Using the Rational Unified Process for Small Projects:

Expanding Upon eXtreme Programming

Gary Pollice

Rational Software White Paper

TP 183, 3/01

Rational®
the software development company

Table of Contents

Abstract

The Rational Unified Process® or RUP® product is a complete software-development process framework that comes with several out-of-the-box instances. Processes derived from RUP vary from lightweight—addressing the needs of small projects with short product cycles—to more comprehensive processes addressing the broader needs of large, possibly distributed, project teams. Projects of all types and sizes have successfully used RUP. This white paper describes how to apply RUP in a lightweight manner to small projects. We describe how to effectively apply eXtreme Programming (XP) techniques within the broader context of a complete project.

Introduction

A Short Story

One morning, a manager came to me and asked if I could spend a few weeks setting up a simple information system for a new venture the company was starting. I was bored with my current project and yearned for the excitement of a start-up, so I jumped at the chance—I could move fast and develop great new solutions, unburdened by the bureaucracy and procedures of the large organization in which I worked.

Things started great. For the first 6 months, I worked mostly on my own—long, fun hours. My productivity was incredible and I did some of the best work of my career. The development cycles were fast and I typically produced some major new parts of the system every few weeks. Interactions with the users were simple and direct—we were all part of one close-knit team, and we could dispense with formalities and documentation. There was also little formality of design; the code was the design, and the design was the code. Things were great!

Things were great, for a while. As the system grew, there was more work to do. Existing code had to evolve as the problems changed and we refined our notions of what we needed to do. I hired several people to help with development. We worked as a single unit, often in pairs on parts of the problem. It enhanced communication and we could dispense with the formality.

A year passed.

We continued to add people. The team grew to three, then five, then seven. Every time a new person started, there was a lengthy learning curve and, without the benefit of experience, it became more difficult to understand and explain the whole system, even as an overview. We started capturing the white-board diagrams that showed the overall structure of the system, and the major concepts and interfaces more formally.

We still used testing as the primary vehicle for verifying that the system did what it needed to do. With many new people on the "user" side, we found that the informal requirements and personal relationships that worked in the early days of the project were no longer enough. It took longer to figure out what we were supposed to build. As a result, we kept written records of discussions so we did not have to continually recall what we had decided. We also found that describing the requirements and usage scenarios helped to educate new users on the system.

As the system grew in size and complexity, something unexpected happened—the architecture of the system required conscious attention. In the early days, the architecture could exist largely in my head, then later on a few scribbled notes or on flip charts. However, with more people on the project it was harder to keep the architecture under control. Since not everyone had the same historical perspective as I did, they were unable to see the implications of a particular change to the architecture. We had to define the architectural constraints on the system in more precise terms. Any changes that might have affected the architecture required team consensus and, ultimately, my approval. We discovered this the hard way, and there were some difficult lessons learned before we really admitted to ourselves that architecture was important.

This is a true story. It describes only some of the painful experiences of this project. The experiences are unusual only in one respect: several of us were there from beginning to end, over a period of years. Usually people come and go on a project, and do not see the downstream impact of their actions.

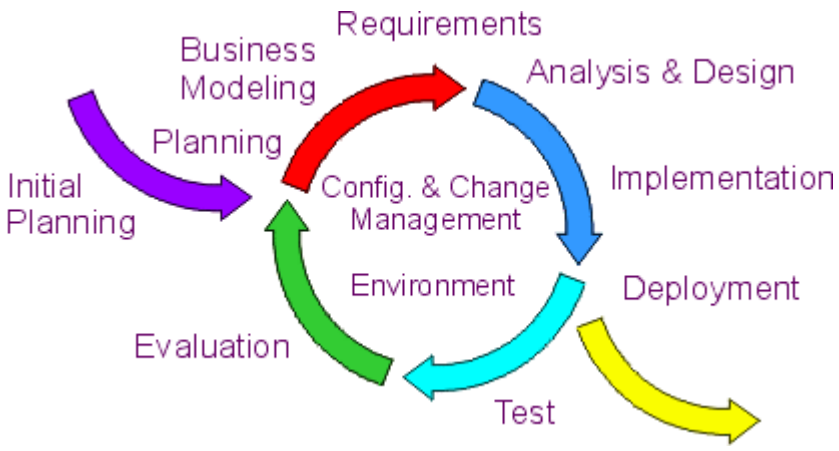
This project could have been helped with a little bit of process. Too much process gets in the way, but lack of process brings new risks. Like the person who invests in high-risk stocks seeing only the high returns, groups who use too little process, ignoring key risks in their project environment, are "hoping for the best but unprepared for the worst."

