. In contrast to the prior lack of structure, it was a rational response; indeed, waterfall methods of this era were called *structured* to emphasize this point. The approach drew inspiration—as is usually the case in new frontiers—from what was known and familiar; that is, from engineering and construction in other domains, such as building—do the requirements, then do the design, then construct. Unfortunately, the approach was adopted and taught without critical research into its real suitability for software development, and several generations of students and educators simply learned and repeated the advice.

Some things should be built like buildings, like, well…buildings. However, it turns out that software is not usually one of them.

There are a number of reasons for this. The most compelling is the wrong assumption that most requirements can be defined in the first project phase. The research deconstructing this myth includes work by Capers Jones [6]. As illustrated in Figure 1, in this very large study of 6,700 projects, creeping requirements—those not anticipated near the start—are a very significant fact of software development life, ranging from around 25% on average projects up to 50% on larger ones. Boehm and Papaccio present similar research-based conclusions in [7].

© Copyright Valtech Technologies & Rational Software, 2001

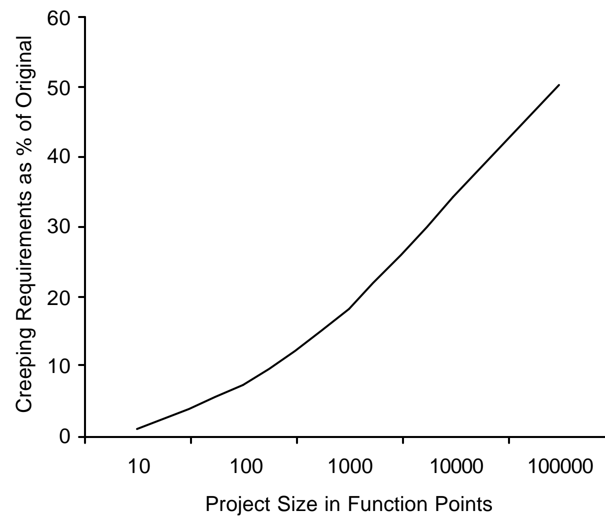


Figure 1 Changing Requirements are the Norm

Waterfall attitudes, which struggle against (or simply deny) this fact by assuming requirements and designs can—with enough effort and skill—be correctly specified and frozen, are starkly incongruous with project reality.

There are varied reasons for this inability in software development to pin down the requirements before design and implementation. They include:

- the unparalleled flexibility and options available for software;
- psychological and organizational forces which impede the ability to fully, accurately, and appropriately speculatively define a software system (without feedback-adaptation cycles);
- imprecise languages of specification;
- imperfect designs and implementations;
- fast-changing market forces, which motivate changes, and so forth.

Whatever the reasons, the skillful response is not to "fight change" and try harder to pin requirements down (as was the waterfall response), but rather, as Kent Beck has evocatively phrased it [8], to *embrace change* as a core driver in the process.

Consequently, in the RUP, development proceeds instead via a series of iterations, each of which is "timeboxed" to a fixed duration (such as exactly four weeks), and which ends in a stable internal release of a subset of the final system. *Timeboxing* is a key concept in iterative development: it means to fix the end date of the iteration, and not normally allow date slippage. If all the objectives can't be met, requirements are removed from the iteration, rather than expanding the iteration duration.

Within an iteration, there is something like a miniwaterfall. A small set of requirements is chosen and more fully analyzed (perhaps prioritized by high risk or business value); a few days are spent on design; and then the team quickly starts implementation, integration, and realistic system and stress testing for a portion of the system. The end of each iteration results in a running partial system, which generates feedback, that leads to adaptation of the requirements and design in future iterations. Over time, these feedback-adaptation iterative cycles reveal an appropriate set of requirements and a robust, proven design and implementation. Note that a sequential waterfall approach is being applied at the scale of weeks, not many months or years,

© Copyright Valtech Technologies & Rational Software, 2001

and that there is a built-in mechanism for feedback and adaptation. At the time scale of a few weeks, a sequential lifecycle can work; however, it breaks down as the length increases.

