RUP and XP, Part I: Finding Common Ground

by Gary Pollice
Evangelist, The Rational Unified Process
Rational Software

eXtreme Programming (XP) is hot! Attend any software development conference today and XP presentations are standing room only. Why? I have observed that XP speaks to the programmer in almost every technical manager and practicing software developer. We all remember simpler times when we sat at the keyboard and coded. Some of us were even happy testing and fixing the code because we wanted it to be perfect. Our code was our legacy to future generations of programmers. And we all had heroes, legends who cranked out more code with fewer errors than we thought humanly possible.

XP hints that we may yet return to the "good old days."

Why don't tears of nostalgia well up in our eyes when we think of the Rational Unified Process (RUP)? Because RUP® contains the dreaded "process" word, and most of us have had bad experiences with process. We recall that it was too heavy and too restrictive. Something that wasted our valuable time; something that kept us from coding. But the process itself was not the real culprit: our bad experiences stemmed from the way the process was implemented and used.

In this two-part series, we will look at how to make the implementation and use of RUP a good experience. We will see how it can be used effectively for a small project, and specifically how to incorporate XP practices into the broader scope of a RUP-based project. This month's installment will examine the areas where the two come together. Next month we will look at how RUP differs from XP, when and why you need to consider the differences, and what benefits accrue with the added strengths of RUP.

If you're not familiar with the two processes, then start with my overviews of XP and RUP.

How do you decide on the right process for a software development project? If you are considering using XP as your process, the first question you need to ask and answer is: Can I actually use XP for this project? This may not be as easy as it sounds. There are lively debates on what makes a project an eXtreme project (see

Overview: eXtreme Programming

eXtreme Programming (XP) is a software development discipline developed by Kent Beck in 1996. It is based on four values: communication, simplicity, feedback, and courage. It stresses continual **communication** between the customer and development team members by having an on-site customer throughout the development lifecycle. The on-site customer decides what will be built and in what order. The development team keeps things **simple** by continually refactoring code and producing a minimal set of non-code artifacts. Many short releases and continual unit testing are the **feedback** mechanisms. **Courage** means doing the right thing, even when it is not the most popular thing to do. It means being honest about what you can and cannot do.

Twelve XP practices support the four values. They are:

- *The planning game*. Determine the features in the next release through a combination of prioritized stories and technical estimates.
- *Small releases*. Release the software often to the customer in small incremental versions.
- *Metaphor*. The metaphor is a simple shared story or description of how the system works.
- *Simple design*. Keep the design simple by keeping the code simple. Continually look for complexity in the code and remove it at once.
- *Testing*. The customer writes tests to test the stories. Programmers write tests to test anything that can break in the code. These tests are written before the code is written.
- *Refactoring*. This is a simplifying technique to remove duplication and complexity from code.
- *Pair programming*. Teams of two programmers at a single computer develop all the code. One writes the

code, or drives, while the other reviews the code for correctness and understandability.

- *Collective ownership*. Everyone owns all of the code. This means that everyone has the ability to change any code at any time.
- *Continuous integration*. Build and integrate the system several times a day whenever any implementation task is completed.
- *Forty-hour week*. Programmers cannot work at peak efficiency if they are tired. Overtime is never allowed during two consecutive weeks.
- *On-site customer*. A real customer works in the development environment full-time to help define the system, write tests, and answer questions.
- *Coding standards*. The programmers adopt a consistent coding standard.

For more information on XP, see:

Kent Beck, *Extreme Programming Explained*. Addison-Wesley, 2000.

Ron Jeffries, et al., *Extreme Programming Installed*. Addison-Wesley, 2001.

Kent Beck and Martin Fowler, *Planning Extreme Programming*. Addison-Wesley, 2001.

You can also find information on XP at:

http://c2.com/cgi/wiki?ExtremeProgramming
http://www.extremeprogramming.org/
http://www.xprogramming.com/

http://c2.com/cgi/wiki?AreYouDoingXp for one view). If you don't refactor your code continuously and write tests before you code, for example, then are you doing XP? In truth, many, if not most, so-called XP projects do not follow XP in an orthodox way. Most use some XP techniques; few use them all.

If you're thinking of using XP straight up, here are some more specific questions you should ask yourself:

- Is the project team small (ten people or less)?
- Is the team co-located, and willing and able to do pair programming?
- Do we have a commitment for an on-site customer?

If you answered "no" to any of these questions, then you may not be a candidate for a full-fledged XP project. You may, however, be able to use the RUP and incorporate selected XP techniques into it.