Overview

This paper discusses how to apply process to projects like the one just described. We focus on getting the "right level" of process. Understanding the challenges faced by the development team and the business environment in which it operates, derives the right level of process formality. Once we understand these challenges, we supply just enough process to mitigate the risks. There is no one-size-fits-all process, lightweight or otherwise. In the following sections, we explore the idea that the right level of process is a function of risk.

We focus on how to get the right level of process by using two popular methodologies: the Rational Unified Process or RUP and eXtreme Programming (XP). We show how to tailor the RUP to a small project and how it addresses many areas not considered by XP. The combination gives a project team the guidance needed for mitigating the risks and achieving their goal of delivering a software product.

RUP is a process framework developed by Rational Software. It's an iterative development methodology based upon six industry-proven best practices (see RUP appendix). Over time, a RUP-based project goes through four phases: Inception, Elaboration, Construction, and Transition. Each phase contains one or more iterations. In each iteration, you expend effort in various amounts to each of several disciplines (or workflows) such as Requirements, Analysis and Design, Testing, and so forth. The key driver for RUP is *risk mitigation*. RUP has been refined by use in thousands of projects with thousands of Rational customers and partners. The following diagram illustrates the flow through a typical iteration:



Typical Iteration Flow

As an example of how risk can shape a process, we might ask if we should model the business. If there is some significant risk that failing to understand the business will cause us to build the wrong system, we should probably perform some amount of business modeling. Do we need to be formal about the modeling effort? That depends upon our audience—if a small team will use the result informally, we might just make some informal notes. If others in the organization are going to use the results or review them, we probably have to invest some extra effort, and focus more on the correctness and understandability of the presentation.

You can customize RUP to fit the needs of almost any project. If none of the out-of-the-box processes, or roadmaps, fits your specific needs, you can easily produce your own roadmap. A roadmap describes how the project plans to use the process and, therefore, represents a specific process instance for that project. What this means is that RUP can be as light or as heavy as necessary, which we illustrate in this paper.

XP is a lightweight code-centric process for small projects (see XP appendix). It is the brainchild of Kent Beck and came to the software industry's attention on the C3 payroll project at Chrysler Corporation around 1997. Like the RUP, it is based upon iterations that embody several practices such as Small Releases, Simple Design, Testing, and Continuous Integration. XP promotes several techniques that are effective for the appropriate projects and circumstances; however, there are hidden assumptions, activities, and roles.

RUP and XP come from different philosophies. RUP is a framework of process components, methods, and techniques that you can apply to any specific software project; we expect the user to specialize RUP. XP, on the other hand, is a more

constrained process that needs additions to make it fit a complete development project. These differences explain the perception in the overall software development community: the big system people see RUP as the answer to their problems; the small system community sees XP as the solution to their problems. Our experience indicates that most software projects are somewhere in between—trying to achieve the right level of process for their situation. Neither end of the spectrum is sufficient for them.

When you combine the breadth of RUP with some of the XP techniques, you achieve the right amount of process that appeals to all members of a project and addresses all major project risks. For a small project team working in a relatively high-trust environment where the user is an integral part of the team XP can work very well. As the team becomes more distributed and the code base grows, or the architecture is not well defined, you need something else. You need more than XP for projects that have a "contractual" style of user interaction. RUP is a framework from which you can extend XP with a more robust set of techniques when they are required.

The remainder of this paper describes a small process based on the four phases of RUP. In each, we identify the activities and artifacts produced.[1] Although RUP and XP identify different roles and responsibilities, we do not address these differences here. For any organization or project, the actual project members must be associated with the proper roles in the process.

Project Start — Inception

Inception is significant for new development efforts, where you must address important business and requirement risks before the project can proceed. For projects focused on enhancements to an existing system, the Inception phase is shorter, but is still focused on ensuring that the project is both worth doing and possible.

During Inception, you make the business case for building the software. The *Vision* is a key artifact produced during Inception. It is a high-level description of the system. It tells everyone what the system is, and may also tell who will use it, why it will be used, what features must be present, and what constraints exist. The *Vision* may be very short, perhaps only a paragraph or two. Often the *Vision* contains the critical features the software must provide to the customer.