Since the reality is that the requirements do change during design and implementation—significantly—the waterfall model postpones dealing with this important risk (and therefore reality) until late in the lifecycle, often until it is too late to do much about it. A waterfall lifecycle project may be as failure-prone as the prior *ad hoc* approaches it replaced, but it goes about it in a more orderly way (the project can be right on track up until a catastrophic failure). Yet, because of the perceived lack a logical alternative, many people have felt that they have had no choice but to continue to use a model based on flawed assumptions. Organizations have become inured to the ongoing obvious failures or difficulties in software development that is based upon waterfall principles, without really challenging the underlying assumptions. Even today, more than a decade after the warning was first sounded, there are consulting companies, managers, teachers, and writers who promote the waterfall lifecycle or attitudes as skillful, which is unfortunate.

Thus, waterfall thinking pervades our development beliefs in systemic and sometimes subtle ways, and as a consequence, it is especially common among RUP adopters to superimpose waterfall values and practices on to the RUP.

To be clear on common waterfall values and practices:

- First, do most of the requirements, making the implicit assumption that requirements can be well defined by the users of a product they have not seen before.
- Second, do the detailed design on the assumption that the solution to a poorly-defined problem can be defined with precision.
- Third, implement even though the design is unproven and often not provable.
- Fourth, integrate, test, and deploy.

Within the scope of these activities:

- Try thoroughly develop an artifact, such as the requirements or design, before moving on. Try hard to get them complete and stabilized. Polish carefully.
- A later discovery that forces significant change in the requirements, models, or design that we tried hard to get correct in the first place, is a sign of some failure. The solution is to try harder or be more skillful—polish more intensely—in the future in order to get the requirements or design closer to being complete and correct.
- It is proper to expect to be able to deliver a believable estimate and plan for a novel system based on new technology, quite early in the project. Failure to do so is a sign of lack of skill.

Many of us were incorrectly taught that the above practices and attitudes were skillful. However, they are not adopted in the RUP and other iterative processes (such as XP). They are not rejected out of a perverse desire to be contrary or novel, but out of recognition that the solution is not to fight change, but to embrace it, and make it a core driver in the process.

The fact that that we are misled by the waterfall approach into increasing the risk of failure (rather than reducing it) runs counter to our education and the old lore of software development. But we cannot keep customers from changing the requirements by requiring them to “sign-off” on the specifications any more than we can alter the laws of physics. The reality is that if the requirements are wrong or the business changes, we need to develop the system in a flexible way that allows us to learn from experience as we go along. Overcoming the conscious or unconscious superimposition of waterfall values on the RUP and iterative development is one of the greatest challenges a project team faces. The most significant shift to truly adopting the RUP is not in technical skills such as doing use cases, object modeling, and so on, or in learning the many possible artifacts and activities that the RUP offers (all of which are optional), but in fundamental *attitude and expectation* adjustments related to waterfall thinking.

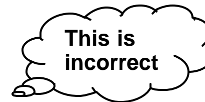
© Copyright Valtech Technologies & Rational Software, 2001

The following sections describe how superimposing waterfall values and practices can be applied to help fail with the RUP.

Superimpose Waterfall Phases on the RUP Phases

A RUP development cycle is composed of four phases: inception, elaboration, construction, and transition. The most common strategy for RUP failure is to in some way consider their definition as similar to the waterfall phases. That is,

1. Inception—do most of the requirements
2. Elaboration—do the detailed design and models
3. Construction—implement
4. Transition—integration, system test, deployment



The above description is quite incorrect, but a common misinterpretation—either consciously or unconsciously—of the RUP phases. It is not hard to find this misinterpretation in various books, articles, presentations, and consulting "advice" on the RUP. One sign of this kind of misconception is to see the phase names turned into verbs ("are we done *incepting* yet?") or states of requirements ("have all the key use cases been *elaborated* yet?").

