RUP and XP
Part II: Valuing Differences

by Gary Pollice
Evangelist, The Rational Unified Process
Rational Software

In the last issue of The Rational Edge*, we looked at the common ground between the Rational Unified Process (RUP®) and eXtreme Programming (XP). They certainly have a lot in common. This month, in Part Two of our comparison, we look at the last three XP practices and at some areas of RUP not covered by XP.*

There are three XP practices we deferred discussing until this issue. They are:

- *Small releases*
- *Collective code ownership*
- *Metaphor*

We will discuss each of these. But first, I'd like to point out that the set of XP practices we are talking about is the original twelve practices set forth by Kent Beck in his book Extreme Programming Explained: Embrace Change*, published by Addison-Wesley in 1999. As of March 15, 2001, there were several additional supporting practices listed on the Extreme Programming Roadmap page: http://c2.com/cgi/wiki?ExtremeProgrammingRoadmap. This indicates the dynamic and somewhat experimental nature of XP, which is not necessarily a bad thing. Any process needs to be dynamic and keep up-to-date with proven best practices. At the end of this article we will also look at some ways RUP and XP can work together to provide a good experience for software development project members.*

Small Releases: How Small and Released to Whom?

What is a release? Depending upon how you answer this question, RUP and XP can seem quite similar in their concepts of a release. RUP defines a release as: "...a stable, executable version of product, together with any artifacts necessary to use this release, such as release notes or installation instructions."[1] Furthermore, according to RUP, releases are either internal or external. By this definition, a "release" creates a forcing function that ensures a shippable product, rather than a system that is only 80 percent complete. XP defines a release as "...a pile of stories that together make business sense."[2] In much of the discussion about small releases on some XP Web pages, the practice of small releases seems to coincide with the practice of continuous integration.[3] If you interpret the stories to mean the code as well as any artifacts necessary to use the release, and you accept the release as either internal or external, then the RUP and the XP concepts of a release are almost identical.

RUP invites you to consider more than just code. A release, especially an external one to the customer, may prove useless unless accompanied by release notes, documentation, and training. XP addresses code and assumes the rest will appear. Since code is the primary artifact of XP, the others need to be derived from it. This implies certain skills that may not be obvious. For example, technical writers might need to be able to read the code to understand how the system works in order to produce the documentation.

I have talked with several people who assume the frequent releases in XP are all to be delivered to an external customer. In fact, XP is not clear about this. In *Extreme Programming Installed*, the authors urge you to get the code into the customer's hands as frequently as possible.[4] The fact is, in many organizations customers cannot accept frequent software updates. You need to weigh the benefits of frequent delivery against the impact on the customer's ability to be productive. When you are unable to deliver a system to the customer, you should consider other ways of getting feedback, such as usability testing. On a RUP-based project, you typically deliver to the customer in the last construction iteration as well as in the transition phase iterations.

Collective Code Ownership: Yours, Mine, and Ours

XP promotes "collective code ownership," which means that when you encounter code that needs to be changed, you change it. Everyone has permission to make changes to any part of the code. Not only do you have permission to make the changes -- you have the responsibility to make them.

There is an obvious benefit with this practice. When you find code that needs to be changed, you can change it and get on with your work without having to wait for someone else to change it. In order for this practice to work, however, you need to also practice "continuous integration" and maintain an extensive set of tests. If you change any code, then you need to run the tests and not check in your code changes until all tests pass.

But will collective ownership work everywhere? Probably not. Large systems contain too much content for a single person to understand it all at a detailed level. Some small systems often include code that is complex due to its domain or the function it performs. If a specialist is required, then collective ownership may not be appropriate. When a system is developed in a distributed environment, it is not possible for everyone to modify the code. In these cases, XP offers a supporting practice called "code stewardship."[5] With code stewardship, one person, the steward, has responsibility for the code, with input from other team members. There are no guidelines when to apply code stewardship instead of collective code ownership.

Collective code ownership provides a way for a team to change code rapidly when it needs changing. Are there any potential drawbacks to this? If you consider all the ways code is changed, then there are some things you may want to control less democratically, in a centralized way -- for example, when code is modified because it lacks some functionality. If a programmer is implementing a story (or a use case, or a scenario), and requires behavior from a class, collective code ownership allows the class to be modified on the spot. As long as the system is small enough for a programmer to understand all of the code, this should work fine. But as the system gets larger, it is possible that the same functionality might be added to code that exists somewhere else. This redundancy might be caught at some point and the code refactored, but it is certainly possible for it to go unnoticed and for the functionality to begin diverging.

You may want to start a project using collective code ownership to allow your team to move quickly. As long as you have good code management tools and effective testing, then it will work for you -- up to a point. As a project leader or manager, however, you need to be on the lookout for the point when the code base becomes too large or too specialized in places. When this happens, you may want to structure your system into an appropriate set of components and subsystems and ensure that specific team members are responsible for them. RUP provides guidance and other help on how to structure your system.

System Metaphor: It's Like Architecture

A metaphor is a figure of speech that allows us to make comparisons. It is one way that we learn: "A motorcycle is like a bicycle, but it has a motor attached." XP uses a system metaphor instead of RUP's formal architecture. This system metaphor is "...a simple shared story of how the system works. This story typically involves a handful of classes and patterns that shape the core flow of the system being built."[6]> Based on comparisons with familiar things, patterns help us understand new and unfamiliar things.