The following example shows a very short *Vision* written for the project to re-engineer the Rational external Web site.

> *To drive Rational's position as the worldwide leader in e-development (tools, services, and best practices), by enhancing customer relationships through a dynamic, personalized Web presence providing visitor self-service, support, and targeted content. The new process and enabling technologies will empower content providers to speed publishing and quality of content through a simplified, automated solution.*

Four essential Inception activities specified in RUP are:

- **Formulate the scope of the project** — If we are going to produce a system, we need to know what it is and how it will satisfy the stakeholders. In this activity, we capture the context and most important requirements in enough detail to derive acceptance criteria for the product.
- **Plan and prepare the business case** — With the *Vision* as a guide, we define our risk mitigation strategy, develop an initial project plan, and identify known cost, schedule, and profitability trade-offs.
- **Synthesize a candidate architecture** — If the system under consideration is something with little novelty and has a well-understood architecture, you may skip this step. As soon as we know what the customer requires, we allocate time to investigate potential candidate architectures. New technology brings with it the potential for new and improved solutions to software problems. Spending time early in the process to evaluate buy versus build trade-offs, as well as selecting technologies, and perhaps developing an initial prototype, can reduce some major risks for the project.
- **Prepare the project environment** — Any project needs a project environment. Whether you use some of the XP techniques, such as pair programming, or more traditional techniques you need to determine the physical resources, software tools, and procedures the team will follow.

[1] XP defines three phases: Exploration, Commitment, and Steering. These do not map well to RUP phases so we choose to use the four RUP phases to describe the process.

It does not take a lot of "process time" to perform Inception on a small project. You can often complete Inception in a couple of days or less. The following sections describe the expected artifacts, other than the *Vision*, of this phase.

An approved Business Case

Stakeholders have the chance to agree that, from a business perspective, the project is worth doing. RUP and XP agree that it is better to find out early that the project is not worth doing rather than spend valuable resources on a doomed project. XP, as it is described in *Planning Extreme Programming*[1], is fuzzy on how projects come into being and what roles are involved (it seems clearest in the context of an existing business or system), but in its Exploration phase XP deals with equivalent RUP Inception artifacts.

Whether you consider the business issues informally as in XP, or make the business case a first-class project artifact, as with RUP, you need to consider them.

Risk List

You maintain the Risk List throughout the project. This can be a simple list of risks with planned mitigation strategies. The risks are prioritized. Anyone associated with the project can see what the risks are and how you address them at any point in time. Kent Beck describes a set of risks that XP addresses and how it addresses them, but no general approach to risk handling is provided.[2]

Preliminary Project Plan

Resource estimates, scope, and phase plans are included in this plan. On any project, these estimates continually change and you must monitor them.

Project Acceptance Plan

Whether you have a formal plan or not depends upon the type of project. You must decide how the customer will evaluate the success of the project. On an XP project, this takes the form of acceptance tests created by the customer. In a more general process, the customer may not actually construct the tests, but the criteria for acceptance must be driven by the customer directly or through another role, such as the System Analyst, who interacts directly with the customer. There may be other acceptance criteria such as the production of end-user documentation and help, which XP does not cover.

A plan for the initial Elaboration iteration

In a RUP-based project, you plan each iteration in detail at the end of the previous iteration. At iteration end, you evaluate the progress against the criteria designated at the start of the iteration. XP provides some good techniques for monitoring and measuring the success of an iteration. The metrics are simple and you can easily incorporate them into the iteration plan and evaluation criteria.

Elaboration

The goal of the Elaboration phase is to baseline the architecture of the system to provide a stable basis for the bulk of the design and implementation effort in the Construction phase. The architecture evolves out of a consideration of the most significant requirements (those that have a great impact on the architecture of the system) and an assessment of risk. The stability of the architecture is evaluated through one or more architectural prototypes.

In RUP, design activities focus on the notion of system architecture and, for software-intensive systems, software architecture. Using component architectures is one of the six best practices of software development embodied in RUP, which recommends

[1] This is one of the three book currently published on eXtreme Programming.

[2] In *eXtreme Programming eXplained*, Kent Beck describes eight risks and how XP addresses the risks (pp. 3-5). We invite the reader to look at them and determine if they are sufficient. We believe there are many other risks and a general risk-handling strategy is a necessary part of any process.

spending time developing and maintaining the architecture. The time spent on this effort mitigates the risks associated with a brittle and inflexible system.