A careful explanation of the RUP phases is not within the scope of this article, but a brief and more accurate description of the phases is:

1. *Inception*—develop the business case for the system; this requires us to explore a small but significant set of the requirements (perhaps 10%) in order to obtain an order of magnitude sense of the scope, the key risks, and to decide whether to fund the elaboration phase.

2. *Elaboration*—iteratively build the core architecture and resolve the technical risks of the project. When we say build the architecture we mean really program, integrate, and test it—this is not a "paper" exercise or throw-away prototyping. In order to do this we may have to iteratively explore most of the requirements in detail (perhaps 80%) while in parallel implementing the core risky parts of the system. The requirements may significantly change throughout this phase, via feedback-adaptation cycles, in response to regularly evaluating partial implementations. Note that in contrast to classic waterfall-style requirements definition, the majority of the requirements are refined in *parallel* with developing the core architecture, and informed from the feedback of that real development. We also need to make the decision whether to fund project completion.

3. *Construction*—iteratively build the elements not done in elaboration; iterative integration and quality assurance; prepare for deployment. Requirements change less in this phase, as most of the requirements instability was iteratively clarified in the prior elaboration phase.

4. *Transition*—complete beta testing, resolve issues, and deploy the system.

Define Iterations Too Long or Too Short

We know of some organizations "adopting" the RUP, when asked about their planned average iteration length, are advising six or even twelve months for an iteration. In the vast majority of cases, this is definitely undesirable. "Ideally, an iteration should span from two to six weeks" is the RUP rule of thumb.

The essence of iterative development and the RUP approach is to take small steps of commitment to a possibly imperfect implementation, rapidly do integration, QA, and testing, quickly obtain *feedback*, and then *adapt* the requirements, design, and implementation based upon that feedback. *Short steps, feedback,*

and adaptation are key ideas. Long iterations miss the point, and so provide an excellent strategy for failure with the RUP.

On the other hand, a project with 300 developers will not run smoothly with two-week iterations, due to the overhead of a large team. In order to achieve sufficient throughput, the iteration length will need to be perhaps even eight or ten weeks. Note that a smaller, temporarily independent, subsystem team may divide the macro-level project iteration of eight weeks into local micro-iterations of two weeks each.

Polish the Requirements before Design

As mentioned, the idea that requirements can be significantly defined and stabilized in the absence of feedback runs counter to software development (and indeed, common human) experience. Think about buying clothing the way that the waterfall method would have us "buy" software: we would have to describe exactly what we want without ever seeing what we are getting; we could not even try the clothing on first to see how it looks. Every measurement would have to be specified precisely, months or years ahead of the actual time when we would receive the clothing. Heaven help us if we changed our minds or gained (or lost) a few pounds. Yet this is exactly how the waterfall method proceeds—specify most requirements before doing design or implementation.

If we wouldn't even buy clothes this way, what makes us think it works for software?

The reality is that we are bundles of inconsistency, a constant contradiction of needs and wants, and the choice of what is right is often a complex balance requiring us to try different approaches to see what works best. The fastest path to a solution is usually one that generates a number of *likely* approaches and then allows us to choose between these reasonable alternatives. An iterative approach allows for this kind of directed experimentation and creativity; waterfall approaches force us to be brilliant on our first try. It runs counter to human nature and—more pointedly—software project research.

Expect Believable Estimates and Detailed Plans Near the Start of a Project

Consider this scenario in the oil business. An oil executive invites you into her office and says, "I hear there's oil in *FooBarKhan*. I want it! Please head back to your office and take a week to write up a report telling me how much oil is there, how much and how long it will take to set up the oil fields, the resources we will need, and a plan with milestones."

In the oil business, the above scenario would be considered ludicrous. Why is it acceptable in the software industry? Oil people, who work rationally, know that such answers cannot be provided without a significant investment in exploratory drilling. Only after investigation to uncover the true (and originally hidden) nature of the reservoirs and geology is an estimate or a semblance of a plan possible.

