A technical discussion of RUP

A Comparison of the IBM Rational Unified Process and eXtreme Programming

John Smith

Table of Contents

Introduction

This paper compares the Rational Unified Process® (RUP), a process framework, refined over the years by IBM Rational. The RUP is widely used on a variety of software projects, from small to large, with Extreme Programming (XP), a software development approach gaining increasing recognition as an effective method for building smaller systems in an environment of changing requirements.

The comparison is directed at the nature and style of each as a process description, and their underlying structure, assumptions, and presentation. This paper also examines what the potential targets are for each and the consequences of their use.

Most processes have some common elements that make a systematic comparison possible. They require sequences or groups of activities, which are performed by roles (usually played by people working as individuals or teams) to generate artifacts or work products, some or all of which are delivered to a customer. Most processes also acknowledge that instances of the process will have a time dimension, with a beginning and an end, and interesting intermediate milestones that represent the completion of significant activities (or clusters of activities) and the production of associated artifacts. Accordingly, this paper explores the following dimensions of process and uses these to make the comparison:

- **Time and Effort Allocation** — discusses how each process is arranged over time and how the staffing effort allocation compares.
- **Artifacts** — compares the work products; those *things produced* in the course of an XP and RUP based project.
- **Activities** — discusses the way in which each process says its artifacts should be produced.
- **Disciplines** — compares the way in which XP and RUP delineate the major areas of concern in software engineering.
- **Roles** — explores the differences between roles (that perform activities) in the RUP and roles that are closer to positions within a team in XP.

The motivation for this paper is to clarify the relative positions of RUP and XP in the process spectrum to see what each can offer the other and to dispel the notion that XP is a *lightweight*, and therefore desirable, alternative to the *heavyweight* RUP.

The main sources of XP information used in writing this paper were three books in the Addison-Wesley XP Series:

- Extreme Programming Explained [Beck00]
- Extreme Programming Installed [Jeffries01]
- Planning Extreme Programming [Beck01]

These books were chosen because they are the most obvious repositories of XP information and many interested readers or potential users of XP will start with them. Other books are planned, but these were unavailable at the time of writing this paper. The source for RUP information is the RUP product itself from IBM Rational.

This paper is not intended to be a tutorial on either XP or RUP, so it will be more easily read by those who already have some acquaintance with these approaches or who have access to some reference material; for example, [Beck00] in the case of XP and [Kruchten00] in the case of RUP.

Time and Effort Allocation

RUP deals with projects (to develop software), the lifespan of which are divided into phases named *Inception, Elaboration, Construction,* and *Transition*. Each phase is further split into iterations and each iteration may require several builds.

For most projects in RUP, the duration of an iteration will be between two weeks and six months[1], and the number of iterations in a project's lifetime will be between three and nine. So at these default settings, RUP covers projects ranging in duration from six weeks to 54 months.

XP's iterations are about two weeks in duration. XP's releases are two months (or slightly longer); they are defined by the customer and released to the customer. In duration, XP's releases are like RUP's iterations — during Elaboration or Construction[2] at least, for projects that are appropriate for XP; that is, for those with a maximum team size of, let's say, 10[3], that are based on an established architectural baseline.[4]

To show this, let's assume an XP maximum team size of 10, discover the maximum sized project that is reasonable for such a team, and then see how RUP would handle projects of this size and smaller. If we use COCOMO II[5] as our estimation model, it predicts that a team of 10 is usually associated with a project having a maximum duration around 15 months. A project that lasts longer than 15 months would usually employ a team larger than 10. You could build larger systems with just 10 people if you were prepared to give them a longer schedule, usually though, schedule is important enough that you will plan to add more staff if you can do so effectively. This example project would deliver about 40,000 to 50,000 source lines of code (slocs) (800-1000 function points[6] in Java) from scratch.

According to the model, this is the largest project you should attempt with an XP-sized team if you want it done expeditiously. Also, perhaps there are other reasons why a small team cannot effectively build a large system; for example, because of loss of familiarity over time with code already built. (What a large team can do in parallel, a small team would have to do serially. So, when the time comes for integration testing and debugging, the small team will have to revisit code it wrote considerably earlier.)

If we look at a set of projects under this size, we see a reasonable mapping between XP's releases and RUP's iterations. The following examples illustrate how they might be planned in RUP. The numbers are indicative only, but are reasonable for projects of these sizes. They include effort and time for the preparation of all required RUP artifacts that are deliverable in addition to the code, such as user guides, installation and operation information, and so forth.

---

[1] The expected duration of an iteration in an e-business development may be shorter — more likely in the range two to six weeks.

[2] Iterations during Inception are usually shorter. Note also that the phase model for RUP accommodates a wide range of shapes, so that in a long transition phase you could envisage a sequence of iterations, each culminating in a delivery of software to the customer, also at two-month intervals, but the entry into transition means that the architecture is completely stable, and the changes contemplated are largely adaptive, perfective, and corrective.

[3] See [Beck00], which says that you probably couldn't run an XP project with 20 programmers, but that 10 is "definitely doable".

[4] XP is acutely focused on delivering direct business value to the customer, primarily as function. XP gives little direct consideration to architecture, allowing this to emerge over a series of refactorings of the software, as functions are added. If the architecture of the solution is not already well established, there is a risk that this will lead to a series of severe breakages, as functions are added that invalidate previous assumptions (and ad hoc solutions) and create problems that are not amenable to local refactoring.

[5] COCOMO II is a rework of the classic COCOMO software cost estimation model originally developed by Dr. Barry Boehm. It's calibrated for projects as small as 2,000 slocs. See [Boehm00].

[6] Function points are a source code-independent measure of software size, obtained by quantifying the functionality provided to the user, based solely on logical design and functional specifications. This definition was taken from the International Function Point Users Group Web site, at http://www.ifpug.org/.

Examples of projects amenable to XP

Example 1 illustrates a very small project consisting of 5,000 lines of new Java code, requiring about 12 person-months over 7 months elapsed time.[7]

Example 1

	Inception	Elaboration	Construction	Transition
Staff	1	1.5	2	2
Duration in weeks	3	6	18	3
Number of iterations (and duration of each in weeks)	1 (3)	1 (6)	3 (6)	1 (3)

Example 2 looks at a small project of 10,000 lines of new Java code, requiring about 27 person-months over 8 months elapsed time.

Example 2

	Inception	Elaboration	Construction	Transition
Staff	1.5	2.5	4	3
Duration in weeks	4	7	20	4
Number of iterations (and duration of each in weeks)	1 (4)	1 (7)	3 (7)	1 (4)

Example 3 illustrates a medium project of 40,000 lines of new Java code, requiring about 115 person-months over 15 months elapsed time.