XP replaces the notion of architecture by "metaphor." The metaphor captures part of the architecture, whereas the rest of the architecture evolves as a natural result of code development. XP assumes that architecture emerges from producing the simplest design and continually refactoring the code.

In RUP, architecture is more than metaphor. During Elaboration, you construct executable architectures, from which it is possible to reduce many of the risks associated with meeting non-functional requirements such as performance, reliability, and robustness. In reading the XP literature, it is possible to infer that some of what RUP prescribes for Elaboration, specifically undue concentration on what XP calls infrastructure, is wasted effort. XP says that effort expended building infrastructure in advance of the need for it, leads to overly complex solutions and the production of things that have no value to the customer. In RUP, architecture and infrastructure are not identical.

The approach to architecture is quite different between RUP and XP. RUP advises that you pay attention to architecture to avoid the risks associated with increased scope over time, additional project size, and the addition of new technologies. XP assumes an existing architecture or that the architecture is simple enough or understood well enough that it can evolve as you code. XP advises not to design for tomorrow, but to implement for today. The belief is that tomorrow will take care of itself if you keep the design as simple as possible. RUP invites you to assess the risks of such a proposition. If the system or parts of it have to be rewritten in the future, XP indicates that it is still better and often less expensive than planning for the possibility now. For some systems, this will be true and, using RUP, your consideration of risk during the Elaboration phase will lead you to this conclusion. RUP does not assert this truth for all systems and experience suggests that for larger, more complex and unprecedented systems, it can be disastrous.

While it is true that paying too much attention to future possibilities that may never occur can be wasteful, paying the right amount of attention to the future is a prudent thing to do. How many companies can afford to continually rewrite or even refactor code?

On any project, you should do at least these three activities during Elaboration:

- **Define, validate, and baseline the architecture** — Use the risk list to develop the candidate architecture from the Inception phase. We are interested in making sure the envisioned software is feasible. If there is little novelty in the chosen technology or little complexity to the system, this task will not take long. If you are adding to an existing system, the task may not be necessary if no changes to the existing architecture are needed. When there are real architectural risks present, you do not want to leave the architecture to chance.

  As a part of this activity, you may perform some component selection and make buy/build/reuse decisions. If this requires a lot of effort, you may break this out into a separate activity.

- **Refine the Vision** — During the Inception phase, you developed a Vision. As you determine the feasibility of the project, and the stakeholders have time to review and comment on the system, there may be changes to the Vision document and requirements. Revisions to it and the requirements typically occur during Elaboration. By the end of Elaboration, you have established a solid understanding of the most critical use cases that drive architectural and planning decisions. Stakeholders need to agree that the current vision could happen if you execute the current plan to develop the complete system, in the context of the current architecture. The amount of change should lessen during subsequent iterations, but you will want to allocate some amount of time in each iteration for requirements management.

- **Create and baseline iteration plans for the Construction phase** — Fill in the details of your plan now. At the end of each Construction iteration, you revisit the plans and adjust as necessary. Adjustments usually occur because the effort was incorrectly estimated, the business environment changed, or the requirements changed. Prioritize the use cases, scenarios, and technical efforts, and then assign them to iterations. At the end of each iteration you plan to have a working product that provides value to your stakeholders.

You may perform other activities during Elaboration. We recommend establishing the testing environment and starting to develop tests. While detailed code may not exist, you may still design and perhaps implement integration tests. Programmers should be ready to develop unit tests and know how to use the testing tools selected for the project. XP recommends writing the test before the code. This is a good idea, especially when you are adding to an existing body of code. However you choose to do your testing, the time to establish a regular testing regimen is in Elaboration.

The Elaboration phase described by RUP contains elements of the Exploration and Commitment phases of XP. The XP approach to dealing with technical risks, such as novelty and complexity, is the "spike" solution, i.e. taking some time to experiment in order to estimate effort. This technique is effective in many cases, when there is a larger risk not embodied in a single use case or story, you need to apply a little more effort to ensure system success and accurate effort estimation.