And yet, with similar uncertainty and very high complexity, we expect to be able to estimate and plan software projects without investing in realistic "exploratory drilling."

If you've never done something before, it's hard to tell how long it will take. Sometimes this fact is used as a reason for rejecting an iterative approach—if you are doing something very novel it's hard to know how long it will take. In fear of this unknown, people sometimes retreat to the comfortable illusion of the waterfall method. But, just because we can make a plan that shows us marching along, completing milestones, it does not mean that we can really do it.

It is easy to put down precise dates on a schedule in absence of any real information about the amount of work to be done—this is the classic problem of planning where the plan becomes nearly useless as soon as it is published. If perfect planning is required to deliver a project, then our failure is assured.

The iterative approach allows for us to learn as we go; as iterations proceed we have more information about the *real* requirements, a better view of the real risks, and a better sense of our abilities in meeting the challenges of the project. In short, experience makes us better planners.

© Copyright Valtech Technologies & Rational Software, 2001

Sometimes the existence of a fixed-price delivery requirement is used as justification for a waterfall approach, as if this justifies making decisions in the absence of real information. The reality is that if you have never done a project of the kind on which you are asked to deliver a fixed bid, you are flying blind. If you want to be successful you had better pad your estimates as much as you can, hire as many people experienced in the problem domain as you can, and hope for the best. Making major decisions in the absence of information is always risky. With an iterative approach, you'll do no better on the initial estimate, but you'll know soon enough how well you are doing and you can start negotiating, setting expectations, and managing scope much earlier, when it may actually do some good.

A rational approach, advocated by the RUP, to fixed-price bids and contracts is a two-step strategy. In step one, contract for an initial short (for example, four week) fixed-length, fixed-price contract to do enough realistic investigation and exploratory programming to generate a more insightful scope, risk, and requirements specification. This improved specification is then used as the basis for bidding in step two (the complete development). The investment in step one investigation is not lost—the results will reduce the effort for the second step, and lead to a more realistic second (final) contract.

Step 2: Apply the RUP as a Heavy, Predictive Process

Some methodologists speak of processes which are heavy versus light, or predictive versus adaptive. A **heavy process** is a pejorative term meant to suggest one with the following qualities [9]:

- rigidity and control
- many activities and artifacts are created in a bureaucratic atmosphere
- lots of documents
- elaborate long-term detailed planning
- significant process overhead on top of the essential work
- process-oriented rather than people-oriented; treats people as pluggable parts in a mechanical method
- predictive rather than adaptive

A **predictive process** is one that attempts to plan and predict the activities and resource (people) allocations in detail over a relatively long time span, such as the majority of the project. It tends to be implicitly waterfall in its values, emphasizing defining the requirements first and a detailed design second, before implementation.

In contrast, a ***light*** or ***agile process*** is meant to suggest "lean and mean." with the following qualities:

- it is stripped of unnecessary bureaucratic process overhead, eliminating low-value or thoughtless document creation;
- it is focused on the realities of human nature in the context of work, making software development fun; and
- it is adaptive.

To encourage failure with the RUP, apply these heavy, predictive practices:

- Plan the entire iterative project in detail. Early on, define the number and dates of all iterations, and specify what will happen in each.
- Create most—or even better, all—of the RUP artifacts
- Add lots of project and process formal ceremony, and even better, several project committees.
- Strive for a mechanistic, clockwork feel in the project, with people as specialist cogs in the project machinery.

The RUP was not meant by its authors to be either heavy or predictive, and it is due to superimposition of incorrect process ideas or misunderstanding of the RUP, exacerbated by the large set of detailed process

© Copyright Valtech Technologies & Rational Software, 2001

documentation that the RUP product provides, that it could be so mischaracterized or poorly implemented. The authors of the RUP intended and encourage it to be applied in a light, agile, adaptive process spirit. Some examples of how this applies:

- A minimal set of RUP activities and artifacts should be created; just those that add real value. As the project proceeds, if any process overhead activity is not adding value, it is dropped.
- There are no *detailed* plans for all the iterations. There is a high-level plan (called the phase plan) that estimates the project end date and other major milestones, but it does not detail the exact path to those milestones. A detailed plan (called the iteration plan) only plans with greater detail one iteration in advance (for example, the following two week iteration). Detailed planning is done adaptively from iteration to iteration. This is not to imply it is impossible to speculatively allocate some work to future iterations, but to appreciate that it is speculation, which is increasingly unreliable the further you project into the future. Adaptation from the original plan is not seen as a failure in planning or execution, but a sensible adjustment to project realities. The macro-level milestone dates and objectives are known, but the path to each milestone is left to adaptively evolve.

Step 3: Avoid Object Technology Skills

The RUP is aimed at developing object-oriented systems, although most practices are applicable to other technologies. Over the years that we've observed (first- or second-hand) object technology (OT) projects fail or face serious problems, a common thread is not having people who are *really* skilled in thinking in objects, object design, design patterns, and object-oriented programming. If we don't have skilled OT developers, no amount of process is going to save the project. The presence of skilled OT developers is a paramount critical success factor, and adopting the RUP (or any process) is a relatively minor element in comparison.

As Grady Booch has said, "People are more important than process" [10]. Similarly, Alistair Cockburn, the author of *Surviving Object-oriented Projects*, succinctly phrased it as "Process is a second-order effect."

Therefore, to really fail with the RUP, ignore having or educating people with deep object skills. Meaningful OT education does not mean a one-week course in Java? technologies followed by a one-week course in object-oriented analysis and design (which is grossly inadequate for the vast and deep discipline of OT), but software engineers with something like eight weeks of intensive teacher-led education, spread out over six months, followed by twelve months of close mentoring by an expert.

This is not a call for software gurus or geniuses, but the recognition that object technology development skills are very non-trivial, and successful designing and programming in objects takes well-educated and mentored developers, not novices.

Lack of appreciation of the large investment required to hire or develop skilled object technologists, or cutting corners on this investment with wishful thinking that we can get by with less skilled engineers, is a superb way to guarantee failure.

Finally, we want to stress the need for intensive education and skill in object *design* and design patterns, not just object programming. To quote David Parnas [12]:

> *"Instead of teaching people that OO is a type of design, and giving them design principles, people have taught that OO is the use of a particular tool. We can write good or bad programs with any tool. Unless we teach people how to design, the language matters very little. The result is that people do bad designs with these languages and get very little value from them."*

Step 4: Undervalue Adaptive Iterative Development

The RUP documents introduce its key ideas and best practices:

- Manage requirements
- Develop iteratively
- Control changes
- Verify quality
- Architecture and component-centric design
- . . .

A person new to an iterative process and the RUP reads this list without a sense of the relative impact of these ideas, viewing the elements more or less as having equal weight. However, compared to the other ideas of the RUP, *iterative development* is in a league of its own in terms of profound impact on how we think about and practice software development. The list should be read like this:

- Manage requirements
- **Develop iteratively**
- Control changes
- Verify quality
- Architecture and component-centric design
- . . .

Properly understood, iterative development is like a revolution, if an organization is moving from waterfall values and practices. Indeed, if an organization is moving from those to the RUP, and does *not* experience a deep and perhaps traumatic transformation at many levels in how they think about developing software, this is probably a sign that they didn't "get" iterative development and truly adopt it.

There are several ways to ignore the implications of iterative development, some a reiteration of superimposing waterfall attitudes, and some with a different emphasis:

Embrace Rigid and Predictive Attitudes

That is, try to freeze the requirements and design, rather than embracing change. Try to plan all the future iterations, rather than adaptively adjusting.

Changing an organization's or a team's process is a BIG change—potentially disruptive, potentially invigorating, but always challenging. There is no way to ease into these kinds of changes—one must approach them directly and deliberately. Change is threatening to some people, and they will actively resist it even when it is good for them in the long run. Change also shifts the balance of power on a project away from the status quo and toward the "new thing." As a result, change must be planned and carefully orchestrated.