Example 3

	Inception	Elaboration	Construction	Transition
Staff	3	5	10	8
Duration in weeks	6	16	36	6
Number of iterations (and duration of each in weeks)	1 (6)	2 (8)	4 (9)	1 (6)

Within these bounds (which cover quite a breadth of development), RUP iterations map very closely with XP's releases, in both intent and duration.

A large system development not suitable for XP

In very large developments—which RUP will handle, but XP will not—RUP iterations are much longer. Example 4 shows a project of 1,500,000 lines of new Java code, requiring 4,600 staff-months over 45 months elapsed time.[8]

[7] This may seem overly long, but observe that it covers the entire lifecycle; from a standing start, with essentially no staff and no defined requirements (only the germ of an idea), to project acceptance and closedown. It's also an output of the COCOMO II model, which allows more schedule compression, but with the penalty of increased cost and risk. However, according to the schedule in the table, something with useful functionality could be available as early as three and a half months after the start of the project (at the end of the first construction iteration).

Example 4

	Inception	Elaboration	Construction	Transition
Staff	35	70	140	100
Duration in weeks	20	50	100	20
Number of iterations (and duration of each in weeks)	2 (10)	2 (25)	3 (33)	2 (10)

Clearly, in this example the staff is not organized as a single, monolithic team—the project is composed of several teams each working on subsystems, which may be composed of subsystems so that, at lower levels, there will be planning at a similar scale to that for an XP-amenable sized project. However, this project is intended to represent an integrated system so that, at the top level, planning is required for iterations at a scale (33 weeks) that is well beyond the dictates of XP. These iterations are of this duration because of the planning inertia of an integrated project of this size. RUP's within-iteration planning is done through integration build plans (which may exist at system and subsystem levels) and these will be constructed at a scale closer to XP's releases (for this very large system) and at the lowest level at a scale close to XP's iterations (which are required to be about two weeks), particularly in late elaboration and construction.

So, returning to the smaller system examples, RUP's iterations are more or less equivalent to XP's releases, and RUP's builds are more less equivalent to XP's iterations.

What do RUP's phases map to in XP?

Well, at least at first glance, RUP phases map to aspects of the XP cycle that are not well delineated. XP appears to be constructed, or at least described, to fit a steady beat of releases (harmonizing with a business cycle) containing iterations. There is recognition that there is a *project* of some kind that has to be bootstrapped into existence (see "Scoping a Project" in [Beck01]) and that the *big plan* constructed at this time will then be divided into releases, which, in turn, will be divided into iterations. [Beck01] says that one thing that the big plan did was help us decide that it wasn't stupid to invest in the project.

So, there is a time in XP before the cycle of releases and iterations begins, where work is occurring on a project to decide whether it's feasible and how much it will cost, and, to do this, some coarse-grained idea of the required functionality has to be constructed. This is what RUP calls the inception phase. You are given the impression in XP that this should be done quickly; very quickly. In RUP, we say that it takes as long as is prudent—depending on the inherent risks. Even in XP, you read about the following things that have to be done at this time (by a small number of people):

- some requirements elicitation (write some "big stories")
- some planning
- some negotiating with the customer (setting expectations)
- factoring in any business constraints

It's easy to see that it may take a few days to go from a cold start to having an agreed Vision (this is what the "big stories" are called in RUP) and Business Case (budget and return on investment (ROI) that justify the project), and to taking a first cut at the Software Development Plan.

In RUP, as in XP, you do enough exploring and planning at project start to get going after you've established that you have a good probability of success. As previously shown in Example 2, RUP planning suggests an investment (in effort) of about $12,000 in inception to justify the expenditure of about $250,000. This will vary: in some well-trodden problem areas, you will not need to do as much; for example, perhaps you are not starting cold. On other occasions, you will spend more starting from scratch in an unfamiliar domain that requires some degree of research and development.

---

[8] You might argue that you should never characterize a project in this way; that a project of this size should be split into many smaller projects (each of which might be amenable to XP). Of course, projects of this size are constructed as systems of subsystems (or for extremely large projects as systems of systems), but when these subsystems or systems need to be integrated into a single product, then you will need to be concerned about architecture and, in particular, about interfaces between aggregates of software and the teams that produce them. XP in its present form does not deal with these issues.

After its equivalent of inception, XP moves straight into planning the first release. RUP says that the elaboration phase follows inception, and in the elaboration iterations, the architecture of the solution is stabilized. In apparent contrast, XP suggests that release planning be function-oriented, so that all releases deliver business value to the customer. XP is quite explicit in its disdain for what it calls infrastructure planning: just enough is to be done to support the functionality selected for that release. More infrastructure is added later as required, and, if the system breaks because earlier infrastructure decisions are invalidated by later additions of function, then the code is refactored to make it work.[9] The idea is not to spend a lot of time building something that does not deliver value to the customer.

In comparison, a couple of points need to be made about the RUP:

- ***RUP's concept of architecture is not simply infrastructure***

  To quote from RUP, "…the architecture of a software system (at a given point) is the organization or structure of the system's significant components interacting through interfaces, with components composed of successively smaller components and interfaces."

  So, architecture addresses the whole solution—not just infrastructure—from a particular set of perspectives, at an appropriate level of abstraction.

- ***RUP's notion of executable architecture delivers business value to the customer***

  It enables the early demonstration of a solution that satisfies the customer's non-functional requirements (as well as, potentially, some key functional ones in the process).

  The reduction of risk in this way—and often the non-functional requirements are the risky ones—is itself of considerable business value to the customer.

Why do we believe that it's necessary to describe the process in this way? Because RUP is all about driving out risks, and we believe that there are classes of problems and systems that are simply not amenable to the XP approach of allowing the architecture to emerge from refactoring the code—typically larger, more complex ones. One risk is that in making local changes in refactoring (and the scope will, of necessity, be limited), local optimizations will be produced that, in the aggregate, lead to a badly suboptimal solution. This isn't just our assertion: Kent Beck says as much himself in [Beck00], in the chapter titled *Four Values*, under the heading *Courage*. The difficulty is that XP says little about the origin of the inspiration for architecturally significant changes—only that it may take courage[10] to apply them!

Then there are changes that are just hard to refactor. For example, to make a single-user, single-machine system into a multi-user, multi-processor system, you may have to redesign a lot of things and still finish up with a system that has many constraints.

RUP's emphasis on architecture is intended to address those cases in which the initial approach may be wrong and needs to be completely reconsidered. For smaller systems, this is less of a risk—refactoring can make a broader sweep over the system and there's less likelihood that the initial architecture was incorrect.