You usually update artifacts, such as Requirements and the Risk List from the Inception phase, during Elaboration. Artifacts that may appear during Elaboration are:

- **Software Architecture Document (SAD)** — The SAD is a composite artifact that provides a single source of technical information throughout the project. At the end of Elaboration, it may contain detailed descriptions for the architecturally significant use cases and identification of key mechanisms and design elements. When the project enhances an existing system, you may use a prior SAD or you may decide there is little risk in not having the document. In all cases, you should perform the thought processes that lead to the document.
- **Iteration Plans for Construction** — You plan the number of Construction iterations during Elaboration. Each iteration has specific use cases, scenarios, and other work items assigned to it. This information is captured and baselined in the iteration plans. Review and approve plans as part of the exit criteria for Elaboration.

On very small, brief projects, you might merge the Elaboration iteration with Inception and Construction. The essential activities are still performed, but resources for iteration planning and reviews are reduced.

Initial Use-Case Model

While this may sound formal and intimidating, it is quite straightforward. Use cases correspond to the "stories" written by the customer in XP. The difference is that a use case is a complete set of actions initiated by an actor, someone, or something outside of the system that provides visible value. The use case may contain several XP stories. In order to define the scope of the project, RUP recommends identifying the use cases and actors during Inception. Focusing on complete sets of actions from the user's point of view helps partition the system into parts that provide value. This helps determine the appropriate implementation features so we have something to deliver to the customer at the end of each iteration (except possibly the early Inception and Elaboration iterations).

Both RUP and XP help us ensure that we are not in the position of having 80% of the complete system done, but nothing completed in deliverable form. We always want to be able to release the system to provide some value to the customer.

The use-case model, at this point, identifies use cases and actors with little or no supporting detail. It can be simple text or UML (Unified Modeling Language) diagrams drawn by hand or with a drawing tool. This model helps us ensure that we have included the right features for the stakeholders and have not forgotten anything, and allows us to easily see the whole system. Use cases are prioritized based upon several factors such as risk, importance to the customer, and technical difficulty.

None of the artifacts for Inception needs to be overly formal or large. Make them as simple or as formal as you need. XP contains guidance on planning and system acceptance while RUP adds a little more during the early part of a project. This small addition may pay big dividends by addressing a more complete set of risks.

Construction

The goal of Construction is to complete the development of the system. The Construction phase is, in some sense, a manufacturing process, where you emphasize managing resources and controlling operations to optimize costs, schedules, and quality. In this sense, the management mindset undergoes a transition from the development of intellectual property during Inception and Elaboration, to the development of deployable products during Construction and Transition.

XP concentrates on Construction. The Construction phase is where you produce code. The XP phases are for planning purposes, but the focus of XP is on building the code.

Each Construction iteration has three essential activities:

- **Manage resources and control process** — Everyone needs to know who will do what and when. You must make sure that the workload does not exceed your capacity and that work is progressing according to schedule.
- **Develop and test components** — You build the components required to satisfy the use cases, scenarios, and other functionality for the iteration. You test them with unit and integration tests.

- **Assess the iteration** — Upon iteration completion, you need to determine if you satisfied the goals of the iteration. If not, you need to re-prioritize and manage the scope to meet your delivery date.

Different types of systems require different techniques. RUP offers the software engineer different guidelines and help to build the right components. Requirements, in the form of use cases and supplementary (non-functional) requirements, are detailed enough for the engineer to do the work. Several activities in RUP provide guidance on designing, implementing, and testing different kinds of components. An experienced software engineer does not need to look at these activities in detail. A less experienced engineer will find a wealth of help on best practices. Each team member can go as shallow or as deep into the process as needed. They all, however, look at a single process knowledge base.

In XP, stories drive the implementation. In the book *Extreme Programming Installed*, Jeffries, et al say that the stories are "promises for conversation" with the programmers.[1] Continual, effective communication is good. Although there are always details that need clarification, if the stories are not detailed enough for programmers to do most of their work, they are not ready. Use cases must be detailed enough for programmers to implement the use case. In many cases, programmers help write the technical details of the use case. Jeffries, et al also say that the conversations will be documented and attached to the story. RUP suggests this as well, except in the form of a use-case specification, which can be as informal as needed. Capture and management of the outcome of the conversations is a task you must manage.