And indeed, the XP system metaphor may be a suitable replacement for architecture in some cases, but usually only in small systems. For many, if not most, software systems, you need more than a simple shared story. How much more you need depends upon many factors.

By contrast, RUP is an architecture-centric process.[7] Architecture is more than a metaphor, although it may include several metaphors. Architecture is concerned with structure, behavior, context, usage, functionality, performance, resilience, reuse, comprehensibility, constraints, trade-offs, and aesthetics. It is usually not possible to capture all of this in a simple shared story. Architecture does not provide a complete representation of the whole system. It concentrates on what is architecturally significant and important in reducing risks.

RUP provides a wealth of guidance on constructing and managing architecture. It helps the practitioner construct different views of the architecture for different purposes. [8] The different views are needed because there are different aspects that need to be highlighted and different people who need to view the architecture.

A RUP-based project will address architecture early. Often an executable architecture is produced during the Elaboration Phase. This provides an opportunity to evaluate solutions to critical technical risks, and the architecture is built upon during subsequent construction iterations.

An executable *architecture* is a partial implementation of the system, built to demonstrate selected system functions and properties, in particular those satisfying non-functional requirements. It is built during the *elaboration phase* to mitigate risks related to performance, throughput, capacity, reliability and other "ilities", so that the complete functional capability of the system may be added in the *construction phase* on a solid foundation, without fear of breakage. It is the intention of the Rational Unified Process that the executable architecture be built as an evolutionary prototype, with the intention of retaining what is found to work (and satisfies requirements), and making it part of the deliverable system.[7]

What's Not Covered by XP That Is in RUP?

Your project may be able to use XP for developing the code. If not, then you may need to add some additional process, but just enough to reduce your risks and ensure that you are able to deliver the right product to your customers on time.

However, when you look at a development project as a complete set of deliverables, code, documentation, training, and support, there are many things RUP addresses that are not considered in XP. Again, you need to determine whether they are needed for your specific project. The following list provides things you may need to consider. The list is not exhaustive. You can find additional information about these items in the Rational Unified Process.

- **Business modeling.** The whole subject of business modeling is absent from XP. Systems are deployed into an organization. Knowledge of the organization can be important when identifying the requirements and for understanding how well a solution might be accepted.
- **Project inception.** XP assumes the project has been justified and does not address how that justification takes place. In many organizations, a business case must be made before a project begins in earnest. RUP helps a team make its business case by

developing stakeholders' needs and a vision.

- **Deployment.** The whole area of system deployment is missing from XP. Any system needs supporting materials, minimally online documentation. Most need more. Commercial software products require packaging, distribution, user manuals, training materials, and a support organization. The RUP Deployment discipline provides guidance for practitioners on how to create appropriate materials and then use them.

Mix and Match for Best Results

Process diversity is important.[9] One size does not fit all projects. The process you use for your project should be appropriate for it. Consider what your project needs and adopt the right approach. Consider all aspects and risks. Use as much as you need, but neither too little nor too much.

RUP and XP provide two different approaches to software development projects. They complement each other in several ways. XP concentrates on code and techniques for a small team to create that code. It stresses face-to-face communication and places minimal effort on non-code artifacts. RUP is a process framework that you configure for different types of projects. It invites you to consider risks and risk mitigation techniques. RUP is often misinterpreted as being *heavy* because, as a framework, it provides process information for many types of projects. In fact, a configured instance of RUP may be very light, depending upon the risks that need to be addressed. It may incorporate some of the excellent techniques of XP and other processes if they are appropriate for the project at hand.

If my project is only about creating code, I may use just XP. However, almost all of the projects I work on require initial business decisions and planning, complete documentation, support, and deployment to customers. For this reason, I would more likely start with RUP and use the appropriate XP practices that will help my team move ahead quickly and mitigate real risks the project faces.

As a software engineer, I try to have a well-stocked toolbox of techniques, processes, and tools that help me succeed. I'm glad to have both RUP and XP as part of my collection. More techniques in my toolbox means that I can provide better value to my project and my organization. In addition, as a project manager or process engineer, I can create an instance of a process for a project that addresses the organization's need for controls while providing individual project members with an environment that can be fun and satisfying.

[1] Rational Unified Process Glossary.

[2] Kent Beck, *Extreme Programming Explained: Embrace Change*. Reading, MA: Addison-Wesley 1999, p.178.

[3] http://c2.com/cgi/wiki?FrequentReleases.

[4] Ron Jeffries et al, *Extreme Programming Installed*. Reading, MA: Addison-Wesley, 2000, p.50.

[5] http://c2.com/cgi/wiki?CodeStewardship.

[6] http://c2.com/cgi/wiki?SystemMetaphor.

[7] This is described in Philippe Kruchten, "The 4+1 View of Architecture." *IEEE Software*, November, 1995.

[8] This definition is taken from the Rational Unified Process glossary.

[9] For more information on process diversity, see Mikael Lindvall and Iona Rus, "Process Diversity in Software Development." *IEEE Software*, July/August 2000.