Note too that where there is little perceived risk (because of size, new technology, unfamiliarity, complexity, other exigencies of performance, reliability, safety, and so on) and the solution can be delivered in a well-understood existing framework, then the RUP elaboration phase (the phase where architectural issues are handled) may be quite brief (and may concern itself more with the elicitation, refinement, and demonstration of important functional requirements). That is, the RUP lifecycle collapses toward something approaching an XP lifecycle. When a problem could be addressed successfully by XP, the RUP Development Case will also yield something similarly 'lightweight' (although not exactly the same).

The construction phase in RUP is equivalent to XP's series of releases with the qualification that, if you were following RUP, you would have to deliver the results of each construction iteration to the customer to have the same effect as XP. XP does not really have an equivalent of RUP's transition phase, because each XP release is already in the hands of the customer. There is, at the end of an XP project, what [Beck00] calls project death (see the information under the heading *XP lifecycle* a bit later), where some of the same project closeout activities would occur.

---

[9] Note though that refactoring only results in local improvements: refactoring does not address the overall framework. If the overall framework is solving the wrong problem, then refactoring will not produce a correct solution. The only solution is to make radical changes or, in the extreme, start over. This has obvious budget and schedule risks.

[10] No one would dispute that courage is a virtue—and by implication one possessed by "good people". However, even the most courageous sometimes will need an organized way of working and a framework in which tough problems can be solved.

What do XP's phases map to in RUP?

XP's phases (which apply to both XP's releases and iterations) are exploration, commitment, and steering. At the release level, these align with particular collections of activities (called workflow details in RUP) in the RUP Project Management discipline. These are Plan for Next Iteration (mapping to exploration and commitment) and Monitor and Control Project (mapping to steering). This is illustrated in the following sections.

XP phases

In [Beck00], there is a chapter on the lifecycle of an ideal XP project, and the first level headings are **exploration**, **planning**, **iterations to first release**, **'productionizing'**, **maintenance**, and **death**. These appear to be the top-level phases, but it turns out this is not quite accurate. An XP project will be bounded by an *initial* exploratory phase and "death" when the project is closed out. But the major beat is the release and *each release has an exploration phase.* So the exploration phase that brackets the project (and leads into the first release) is a special case in that there is an initial gate to pass (see *Scoping a Project* in [Beck01]) at which a decision is made on the wisdom of continuing at all. Both XP releases and iterations have the three phases—exploration, commitment, and steering. "Maintenance" really characterizes the nature of the XP project after the first release.

XP lifecycle

Overall, the XP lifecycle for a project with seven releases over 15 months might look something like Figure 1.

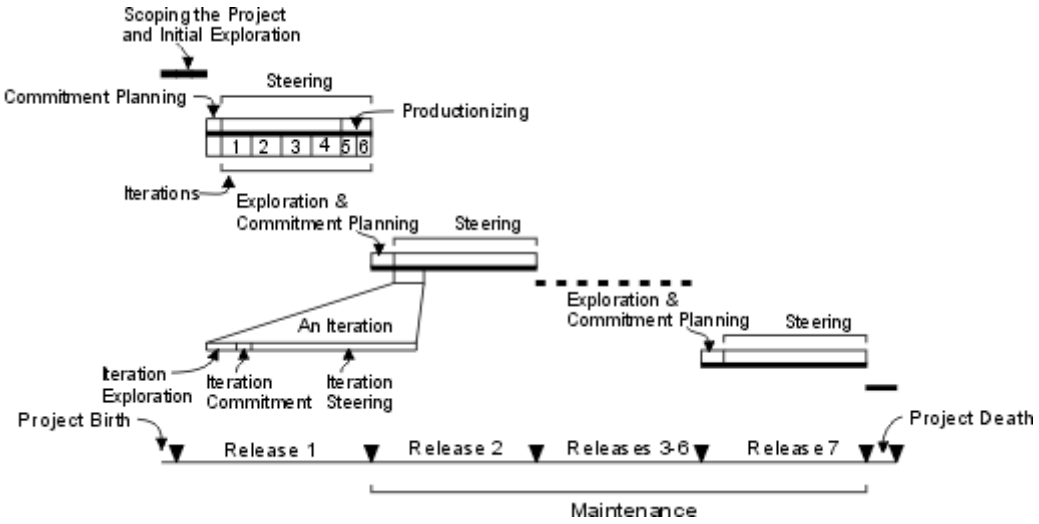

Figure 1

Each release after the first (which is three months) is about two months in duration, and each iteration is about two weeks, except in the later stages of a release ("productionizing", as it is called in [Beck00]) when the pace of iteration accelerates.

Does this make sense from a planning metrics viewpoint? In this contrived example, we've tried to follow the guidelines prescribed by XP—that releases should be around two months in duration, with the first release being between two and six months—and we've kept the team size below 10. If this project is like Example 3 illustrated previously, it will deliver about 40,000 lines of Java code or 800 function points (if we accept the conversion factor found in the COCOMO II model) in total. If this is evenly divided between releases, then each release delivers about 115 function points.

The doubtful part will be the first release: to go from a standing start (say, one staff member assigned, no requirements captured, no scoping done yet), to a delivery to a customer of something of production quality in three months, will be a challenge. Even if we allow for high productivity, say 12 function points/staff-month (based on industry data from The David Consulting Group, see http://davidconsultinggroup.com/indata.htm), it is not possible to ramp-up the staffing at an arbitrarily high rate—the problem space we are dealing with here (115 function points) is quite small, and cannot be divided into arbitrarily small pieces so that a large team could attack it.

However, if we assume it can be done, then an average team size of about four is needed and the project will exit release one with a team of seven staff. If one more staff member is added to the team for the duration of the project, then for the remainder of the project we are in XP's normal mode (maintenance) with a team of eight—which is in the XP comfort zone for size. As the delivered size increases, the productivity will almost certainly drop a little, and the decreased release interval (two months) will offset the increased staff numbers, so that each release delivers about the same additional functional weight as the first. Overall project effort is then 108 staff-months—a little lower than in Example 3 shown previously.