The waterfall model comes with its own special challenges with respect to change. Since the waterfall model evolved out of a response to uncontrolled change, it comes with an extraordinary bias against change. To some degree, the entire waterfall approach might be viewed as largely antagonistic to change—once requirements are "approved" (frozen), it requires an act of great effort to make changes to a requirement even when it has been found to be incorrect. It is as if the waterfall method says "well, you agreed that these were the requirements, so you better just be quiet and take what we give you."

In transitioning to an iterative lifecycle, change must be embraced at two levels: the organizational level and the project level. At the organizational level, one must embrace the change to a new way of doing things. At the project level, one must allow for feedback and continuous learning to say that it's acceptable to make a considered response to new information.

© Copyright Valtech Technologies & Rational Software, 2001

Pervert the Practice of Iterative Development

As Andy Grove observed, "what gets measured gets done." If you measure and reward people for "freezing requirements", "freezing design", and so forth, you're going to get a waterfall lifecycle no matter what you call the phases. If you don't embrace change, you're practicing the waterfall method no matter what you say.

Project managers seem to like the superficial certainty of the waterfall lifecycle because it has *apparent* stability and certainty: it's possible to say "it's December the 2nd and we're done analyzing all the requirements." The problem is that this kind of statement is delusional—you're only saying that you're done but you know at some level that this is not true—since nothing has been implemented yet, there is no real way to be sure that you are *really* done. It's only by testing that your assumptions about something being true or false can be assessed; it's only by implementing something that you know how much work it will really take.

On a project, we are all playing a game—we make assumptions, we have hunches about how things will be and what the solution should look like, and we pretend we are sure of what we are saying. But in truth we really don't know much for certain; we are bluffing a lot and hoping for the best. We are acting on imperfect information, and the results often show that.

The wise project manager takes this into account, building in checkpoints at which assumptions can be validated and undertaking activities to gather better information. The iterative approach is perfect for this: it allows information to be gathered while it can still be used to improve our plans and change direction. There's a saying that "no plan survives intact contact with the enemy." The wise project manager understands this and builds in opportunities to adjust the plans based on new information.

Probably the greatest failing of the waterfall approach as it is popularly practiced is that plans are made in isolation from reality and then must not be updated as new information is gathered. Changing the plan is considered a failure to foresee the future. But no one can foresee the future, and adjusting the plan is not so much a failure to predict as it is an opportunity to improve.

Don't Educate Stakeholders in the Implications of Iterative Development

Customers and management have become acculturated to the waterfall approach. Customers have come to expect that they can hand over requirements and wash their hands of the project until the completed product is delivered.

Management has become conditioned that they should be able to expect that perfect plans can be formulated on Day 1 of a project and then merely executed, and that reliable estimates can be generated with minimal investigation and virtually no meaningful exploratory programming or proof of concepts. If a project fails, they think, it is not because the plan was bad but because it was not followed, or because of unskillful planners or estimators. This kind of the thinking is the very sort of thing that gives management a bad name.

In order for iterative development to work, the customer must be involved. The heart and soul of iterative development is to adapt based upon feedback, rather than speculation. If the customer is not interested and actively engaged in making sure that the system that they want and *need* gets built, then they will get exactly what they deserve. The hardest thing about building software is building the *right* thing, getting the functions and usability solid. Developers are rarely domain experts and they need help understanding what is needed to solve the problem. They need help building the *right* system. The waterfall approach robs the team of meaningful interaction between the development team and the customer.

Customers have to be reeducated to appreciate that they need to take an active role in the definition of what needs to be developed, and they have to come to understand that there is an interplay between what a

system should do and inspiration from new technologies. Perhaps they ask for some kind of "traditional" system, unaware of the possibilities that wireless and mobile-networked devices could bring to the vision. The thing about technology is that it changes the nature of the problem. The Internet is a great example of this: it allows us to approach problems in new ways that are not always obvious to the people who are steeped in an existing business process.