The XP forté is Construction. There are some nuggets of wisdom and guidance appropriate for most teams. The most noteworthy XP practices are:

- **Testing** — Programmers continually write tests to go along with their code. The tests reflect the stories. XP urges you to write the tests first, which is an excellent practice because it forces you to deeply understand the stories and ask more questions where necessary. Whether you write them before or after you code, write them! Add them to your test suite and make sure to run them every time code changes.
- **Refactoring** — Restructure the system continually, without changing its behavior, to make it simpler or add flexibility. You need to determine if this is a good practice for your team. What is simple to one person may be complex to another. There is an example where two very smart engineers on a project spent every evening rewriting the other's code because they thought it was too complex. As a by-product, they continually broke the builds the next day for the rest of the team. Testing helps, but the team would have been better off had they not gotten into the coding wars.
- **Pair programming** — XP claims that pair programming produces better code in less time. There is evidence that this is the case.[2] There are many human and environmental factors to consider if you implement this practice. Are the programmers willing to try this? Is your physical environment such that there is room for two programmers to work effectively at a single workstation? What do you do with programmers who are telecommuting or in other locations?
- **Continuous integration** — Integrate and build the system often, perhaps more than once a day. This is a great way to ensure the structural integrity of the code and it allows for continual quality monitoring through integration testing.
- **Collective ownership** — Anyone has permission to change any code at any time. XP relies on the fact that a good set of unit tests will reduce the risk of this practice. The benefits of having everybody familiar with everything does not scale beyond some point—ten thousand lines of code; twenty thousand; surely less than fifty thousand?
- **Simple design** — As with refactoring, the design of the system is continually modified to remove complexity. Once again, you need to determine how large this might scale up to before it does not work. If you spend time during Elaboration designing the architecture, we believe that the simple design will emerge and become stable much sooner.
- **Coding standards** — This is always a good practice. It does not matter what the standards are as long as you have them and everyone agrees to use them.

RUP and XP both agree that you must manage (or steer) iterations. Metrics can provide good planning information since they help you choose what is best for your team. There are three things to measure: time, size, and defects. From this you can

[1] This description is attributed to Allistair Cockburn.

[2] *Strengthening the Case for Pair Programming*, IEEE Software, July/August, 2000.

obtain all sorts of interesting statistics. XP provides simple metrics to use to determine progress and predict accomplishment. These metrics center around the number of stories finished, the number of tests passed, and the trends in these statistics. XP makes a strong case for using a minimal amount of metrics since looking at more will not necessarily improve your project's chances for success. RUP provides guidelines on what you might measure and how to measure, and gives examples of metrics. In all cases, metrics must be simple, objective, easy to collect, easy to interpret, and difficult to misinterpret.

What artifacts appear during Construction iterations? Depending upon whether the iteration is an early or late Construction iteration, you might create any of the following:

- **Component** — A component represents a piece of software code (source, binary, or executable), or a file containing information; for example, a startup file or a ReadMe file. A component can also be an aggregate of other components, such as an application consisting of several executables.
- **Training materials** — Based on use cases, produce a preliminary draft of user manuals and other training materials as early as possible if the system has a strong user interface aspect.
- **Deployment Plan** — The customer needs a system. The Deployment Plan describes the set of tasks necessary to install, test, and effectively transition the product to the user community. For Web-centric systems, we have found that the Deployment Plan has increased importance.
- **Transition Phase Iteration Plan** — As you approach the time to deploy the software to your users, you complete and review the Transition Phase Iteration Plan.

Is it really all about the code?

Besides the differences in approach to architecture between RUP and XP, there are others. One is the way you communicate the design. XP indicates that the code is the design and the design is the code. It is true that the code is always in agreement with itself. We believe that some effort to capture and communicate the design, other than the code, is time well spent. This short story might illuminate this.

One engineer had two experiences on software projects where the design was in the code and it was the only place to find design information. Both of these projects were compiler related: one was to improve and maintain an optimizer for an Ada compiler, and the other was a project to port the front-end of a compiler to a new platform and link a third-party code generator to it.

Compiler technology is complicated, but well known. On both projects, the engineer wanted an overview of the design and implementation of the compiler (or optimizer). In each case he received a pile of source code listings, several inches thick, and was told to "look in there." He would have given anything for a few well-constructed diagrams with some supporting text. The optimizer project was never completed. The compiler project did get finished successfully, with great code quality because of an extensive set of tests developed along with the code. The engineer spent days walking through the code in a debugger to try to understand what it did. The personal cost of minor burnout and the cost to the team was not worth it. We did not have the option of stopping after 40 hours as XP suggests and we expended a lot of heroic effort to get the job done.