Default RUP lifecycle

In contrast, the *default* lifecycle for a RUP project of similar size looks like this:

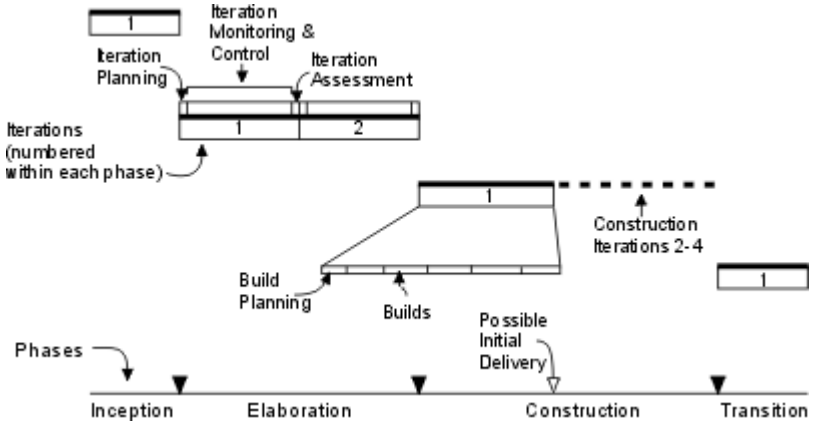

Figure 2

Again, this plan is based on Example 3. The RUP iterations are numbered within each phase. Using the default RUP lifecycle, most probably the earliest opportunity to *deliver* anything approaching production quality to the customer would be at the end of the first construction iteration, which occurs seven months into the project. However, at this time, RUP could deliver about one quarter, or 200 function points, of the total capability (as against XP's 115 function points after three months). Normally the first delivery to the customer would occur at the end of construction, which in this case would be 13 months into the project, at which time essentially all capability would be present. This does not mean this is the first time the customer has seen the product; with RUP, the customer is invited to the iteration assessments and can see the evolving product under test.

Why does this difference exist between RUP and XP? It occurs because, in this default case, the assumption is that there is enough risk in the architecture of the solution (because, for example, it is unprecedented, or the technology is new, or the non-functional requirements are particularly onerous) to warrant two iterations in elaboration to stabilize it, in advance of trying to build the customer-required functionality. This means that by the end of elaboration, only the architecturally-significant requirements will have been explored (and possibly implemented).

XP does not do this. It proceeds to deliver a series of vertically complete solutions with each release, the assumption being that the architecture will not break between releases or that refactoring can repair any breakage that does occur. We believe that this limits the sensible application of XP to classes of system that can be built on existing architectures of known capability.

Actual RUP lifecycle

For such systems, the actual RUP lifecycle would look a little different: the elaboration phase would be much shorter and the inception phase would probably be shorter too. If we trade one elaboration iteration for a construction iteration, we can bring forward the end of the first construction iteration by a couple of months, to five months instead of seven.

This will likely deliver a little less than the 200 function points in the default example (because the number of staff working in the earlier construction iteration would be smaller), but this is approaching the agility of the XP schedule—with enough relaxation to significantly reduce the risk. In the extreme, we can imagine a RUP-based approach established around the

initial delivery of a small system (say, 115 function points as in the XP example), followed by a number of complete evolutionary cycles of RUP (complete inception, elaboration, construction, and transition sequences) to deliver the rest of the system in a maintenance mode analogous to XP.

Again, at around six months, the initial delivery is likely to be at the outer limit of the time range recommended for XP, but then RUP acknowledges and requires you to plan (and so allocate time and effort) for things that XP seems to gloss over, such as the deployment of releases.

We can infer other restrictions, from these observations, on the nature of projects for which XP is suitable. XP explicitly requires a very intimate relationship between customer and development team, requiring a full-time customer on-site, and requiring that customers write the stories and define the releases. In XP, the customer is actually a role in the team: the person or group playing the role may or may not belong to a separate procuring company, but they have the authority to speak for whoever is procuring the software. Essentially, they have the requirements.

This kind of relationship is more often found in in-house developments than in formally contracted developments for outside clients. A two-month release cycle would also be more acceptable in such environments. Deployment of a new release (with new features, not just defect fixes) every two months, across company boundaries, is likely to have an unacceptable logistical overhead.

With these kinds of relaxation (small team, already established architecture, in-house style development with low ceremony deployment) an appropriately tailored RUP cycle (or cycles) would have an effect similar to an XP development. The initial release duration may be slightly longer than the XP optimum, but it would have lower risk.

In the opposite direction (large team, unprecedented architecture, formally contracted development with constraints on delivery and deployment) it's difficult to see how XP can scale—and to be fair, the sources for this paper do not claim it can. RUP, on the other hand, is designed to cope with the formality and ceremony that accompanies such projects. Perhaps an XP-like approach (that is, using XP techniques such as pair programming and refactoring) could be embedded *within* a larger RUP project, but only once the architecture is stable.

Artifacts

RUP describes over 100 artifacts, which has led to the criticism that it imposes an intolerable bureaucratic overhead for small projects. This view is incorrect on four counts:

- Projects don't *have* to produce all artifacts: selecting and tailoring artifacts is a necessary part of the process. RUP provides guidelines on what can be omitted and what can be tailored.
- An artifact (in RUP, an artifact *description*) is a specification entity—it specifies the information to be produced by an activity and, to do this systematically, it may describe the abstract *form* of an artifact. The RUP characterization of artifacts as models, model elements, or documents (about half of the RUP artifacts are classified as documents) is not intended to imply a particular *realization.*

  A UML model could be captured and presented using a purpose-built tool (such as IBM Rational Rose), or, at the other extreme, as diagrams made by using a simple graphics tool, or even as a sketch on a whiteboard—this would still count as an artifact in RUP terms (although there is a risk of loss with a whiteboard!).

  The term "document" has historical connotations that still bias people toward thinking of a document as a discrete, paper-based work product with a particular (and required) set of paragraph headings—all of which have to be filled out with text (and perhaps diagrams) for the document to be complete. But this is just one realization of a RUP document.

  RUP documents are information sets composed of text and diagrams. To be useful, RUP specifies what this information should be, and a convenient and systematic way to do this is to point to a template, which enumerates and describes the items and issues to be considered. It's certainly possible to use these templates in a direct and formal way to realize the artifacts in document form (in the general sense); that is, electronic or paper. Recognizing that many projects will want to do this, we chose to present a set of templates that could be used, more or less directly, in this way. But it isn't essential to do this—not everything presented in a template is necessarily relevant to a particular project—and RUP gives a project the freedom to choose any appropriate realization (and delivery mechanism) for an artifact. The important thing is the information it contains.
- We felt it was essential for all potential work products of the process to be clearly identified for consideration by the process engineer and project manager, so decisions to omit or tailor them could be made consciously and with the full understanding of the consequences of omission. We believe that making the artifact list complete and explicit in this way lends objectivity to configuring the process. It also supports the less experienced project manager by bringing to his or her attention those things that a more experienced manager would "take as read", were they not mentioned, but which the novice might simply overlook. Having a well-defined document set also helps the customer and project manager to agree early on what is to be delivered and in what form to avoid the unpleasant arguments that frequently characterize project closedown.
- RUP describes several artifacts that are composites, and also goes on to describe their components as artifacts. This permits a fine-grained description of activity inputs and outputs where this is useful. In its description, RUP could have stopped at the composite, for example, the Design Model, and referred to Classes simply as parts of the Design Model and not as artifacts in their own right. Identifying these as artifacts permits a precise description of their use, at the same time complying with the process metamodel at the expense of increasing the overall artifact count.