The ideal approach is one in which we can start with an understanding of the problem that needs to be solved and explore that problem (its challenges and opportunities) in an evolutionary way, in a partnership between developers and the users/customer. An iterative approach allows for the kind of adaptive feedback that is essential to this style of development.

Over-Model and Create Many Low-Value Artifacts

The Rational Unified Process is a framework that solves a lot of different problems. It is unlikely, however, that you will encounter all of these problems. As a result, it is unlikely (and even undesirable) that you would use *all* of the Unified Process on your project.

Further, the idea of iterative development is to quickly start programming, when only partial requirements and designs are developed, in order to get realistic feedback, rather than speculate on the requirements and design.

Therefore, a skillful means to RUP demise and iterative failure is to create *all* the RUP artifacts, and try to elaborately, and with great specificity, detail each one. Draw at least twenty pages of UML diagrams before programming.

The RUP is like a medicine cabinet or a drug store. Most of us would not think of walking into a drug store, buying one of every kind of medicine and then taking those medicines. We know that we would get sick (or maybe worse); we have learned that we should only take what we need, and then sometimes only under the advice of an expert.

Some people suggest that the RUP is too big; this is like saying that there are too many medicines. There are many medicines that we do not have that many people need, even though most people do not need very many medicines at all (maybe we just have a few aches and pains now and then). The real issue is that people need to understand better how to know what they need.

A good way to look at how much of the RUP one should use is to use risk mitigation as a guide.

- Is there a risk that developers may not understand what the system should do? You may want to use some "requirements management."
- Is there some behavior that has complex control flow or is hard to understand? You may want to use "use cases."
- Is the technical solution to the system novel and complex? You may need "architecture."

You begin to see how to decide what to choose. Understand the problems you face, and then choose techniques (medicines) that will help reduce these problems.

Step 5: Avoid Mentors Who Understand Iterative Development

There is nothing quite like inexperience, misunderstanding, and ignorance to help a new practice really fail spectacularly. Therefore, on your first iterative RUP project, only use a team that has never done short timeboxed adaptive iterative development before—especially the project leaders. Look for people wedded to waterfall attitudes and practices. As an aside, it is also helpful to choose an architect for your first Java technology project who is not a Java expert and can't do Java programming.

Definitely avoid contracting with a mentor who knows how to do iterative development. If a knowledgeable mentor unfortunately joins the project, ensure they have a role with no influence or leadership, and that their advice is debated, viewed skeptically, and ultimately rejected by the project

leadership. Ridicule the mentor. If a mentor must join the team, search for one who misunderstands the RUP and subtly superimposes waterfall predictive attitudes, such as thinking that the elaboration phase is to clarify the models which are implemented in construction, or that the work for all iterations should be planned and allocated near the start of the project. They should be easy to find.

Step 6: Adopt the RUP in a Big Bang

If possible, introduce the RUP to the entire IT organization on a Monday, and demand all projects do it by Tuesday, but don't tell anyone the details. If that isn't possible, and education is forced on the organization, educate only the developers (rather than managers or the IT executives). It is especially amusing if management has waterfall attitudes and expectations, and the development team was taught instead to drop waterfall habits. If everyone must be educated in a RUP course, try to do so with at least a six-month lag before RUP adoption, so it will be completely forgotten, and make it one of those short one-day seminars from a RUP "educator" who superimposes waterfall ideas. If the RUP adoption must be applied right after education, then at least do it for the entire organization and switch everyone at the same time, across all projects. Avoid doing a small pilot project with a skilled mentor using a simple minimal set of RUP practices, learning from the experience, incrementally adopting it, and moving on to a second project with the experience of the first. If someone suggests that idea—fire them. At all costs, do not explain or solicit reasons for adopting the RUP—that could rationally justify the expense and alleviate the discomfort of change. Identify RUP/iterative skeptics and waterfall proponents, and put them in charge of the adoption project.