The principal problem with having only the code is that the code—no matter how well documented—does not tell you the problem it solves, it only conveys the solution to the problem. Some documentation of the requirements goes a long way to explain the original goals long after the original users and developers have moved on. To maintain a system, you often need to know what the original project team had in mind. Some high-level design documents are similar – often the code is at too low a level of abstraction to really tell what the system, as a whole, is trying to do. This is especially true in Object-Oriented systems in which the thread of execution is hard or impossible to follow by just looking at the classes. Design documents guide you about where to look when there are problems later—and there are *always* problems later.

The moral of the story is that some time spent capturing and maintaining design documents really does help. It reduces the risk of misunderstanding and it can speed up the development. The XP approach is to spend a few minutes sketching out the design or to use CRC cards.[1] The team does not maintain these, but goes to work on the code. There is an implicit assumption that the tasks are simple enough that we already know how to proceed. Even if we do, the next person coming along might not be so lucky. RUP suggests you spend a little more time capturing and maintaining these design artifacts.

[1] CRC (Class, Responsibility, and Collaboration) cards, developed by Kent Beck and Ward Cunningham, teach practitioners the principles of Object-Oriented design.

Transition

The focus of Transition is to ensure that software is available for its end users. The Transition phase includes testing the product in preparation for release and making minor adjustments based on user feedback. At this point in the lifecycle, user feedback needs to focus mainly on fine-tuning the product, configuring, installing, and usability issues.

Release early and often is a great idea. But, what do we mean by a release? XP is vague about this and does not address the manufacturing issues necessary to release commercial software. You may be able to shortcut some of the issues on an internal project; but even then, you need documentation, training, and so forth. What about support and change management? Is it realistic to expect that the on-site customer will control these too? Bruce Conrad points out in his InfoWorld review of XP[1] that customers may not want to get software that is continually changing. You must weigh the benefit of getting changes to the customer quickly with the disadvantages of change and possible destabilization.

When you decide to release, you need to provide your end-user with more than just code. The activities and artifacts in Transition guide you through this part of the software development process. The activities center on getting a usable product to your customer. The essential Transition activities are the following:

- **Finalize end-user support material** — This activity can be as simple as checking items off a list, but you need to make sure that your organization is prepared to support your customer.
- **Test the product deliverable in a customer environment** — If you can simulate the customer environment at your location, do so. Otherwise, go to the customer and install your software to ensure that it works. You do not want to be in the unenviable position of saying to the customer "but it worked on our system".
- **Fine-tune the product based upon customer feedback** — If possible, plan one or more beta-testing periods where you deliver the software to a limited number of customers. If you do this, you need to manage the beta-testing period and consider the customer feedback in your "end game".
- **Deliver the final product to the end user** — Depending upon the type of software product and release, there are many details about packaging, manufacturing, and other production issues to address. Rarely do you just put the software in a directory and send out mail letting your customer community know that the software is there for them.

As with most of the other phases, the amount of formality and complexity of your process will vary. However, if you do not pay attention to deployment details, you can negate weeks and months of good development effort and end up with a product that is a failure in your target marketplace.

You can produce several artifacts during the Transition phase. If your product is one that will have future releases (and how many do not?), you will have begun identifying features and defect fixes for the next release. The essential artifacts for any project are:

- **Deployment Plan** — Finalize the Deployment Plan you started in the Construction phase and use it as the roadmap for customer delivery.
- **Release Notes** — It is a rare software product that does not have last minute instructions for the end user. Plan on it and have a usable, consistent format for your notes.
- **Training Materials and Documentation** — There is a broad spectrum of forms these materials can take. Will you provide everything on-line? Will you have a tutorial? Is your product help complete and usable? Do not assume that your customer will know what you know. Your success depends on helping them succeed.

Summary

Building software is more than writing code. A software development process must focus on all activities necessary to deliver quality to your customers. A complete process does not have to be heavy. We have shown how you can have a small, yet complete, process by focusing on the essential activities and artifacts for your project. Perform an activity or produce an artifact if it helps mitigate risk on your project. Use as much, or as little, process and formality as you need for your project team and your organization.