XP seems to be immune from the criticism of being artifact-heavy, but this is an oversimplification for two reasons:

- XP is *already tailored* to suit a certain kind of development, dealing with a subset of the RUP disciplines at a certain scale and so would be expected to call for fewer artifacts.
- XP emphasizes the importance of user stories and code, but other things that are work products are mentioned in passing when describing the process, so the artifact count is not as small as it appears at first sight.

Considering the XP approach, it's perhaps not surprising that in the three books we used as source material for this comparison (because they are the flagships of the process), the terms "artifact" and "work product" do not appear in the index. However, it's not difficult to read through the text and pick out references that are artifacts. Here are some examples:

- Stories
- Constraints
- Tasks
- Technical tasks
- Acceptance tests
- Software—code
- Releases
- Metaphors
- The design—CRC, UML sketch
- Design documents—produced at the end of the project
- Coding standards
- Unit tests
- Workspace (development and other facilities)
- Release plan
- Iteration plan
- Reports and notes on any meetings
- Overall plan—budget
- Reports on progress
- Story estimates
- Task estimates
- Defects (and associated data)
- Additional documentation from conversations
- Supporting documentation
- Test data
- Testing framework tools
- Code management tools
- Test results
- Spikes (solutions)
- Task work time records
- Metrics data
    - Resources
    - Scope
    - Quality
    - Time
- Other metrics
- Tracking results

There we have around 30 artifacts, some of which are composites too. The list is intended to be indicative, not exhaustive. The XP books don't spend a lot of time describing most of these and, at the level at which the books are written, we would not expect them to. However, once a project has to realize them, more detail of their content and form will be needed. A project can certainly do that on the fly, but that takes time away from the real work; in contrast, RUP provides up-front guidance, thereby saving the project time.

Why doesn't XP need all of the RUP artifacts?

One reason is that XP doesn't have the scope of RUP. This is quite intentional: XP is about programming to meet a business need. How that business need occurred—and how it's modeled, captured, and reasoned about—is not XP's main concern. The XP team member who is the customer presents the distillation of requirements as stories to the XP development team; they are also the arbiters of business value. The magic of how the stories came to be expressed (or expressible) in that form is not the concern of XP. So, for example, what RUP describes in its Business Modeling discipline is outside the scope of XP (Business Modeling in RUP has ~14 artifacts). XP describes a process that has regular releases to the customer. The logistics of deployment of these releases are not the concern of development, so the RUP Deployment discipline is largely outside the scope of XP (Deployment in RUP has ~9 artifacts). Therefore, for a small project amenable to XP, you would expect to omit ~23 artifacts when tailoring RUP.

Another reason is that XP asserts that requirements and design capture can be done rather simply: requirements are captured as user stories (which may sometimes have to be split) that are then decomposed into tasks—which, in essence, are the design—keeping in mind a metaphor for the system. Is this possible for all systems? Definitely not. Is it possible for some systems? Certainly, and to be fair to XP it doesn't claim to address all systems. Also, perhaps the XP authors had their tongues firmly planted in cheek when making these claims.

The desired behavior of larger, more complex systems can be very difficult to articulate without some systematic approach such as use cases. Neither will it be possible to rely on conversation between customer and developer to consistently elaborate complex user stories, human memory being as fallible as it is. Also, developing the structures that *realize* complex behavior in large systems is aided by the ability to form and reason with various abstract views of the architecture of that system. The artifacts RUP describes in the Requirements (~14 artifacts) and the Analysis and Design disciplines (~17 artifacts), enable it to cope with the variety, size, and complexity of these systems.

In RUP's small project roadmap, the count of artifacts (for Requirements and Analysis & Design) is reduced to seven, with some of the reduction achieved by simply referring to the composite artifact. This is no sleight-of-hand; it deals with artifacts in just the same way as XP. For example, because of the way it's likely to be realized in a small project, RUP says the following about the Design Model:

> "The Design Model is expected to evolve over a series of brainstorming sessions in which developers will use CRC cards and hand-drawn diagrams to explore and capture the design. The Design Model will only be maintained as long as the developers find it useful. It will not be kept consistent with the implementation, but will be filed for reference."

Finally, RUP allows for a greater formality (and "ceremony") in project management, where this is necessary, and in some contract arrangements it will be**.** Again, many of RUP's project management artifacts (there are 15 in the Project Management discipline) are part of composite artifacts, and need only be realized as separate documents when the formality of the project demands it. For example, in RUP the Software Development Plan "contains" artifacts such as the Risk Management Plan and Product Acceptance Plan. In smaller, less formal projects it may be possible to deal with the issues addressed by these plans simply with a paragraph or two in the Software Development Plan. In the small project roadmap in RUP, the number of project management artifacts is reduced to six.

Comparing artifacts for a small project

So, when you tailor RUP for a small project and tailor the artifact requirements, what happens overall? When you look at the small project roadmap in RUP and count the artifacts, the number is 30 (26 if you leave out some of deployment)—not so artifact-heavy after all! RUP simply delineates clearly what XP leaves fuzzy, allows you to decide what is necessary, and provides guidance on how to make the selection. Now, the granularity varies on both sides, but the point is that the artifact count in RUP for small projects of the type that XP would comfortably address is of the same order as XP's.

Rough Mapping of XP artifacts to RUP artifacts

XP	RUP Small Project Roadmap
Stories Additional documentation from conversations	Vision Glossary Use-Case Model
Constraints	Supplementary Specifications
Acceptance tests Unit tests Test data Test results	Test Model
Software—code	Implementation Model
Releases	Product Release Notes
Metaphor	Software Architecture Document
Design—CRC, UML sketch Tasks Technical Tasks Design documents—produced at the end of the project Supporting documentation	Design Model
Coding standards	Design Guidelines Programming Guidelines
Workspace (development and other facilities) Testing framework tools	Tools
Release plan Story estimates Task estimates Iteration plan	Software Development Plan Iteration Plan
Overall plan—budget	Business Case Risk List Product Acceptance Plan
Reports on progress Task work time records Metrics data: Resources, Scope, Quality, Time Other metrics Tracking results Reports and notes on meetings	Status Assessment
Defects and associated data	Change Requests
Code management tools	Configuration Management Plan Project Repository Workspace
Spike	Prototypes
	Development Case Project-Specific Templates

Guidelines

Associated with artifacts, RUP provides guidelines, which are essentially more information about that artifact; its meaning, representation, relationships to other artifacts, content, and use. These guidelines would cover several hundred physical pages, if printed, which may seem daunting, but then you only read what you have to, for those artifacts you need.