Unlike XP, which focuses narrowly on small, co-located teams with on-site customers, RUP is broader and more flexible. It addresses many styles of software development projects and urges users to adapt its elements to suit their specific projects. It is not hard to imagine an adaptation of RUP that matches XP very closely. In fact, this is exactly the claim that Robert Martin makes for his dX Process.[1]

XP Practices and the RUP

The focus of XP is code: writing the code, keeping it simple, and getting it correct. This is a good thing, as far as it goes. If you build software, ultimately it comes down to delivering executable software to your customers.

Let's look at nine specific XP practices to see how they complement or overlap with the RUP. For each practice, we will briefly discuss benefits as well as limitations and hidden assumptions. See the XP overview for a brief description of each specific practice.

Everything Starts with Planning

When it comes to planning, RUP and XP agree: plans change, and you cannot, practically speaking, plan a complete project in detail. The best approach is to anticipate changes and ensure that you control the associated risks. According to XP, you should prioritize the "stories" you want your system to fulfill, and get technical estimates for the effort required to implement them. Is prioritizing stories any different from prioritizing use cases? Not really, if you equate stories with use cases. Some of the example stories from XP literature are not really use cases, so it may not make sense to equate the two. A story describes a unit of work, and XP assumes that the story's context is obvious. A use case provides a complete set of operations that provide value to a system user. I believe stories and use cases complement each other, and that a use case can be realized through multiple stories. A use case speaks to all stakeholders, whereas stories speak in more detail to developers. You can produce use-case realizations (according to the RUP) by filling in a complete, more detailed context for the stories.

Alistair Cockburn says that stories are promises for conversations between the on-site customer and the programmer. These conversations are of great value, and the RUP specifically asks you to consider capturing their results in use cases and other requirements artifacts. XP *implies* that you should capture the results but provides little guidance on how to do it. In XP, the final resting place for requirements or design decisions is the code. Unfortunately, code is not an effective communication medium for all stakeholders.

Getting technical estimates from the developer who will implement a feature is good practice. RUP does not go into detail about how to obtain these estimates, but if you have confidence in the developer, then adopt the practice as part of your planning process. In fact, go beyond this practice: When you get into the details of project deliverables, do estimates for documentation, training, support, and manufacturing.

Simple Design: No Arguments

Every technical discipline preaches simplicity. XP tells us to build the simplest system that meets current requirements, recasting this principle as *You Aren't Going to Need It*. What this means is that you should implement things when you actually need them, not when you realize that you might need them in the future. The RUP says almost the same thing using different words and at different levels: Manage your requirements, continually prioritize, and assess progress. Well-defined, prioritized requirements simplify the developer's decision making about what to do. The RUP also encourages the use of components and the Unified Modeling Language to help manage design complexity.

It is easy to misinterpret the XP advice and mistakenly assume that you do not have to pay attention to infrastructure and architecture. But simple design does not mean that you can ignore required infrastructure or architecture. There is a sharp difference between RUP and XP in this area, which we will discuss next month under the XP practice of "metaphor."

Testing: The Last Word?

Test first, then code! This is wisdom from XP, and it is good. The other testing pearl from XP is that the customer provides the acceptance test. Programmers write unit tests to ensure that the code does what they think it does. Customers write acceptance tests to ensure that the system does what it is supposed to do. RUP has a general framework for testing and provides guidance on how to write effective tests. In addition to unit and customer written tests, others may be required: for example, load tests for Web sites. Combine RUP and XP, and you get an excellent quality focus for your team.

Overview: The Rational Unified Process

The Rational Unified Process (RUP) provides a disciplined approach to software development. It is a *process product*, developed and maintained by Rational Software. It comes with several out-of-the-box roadmaps for different types of software projects. The RUP also provides information to help use other Rational tools for software development, but it does not require the Rational tools for effective application to an organization; integrations with other vendors' offerings are possible.

The RUP provides guidance for all

In XP, the development team uses the test results to decide whether the system is ready for the customer. If the system passes all acceptance tests, then the software is ready. RUP suggests other acceptance criteria in addition to testing. Depending upon the project, you might consider including customer training, on-site installation, documentation, and several other items in your product acceptance criteria. Simply because a system passes the tests does not ensure that a programmer (or programming pair) has not inserted a trap door or some other time bomb in your software. Sometimes, depending upon the type of system, you need more rigorous code inspections by independent auditors.

Refactoring: A Little Goes a Long Way

Refactoring is the act of rewriting code to improve it. It is also a technique for keeping the design simple. RUP does not address code refactoring, but that does not mean you should not consider it. Be aware, however, that refactoring may create a risk for your team. What's simple to one programmer may be complex to another. If you do too much refactoring, then the team may thrash around and lose valuable time developing the code.