Step 7: Take Advice from Misinformed Sources

Even among so-called experts who do books, articles, teaching, or speeches on the RUP, there are some who don't really "get" iterative development and its fundamental rejection of waterfall values and predictive practices. The "RUP" information they present is not correct, and usually belies an underlying waterfall mentality. Don't take our word for it. Study the original RUP documents deeply and thoroughly, understand the true values of iterative development, and then compare for yourself.

We have recently seen misinformation purporting to explain the RUP in at least the following sources:

- A book on object-oriented project management
- A book on Java? server-side programming
- A lecture at a UML conference

The typical sign of misinformation is usually some variation of describing the four phases as follows:

1. Inception—do most of the requirements
2. Elaboration—do the detailed design and models
3. Construction—implement; translate the models into code
4. Transition—integration, quality assurance, deployment

We have also observed so-called RUP experts make *incorrect* statements such as:

- "In the RUP it is important to get 100% of the requirements defined before starting the design."
- "A good, typical iteration length in the RUP should be around 6 months."

To increase your chances of failing with the RUP, we recommend you accept what you read or hear about the RUP without discrimination. There are presently many sources that can help misinform you.

© Copyright Valtech Technologies & Rational Software, 2001

You Know You Didn't Understand the RUP When…

In order to ensure absolute misunderstanding and failure in RUP adoption, we provide the following checklist or score sheet. Of course, the more points scored, the more successful the RUP failure.

You know you didn't understand the RUP when ...

- You think that inception = requirements; elaboration = design; and construction = implementation.
- You think that the purpose of elaboration is to fully and carefully define models, which are translated into code during construction.
- You think that only prototypes are created in elaboration. In reality, the production-quality core of the risky architectural elements should be programmed in elaboration.
- You try to define most of the requirements before starting design or implementation.
- You try to define most of the design before starting implementation.
- A "long time" is spent doing requirements or design work before programming starts.
- An organization considers that a suitable iteration length is measured in months, rather than weeks.
- You think that the pre-programming phase of UML diagramming and design activities is a time to fully and accurately define designs and models in great detail, and of programming as a simple mechanical translation of these into code.
- You try to plan a project in detail from start to finish, allocating the work to each iteration; you try to speculatively predict all the iterations, and what will happen in each one.
- An organization wants believable plans and estimates for projects before they have entered the elaboration phase.
- An organization thinks that adopting the RUP means to do many of the possible activities and create many documents, and thinks of or experiences the RUP as a formal process with many steps to be followed.

Conclusion

We are confident that by following these seven steps, and applying the checklist of misunderstandings, your adoption of the RUP and iterative development will be a spectacular mess.

References

1. Kruchten, Philippe. *The Rational Unified Process—An Introduction, 2nd ed*. Reading, MA: Addison-Wesley (2000).

2. Jacobson, I., Booch, G., and Rumbaugh, J. *The Unified Software Development Process.* Reading, MA: Addison-Wesley.(1999)

3. Royce, Walker. *Software Project Management—A Unified Framework.* Reading, MA: Addison-Wesley.(2000)

4. Johnson, Jim. *Chaos: Charting the Seas of Information Technology*. Published Report. The Standish Group (date?)

5. Druffel, Larry and Heilmeyer, George. *Report of the Defense Science Board Task Force on Acquiring Defense Software Commercially*. (USA Department of Defense)

6. Jones, Caper. *Applied Software Measurement*. NY, NY: McGraw-Hill.

© Copyright Valtech Technologies & Rational Software, 2001

7. Boehm, B. and Papaccio, P. "Understanding and Controlling Software Costs," in *IEEE Transactions on Software Engineering*. Oct 1988.

8. Beck, K. *Extreme Programming Explained—Embrace Change*. Reading, MA: Addison-Wesley.(2000)

9. Fowler, Martin "Put Your Process on a Diet," in *Software Development*. Dec 2000. CMP Media.

10. Booch, G. *Object Solutions: Managing the Object-oriented Project*. Reading, MA: Addison-Wesley.(1996)

© Copyright Valtech Technologies & Rational Software, 2001