The XP books also contain much guidance on both practices and artifacts, without attempting (or needing) to model the relationships rigorously. The total body of literature on XP is not tiny though. The three books available at the time of writing total almost 600 pages, and two others will be available soon adding another 700 or so pages. Then there's the book on refactoring [Fowler99] at 400+ pages.

In addition to this, XP is covered on several Web sites. The coverage does tend to overlap though—examining XP from different viewpoints and adding to the experience base. Actually, this illuminates another difference between RUP and XP: RUP is a product; XP is not. There is no single source for XP, although the books are a good start. The quickest way to "acquire" XP would be through the training that's commercially available from the thought leaders behind it.[11]

[11] The RUP is sometimes portrayed as "proprietary", but anyone can buy RUP and use the ideas in it, add to it, delete parts that are not relevant to their organization or project, and so on, provided they respect the copyright and the license agreement. XP's manifestation in book form is also intellectual property owned by the authors and publishers; the only difference is the extent to which each can be fully comprehended from their sources. RUP, as a product, attempts to be complete; the current XP books need some supplementation (either through immersive training or consulting).

Activities

RUP formally defines the term "activity" as work performed by a role, using and transforming input artifacts, and producing new and changed output artifacts. RUP then goes on to enumerate these activities and categorize them according to "discipline" or major "area of concern" within project (as RUP defines it). These disciplines are:

- Business Modeling
- Requirements
- Analysis & Design
- Implementation
- Test
- Deployment
- Configuration & Change Management
- Project Management
- Environment

Activities are time-related through the artifacts that they produce and consume: an activity can logically begin when its inputs are available (and in an appropriately mature state). This means that producer-consumer activity pairs can overlap in time, if the artifact state permits; they need not be rigidly sequenced.

Activities in RUP are intended to make the intellectual process of producing an artifact less opaque—to give some strong guidance on how something should be produced. Activities may also be used to help the project manager with his or her planning. Woven through the RUP as it's described in terms of lifecycle, artifacts, and activities, are the "best practices": software engineering principles proven to yield quality software built to predictable schedule and budget.

The RUP, through its activities and their associated artifacts, supports and realizes these best practices—they are themes running through the RUP. Note that XP uses the notion of "practices" as well, but as we will see, there is not an exact alignment with RUP's concept of best practice.

XP (see [Beck00], Chapter 9) presents an engagingly simple view of software development as having four basic activities that are to be enabled and structured according to some supporting practices:

- Coding
- Testing
- Listening
- Designing

Actually, XP's activities are closer in scope to RUP's disciplines than to RUP's activities, and much of what happens on an XP project (in addition to coding, testing, listening, and designing) will come from the elaboration and application of its practices.

Is there an XP equivalent of RUP's activities?

Yes there is, but XP's "activities" are not formally identified or described: for example, looking at Chapter 4, *User Stories* in [Jeffries01], you'll find it headed "*Define requirements with stories, written on cards*" and then throughout the chapter is a mixture of process description and guidance about what use stories are and how (and by whom) they should be produced. And so it goes on, as the books describe XP under major headings (which are a mixture in [Jeffries01]—some are artifact focused, some are activity focused); "things done" and "things produced" are described with varying degrees of prescription and detail.

RUP's apparent prescription comes from its completeness and greater formality in its systematic treatment of activities, and their inputs and outputs. XP does *not* lack prescription, but perhaps in the attempt to remain "lightweight", the formality and detail are simply omitted. The detail that is present in RUP will have to be added when XP is implemented on a project. Lack of specificity is not a strength or a weakness, but you should not confuse the lack of detailed information in XP with simplicity. At some point, the people on the project will need to know what to do and at that time will need the detail.

Disciplines and Workflows

RUP recently substituted the term "discipline" for "core workflow". A discipline in RUP is the collection of activities (and associated concepts) producing a particular set of artifacts, which represents some important aspect or concern in software development. This usage aligns quite well with the dictionary definition of discipline as a branch of knowledge or teaching.

As noted previously, RUP's disciplines are Business Modeling, Requirements, Analysis & Design, Implementation, Test, Deployment, Configuration & Change Management, Project Management, and Environment. This does not cover every aspect of what an organization or business might do when employing people to develop, deploy, operate, support, sell, market, or otherwise deal with systems that are largely software. RUP currently does not cover Systems Engineering, for example. Nor does it cover all of the requirements of some international software process standards such as ISO 15504 (which draws in aspects related to software acquisition and human resource management, for example). This is by choice: these other aspects, while important, are outside the engineering focus of the RUP.

XP explicitly restricts itself even further: it includes four basic activities—coding, testing, listening, and designing (noted previously as more closely aligned with RUP disciplines)—performed using a set of practices, which requires performing other activities, which map to some of the other disciplines in RUP. XP's practices, quoted verbatim from [Beck00], are:

- *"The Planning Game*—Quickly determine the scope of the next release by combining business priorities and technical estimates. As reality overtakes the plan, update the plan.
- *Small Releases*—Put a simple system into production quickly, then release new versions on a very short cycle.
- *Metaphor*—Guide all development with a simple shared story of how the whole system works.
- *Simple Design*—The system should be designed as simply as possible at any given moment. Extra complexity is removed as soon as it is discovered.
- *Testing*—Programmers continually write unit tests, which must run flawlessly for development to continue. Customers write tests demonstrating that features are finished.
- *Refactoring*—Programmers restructure the system without changing its behavior to remove duplication, improve communication, simplify, or add flexibility.
- *Pair programming*—All production code is written with two programmers at one machine.
- *Collective ownership*—Anyone can change any code anywhere in the system at any time.
- *Continuous integration*—Integrate and build the system many times a day, every time a task is completed.
- *40-hour week*—Work no more than 40 hours a week as a rule. Never work overtime a second week in a row.
- *On-site customer*—Include a real, live user on the team, available full-time to answer questions.
- *Coding standards*—Programmers write all code in accordance with rules emphasizing communication through the code."

Something to note on the topic of *Simple Design* is that "extra complexity" is a somewhat subjective term and is difficult to define, being largely in the eye of the experienced beholder.

Activities performed as a result of the practice *The Planning Game*, for example, will mainly map to RUP's Project Management discipline. However, some topics are outside the scope of XP; for example, Business Modeling. Requirements *elicitation* is largely outside the scope of XP—the customer (on-site with the team) defines and provides the requirements (in the form of stories). Deployment of the released software is outside the scope of XP. Because of the scale and types of development it addresses, XP can deal *very* lightly with issues in Environment and Configuration & Change Management disciplines that the RUP covers in detail.