Pair Programming: Are Two Heads Better Than One?

There is evidence that pair programming is an effective way to improve programmer productivity. Programmers remain focused, and because they get immediate feedback,

aspects of a software project. It does not require you to perform any specific activity or produce any specific artifact. It does provide information and guidelines for you to decide what is applicable to your organization. It also provides guidelines that help you tailor the process if none of the out-of-the-box roadmaps suits your project or organization.

The RUP emphasizes the adoption of certain *best practices* of modern software development, as a way to reduce the *risk* inherent in the development of new software. These best practices are:

- Develop iteratively
- Manage requirements
- Use component-based architectures
- Model visually
- Continuously verify quality
- Control change

The Rational Unified Process weaves these best practices into the definitions of

- *Roles* - sets of activities performed and artifacts owned
- *Disciplines* - focus areas of software engineering effort such as *Requirements, Analysis and Design, Implementation*, and *Test*
- *Activities* - definitions of the way artifacts are produced and evaluated
- *Artifacts* - the work products used, produced or modified in the performance of activities

The RUP is an iterative process that identifies four phases of any software development project. Over time, the quality improves. Pair programming is one way to avoid code reviews, but it places some constraints on a project team:

- The team must be in one location.
- Paired team members must have compatible personalities plus well-matched programming skills.

project goes through Inception, Elaboration, Construction, and Transition phases. Each phase contains one or more iterations in which you produce an executable, but possibly incomplete, system (except possibly in the Inception phase). During every iteration, you perform activities from several disciplines in varying levels of detail.

You'll find a good overview of the RUP by Philippe Kruchten in the January issue of *The Rational Edge*. You can also find further information and an evaluation of the RUP on the Rational Software Web site: http://www.rational.com.

Continuous Integration: Do One Build or More Per Day

Every programmer on an XP project must be able to change code and ensure that it works, not just for unit tests but also for acceptance tests. This requires frequent builds: one or more a day. This practice is an excellent one. In order for it to really work, however, you need powerful configuration management tools and an effective process for using them. The RUP provides general guidelines for continuous integration as well as specific information on using Rational ClearCase. In fact, at Rational, we perform daily integrations on the complete Rational Suite product family.

Forty Hour Work Week: No Sleeping Under the Desk

What a great idea! Is it practical for your organization? Studies indicate that most people experience rapidly diminishing returns when they invest more than forty hours in their work -- especially when it is habitual. An XP project forbids two consecutive weeks of overtime.

On-site Customer: A Must-Have?

Originally, XP said, "A real customer must sit with the team, available to answer questions, resolve disputes, and set small-scale priorities." This has been further refined to, "An XP project is steered by a dedicated individual who is empowered to determine requirements, set priorities, and answer questions as the programmers have them."

The RUP is more flexible. Although it has always maintained that the customer, in fact all stakeholders, must be adequately represented in steering a project, the RUP also acknowledges that it is not always possible or desirable to have a real customer co-located with the development team. Instead, RUP defines several roles that are responsible for determining the project goals, scope, and so on, and says that a customer (on-site or not) or some other appropriate person in the organization can perform the activities mapped to these roles. It's not important whether the person is an actual customer, or whether he or she is actually on site. What *is* important is that the person be available to clarify issues, and capable and responsible enough to produce the information necessary for the team to progress as quickly as possible -- including feedback in a form the team can understand.

Coding Standards: Have Them and Use Them

No one is going to argue against having coding standards. But what is *really* important? To use them! It doesn't matter what the standards *are* as long as everyone uses them. The RUP provides three coding standard examples to get you started: Ada, C++, and Java. As a complement to these coding standards, the RUP also encourages that you define architectural mechanisms, which standardize not only the language of the code, but also its structure and usage (error handling and transaction locking are examples of common mechanisms).

Summary

As you can see, there is significant agreement between RUP and XP on nine of the twelve XP practices. Typically, RUP provides complementary guidance for doing more to address specific risks.

As you evaluate both processes *vis à vis* your next project, it's important to keep in mind this primary rule:

Ask yourself, "If we don't perform a specific activity, produce an artifact, or adopt a practice, will anything bad happen?"
If the answer is "no," then don't do it!

Next month we will look at the three remaining XP practices and discuss additional pitfalls associated with XP practices. And finally, we'll examine what important areas XP does *not* address.

---

1 See http://www.objectmentor.com/publications/RUPvsXP.pdf, a chapter from Martin's forthcoming *Object Oriented Analysis and Design with Applications, Third Edition*, from Addison Wesley.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!