Use of XP practices in RUP

In the disciplines in which XP and RUP overlap, some of the practices described in XP could be employed in RUP (and some are already); for example:

- *Pair programming:* XP requires that production code is produced by programmers working in pairs at a single workstation, and claims that this has beneficial effects on code quality and, once the skill is acquired, is more enjoyable. RUP does not describe the mechanics of code production at this fine-grained level and it would certainly be possible to use pair programming in a RUP based process. Some information on this practice and on *Test-first design and refactoring* (see below) is now provided with RUP in the form of white papers. Obviously, it's not a requirement to use this practice in RUP, nor do we believe it's necessary to mandate it.

  Evidence for the benefits *at the industrial scale* is scanty and anecdotal; studies by ([Nosek98] and [Williams00]) have compared pairs against individuals (working in isolation), but good teams don't work that way. In a team environment, with a culture of open communication, where individuals feel free to ask questions of their peers (and team leads or management), and work to a defined process, we'd hazard a guess that the benefits of pair programming (in terms of effect on total life-cycle costs) would be much harder to discern. People will come together to discuss and solve problems quite naturally in a team that's working well, without being obliged to do so.

  [Beck00] takes a somewhat gloomy view of people and process when it says: "Another powerful feature of pair programming is that some of the practices wouldn't work without it. Under stress, people revert. They will skip writing tests. They will put off refactoring …". These are supposed to be natural practices that lead to a better result—why would anyone not do them?

  The instantiation of a good process is non-intrusive; the suggestion that it has to be enforced at the "micro" level is unpalatable. However, in some circumstances, working in pairs is obviously advantageous, as each can help the other along; for example:

  - In the early days of team formation, as people are getting to know one another
  - In teams inexperienced in some new technology
  - In teams with a mix of experienced and novice staff

- *Test-first design and refactoring:* These are good techniques that can be applied in the Implementation discipline in RUP. Test-first design, in particular, is an excellent way to clarify requirements at a detailed level.
- *On-site customer:* Many of RUP's activities would benefit greatly in terms of increased efficiency through having a customer on-site, as a team member. With the customer on board in this way, the need for many intermediate deliverables (particularly documents) can be reduced.

  XP is reluctant to say that anything other than code should be captured and stresses conversation as the preferred communication medium. This relies on continuity and familiarity to succeed. When a system has to be transitioned, then, even for small systems, other things will need to be produced.

  XP does finally allow this, but as something of an afterthought; for example, design documents at the end of a project. In fact, XP does not prohibit the production of documents or other artifacts (it just keeps quiet about it), saying rather that you should produce only those that you really need; those that are used. RUP agrees, but goes on to list and describe what you *might* need when continuity and familiarity are not ideal.

- *Coding standards:* RUP has an artifact (Programming Guidelines) that would almost always be regarded as mandatory (most project risk profiles, being a major driver of tailoring, would make it so).
- *Continuous integration:* RUP supports this through builds at the subsystem and system level (within an iteration); unit-tested components are integrated and tested in the emerging system context.

XP practices that do not scale

However, some practices do not scale (and XP does not claim that they do), and we would make their use subject to this proviso in RUP; for example:

- *Collective ownership:* It's useful to have the members of a small team, who are responsible for a small system or a subsystem of a larger system, familiar with all of its code. Whether you'd like to have all team members equally empowered to make changes anywhere depends on the nature and complexity of the system or subsystem. It will often be faster (and safer) to have the individual (or pair) currently working with a code segment make a fix.

Familiarity with even the best-written code diminishes rapidly as time passes, particularly if it's algorithmically complex.

- *Refactoring:* In a large system, frequent refactoring is no substitute for a lack of architecture. [Beck00] says, "XP's design strategy resembles a hill-climbing algorithm. You get a simple design, then you make it a little more complex, then a little simpler, then a little more complex. The problem with hill-climbing algorithms is reaching local optima, where no small change can improve the situation, but a large change could."

  In RUP, architecture provides the view and access to the "big hill" to make large, complex system tractable.

- *Metaphor:* For larger, complex systems, architecture as a metaphor is simply not enough. RUP provides a much richer descriptive framework for architecture that is not just, as [Beck00] dismissively describes it, "big boxes and connections".

- *Rapid releases:* The rate at which a customer can accept and deploy new releases will depend on many factors; one of which is usually the size of the system, which is usually correlated with business impact. A two-month cycle may be far too brief for some classes of system—the logistics of deployment may prohibit it.

And some practices, that are at first glance obviously sound and potentially usable in RUP, need a little elaboration and caution when applied generally:

- *Simple design:* XP is very much functionally driven: user stories are selected, decomposed into tasks, and then implemented. According to [Beck00], the "right design for the software at any given time is the one that:

  1. Runs all the tests
  2. Has no duplicated logic…
  3. States every intention important to the programmers
  4. Has the fewest possible classes and methods."

  XP does not believe in adding anything that is not needed now. There is a problem here somewhat akin to the local optimization problem in dealing with a class of requirements called non-functional in RUP. These requirements carry business value to the customer, but are more difficult to express as stories—some of what XP calls constraints fall into this category.

  RUP does not advocate designing for more than is required in any kind of speculative way either, but it does advocate designing with an architectural model in mind; that architectural model being one of the keys to meeting non-functional requirements.

  So RUP agrees with XP: the "simple design" should run all the tests, with the rider that this includes tests that demonstrate that the software will meet the non-functional requirements. Again, this only looms as a major issue when system size and complexity increase, when the architecture is unprecedented, or when the non-functional requirements are onerous. For example, the need for marshalling of data (to operate in a heterogeneous distributed environment) seems to make code overly complex, but it's still a global necessity.

- *40-hour week:* As in XP, RUP suggests that working overtime should not be a chronic condition. XP does not suggest a hard 40-hour limit, recognizing different tolerances for work time. Software engineers are notorious for working long hours without extra reward just for the satisfaction of seeing something completed and a manager need not necessarily put an arbitrary stop to that.

  What a manager should not do is exploit it or impose it—and a manager should always be collecting metrics on hours *actually* worked, even if uncompensated. If the log of hours worked by anyone seems high for an extended period, then certainly it should be investigated. But these are issues to be resolved in the particular circumstances in which they arise, between the manager and the individual, recognizing any concerns the rest of the team might have. Forty hours is only a guide (but a strong one).

Roles

In RUP, activities are performed by roles.[12] Roles also have responsibility for particular artifacts—the responsible role will usually create the artifact and ensure that any changes made by other roles (if such changes are allowed), do not break the artifact. A role in RUP may be performed by an individual or by a group of people. Equally, an individual or a group may perform several roles. A role does not have to be mapped to a single position or "slot" in an organization; the mapping of roles to organizational units may also be many-to-many.

RUP roles

RUP defines a total of 30 roles: there is not an exact mapping to disciplines because a role—for example, a software architect—is not necessarily confined to one discipline, but roughly (placing the role where it primarily belongs):

- Business Modeling has three roles
- Requirements has five roles
- Analysis & Design has six roles
- Implementation has three roles
- Test has two roles
- Deployment has four roles
- Configuration & Change Management has two roles
- Project Management has two roles
- Environment has three roles

Why are there so many roles in RUP?

Roles in RUP are used to partition activities, and to finely discriminate the skills and competencies needed to perform the role, which helps guide the selection of staff to perform the roles. The level of division also facilitates the identification of new organizational positions when the importance of a role changes. For example, on a small, informal project, the role of Project Manager and Configuration Manager (RUP roles) may well be performed by the same individual; on a large, formal, mil-aerospace project, the work of the Configuration Manager may well be quite specialized and onerous enough to warrant a small team. By describing roles in this way, RUP facilitates this mapping.

XP roles

[Beck00] identifies seven roles applicable to XP, and then goes on to describe the responsibilities of the roles, the skills, and the traits required of the people who will perform them. These roles are:

- Programmer
- Customer
- Tester
- Tracker
- Coach
- Consultant
- Big Boss

References are made to these roles in some of the other XP books, in places elaborating on the activities the role performs.

The difference in the number of XP and RUP roles is easily explained:

[12] To be more precise (and accurate), activities are performed by individuals or teams *playing* the roles.

- XP is not covering all of the RUP disciplines.
- XP roles are actually closer to positions than RUP roles; for example, XP's programmer actually performs multiple RUP roles—those of Implementer, Integrator, and Code Reviewer—and these require slightly different competencies.

When RUP roles are mapped to a small project (as in the RUP's small project roadmap), the number of XP-like roles, that is positions, that are mapped reduces considerably from 30. In the small project roadmap, the number of positions is five, as shown in the following table.

RUP Roles Mapped to a Small Project for ABC Company

ABC Company Job Title	RUP Role
Project Manager	Project Manager Process Engineer Deployment Manager Requirements Reviewer Architecture Reviewer
ABC Company Executive	Project Reviewer Stakeholder Requirements Reviewer
Chief Programmer	System Analyst Requirements Specifier User Interface Designer Software Architect Design Reviewer Process Engineer Tool Specialist Configuration Manager Change Control Manager *and, to a lesser extent, the same roles as the* Programmer
Programmer	Designer Implementer Code Reviewer Integrator Test Designer Tester
Administrative Assistant	*Responsible for:* • *maintaining the "Small Project" Web site* • *assisting the* Project Manager *role in planning or scheduling activities* • *assisting the* Change Control Manager *role in controlling changes to artifacts* • *may also provide assistance to other roles as necessary*

Conclusions

XP is actually not the same kind of thing as RUP. RUP is a process framework from which particular processes can be configured and then instantiated. RUP *has* to be configured and this is actually a required step defined in RUP itself. Strictly speaking, one should compare a tailored version of RUP with XP, with RUP tailored to the project discriminants that XP explicitly establishes (and those which can be inferred). Such a tailored RUP process could accommodate some XP practices (such as pair programming and test-first design and refactoring), but would not be identical to XP because of its acknowledgment of the importance of architecture, abstraction (in modeling), and risk, and its different structure in time (phases, iterations).

RUP will permit the construction of processes to accommodate projects that are outside the scope of XP in scale or kind. RUP is heavyweight only in that it's a complete *description* of a family of processes that can be as light or heavy—in artifacts, deliverables, formality, prescription, ceremony, or any other measure of "weight"—in implementation, as desired. XP is certainly lightweight in that it's intentionally directed at the implementation of a (lightweight) process, but XP's descriptions (at least in the books) are not elaborated either. So in an XP implementation, there will be things that will need to be discovered, invented, or defined on the fly. Therefore, compared with RUP, XP is also lightweight in descriptive material.

This is likely to change, in fact, it may already be changing with the publication of another two books, one of which, [Succi01], runs to 512 pages. However, as things stand, the profile of the adoption effort would be different between the two approaches. RUP shifts much of the effort up-front, both in training requirements and process tailoring. An organization will also, more than likely, tailor RUP for organization-wide application on particular types and sizes of projects, and will use the results in several projects. With XP, there will be some up-front training required, but the rest of the adoption effort will be spread over a project, as it expands on and captures all of those ancillary things that turned out to be needed to make XP work. XP does not obviously motivate the capture of "corporate memory", leaving an adopting organization (if it does not save its process experience) vulnerable to staff turnover.

Labeling RUP as heavyweight and XP as lightweight without further qualification does both a disservice by obscuring what each is and what each was intended to do. And, when done in a pejorative way, it's simply meaningless posturing. It is the implementations of these as processes that will be either "heavyweight" or "lightweight", and they should be as heavy or light as circumstances require them to be.

XP is not a free form, anything goes discipline—it focuses narrowly on a particular aspect of software development and a way of delivering value, and is quite prescriptive about the way this is to be achieved.

RUP's coverage is much broader and just as deep, which explains its apparent "size". However, at the micro level of process, RUP occasionally allows and offers equally valid alternatives, where XP does not; for example, the practice of pair programming. This is not intended as a criticism of XP; simply an illustration of how XP, as its name implies, has narrowed its focus.

References

[Beck00] *Extreme Programming Explained,* Kent Beck, Addison-Wesley, 2000

[Beck01] *Planning Extreme Programming,* Kent Beck, Martin Fowler, Addison-Wesley, 2001

[Boehm00] *Software Cost Estimation with COCOMO II,* Barry W. Boehm et al, Prentice Hall PTR, 2000

[Fowler99] *Refactoring: Improving the Design of Existing Code,* Martin Fowler et al, Addison-Wesley, 1999

[Jeffries01] *Extreme Programming Installed,* Ron Jeffries, Ann Anderson, Chet Hendrickson, Addison-Wesley, 2001

[Kruchten00] *The Rational Unified Process, An Introduction, Second Edition,* Philippe Kruchten, Addison-Wesley, 2000

[Martin01] *Extreme Programming in Practice,* Robert C. Martin, James W. Newkirk, Addison-Wesley, 2001 (not yet published)

[Nosek98] *The Case for Collaborative Programming,* John T. Nosek, *Comm. ACM,* Vol. 41, No. 3, 1998, pp. 105-108

[Succi01] *Extreme Programming Examined,* Giancarlo Succi, Michele Marchesi, Addison-Wesley, 2001 (not yet published)

[Williams00] *Strengthening the Case for Pair Programming,* Laurie Williams, Robert R. Kessler, Ward Cunningham, Ron Jeffries, IEEE Software, Vol. 17, No. 4, 2000, pp. 19-25

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

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

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

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

Printed in the United States of America
01-03 All Rights Reserved.
Made in the U.S.A.

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