[1] http://www.infoworld.com/articles/mt/xml/00/07/24/000724mtextreme.xml

RUP and XP are not necessarily exclusive. By incorporating techniques from both methods, you can arrive at a process that helps you deliver better quality software quicker than you do today. Robert Martin describes a process called the *dX process*, which he claims to be RUP compliant.[1] It is an instance of a process built from the RUP framework.

A good software process incorporates industry-proven best practices. Best practices are those that have been tested over time, in real software development organizations. XP is a method that is the focus of much attention today. It is code-centric and offers a promise of minimal process overhead and maximal productivity. There are many techniques in XP that warrant consideration and adoption in the right situations.

XP focuses on stories, tests, and code—it discusses planning at some length, but treats the capture of plans lightly. XP implies that you may produce other things such as "do a CRC design with a few cards, or sketch some UML…" or "Don't produce documents or other artifacts that aren't being used …", but treats them in passing. RUP invites you to consider producing only what is useful and required as you formulate and update the development plan, and it identifies what these things might be.

RUP is a process that addresses the complete software development lifecycle. It focuses on best practices that have evolved on thousands of projects. We encourage the investigation and invention of new techniques that lead to best practices. As new best practices emerge, we look forward to incorporating them into RUP.

Appendix A: The Rational Unified Process

The Rational Unified Process or 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. RUP also provides information to help you 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.

RUP provides guidance for all 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.

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

1. Develop iteratively
2. Manage requirements
3. Use component-based architectures
4. Model visually
5. Continuously verify quality
6. Control change

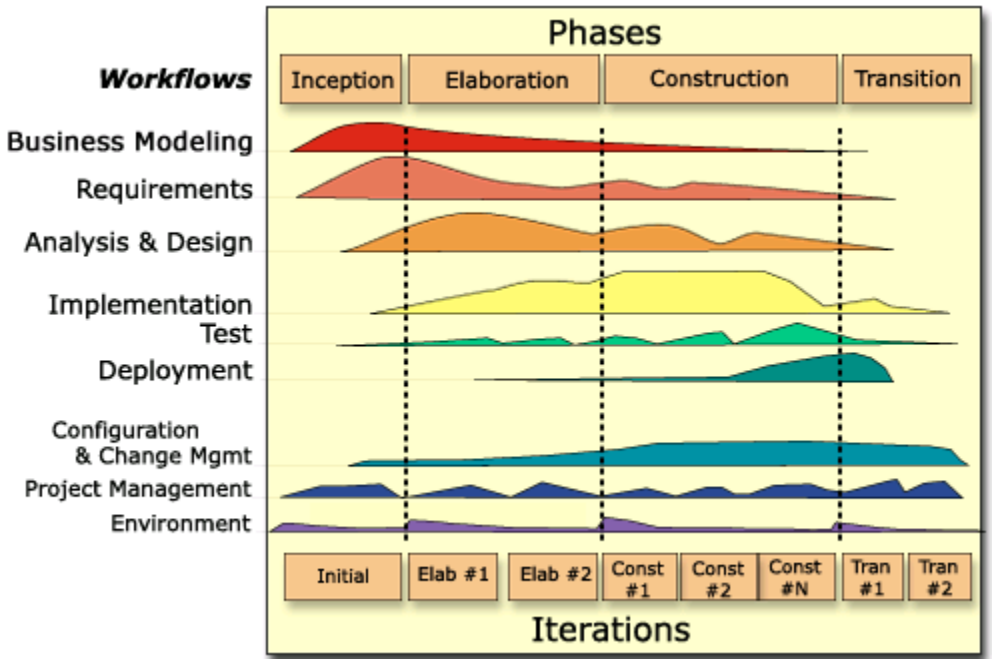
These best practices are woven into the Rational Unified Process definitions of:

- *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

RUP is an iterative process that identifies four phases of any software development project. Over time, the project goes through Inception, Elaboration, Construction, and Transition phases. Each phase contains one or more iterations where you

[1] http://www.objectmentor.com/publications/RUPvsXP.pdf. This is a chapter from an unpublished book by Martin, Booch, and Newkirk.

produce an executable, but perhaps incomplete system (except possibly in the Inception phase). During each iteration you perform activities from several disciplines in varying levels of detail. The following is an overview diagram of the RUP.



RUP Overview Diagram

The Rational Unified Process, An Introduction, Second Edition is a good overview of the RUP. You can find further information and an evaluation of the RUP on the Rational Software Web site at: www.rational.com.

Appendix B: eXtreme Programming

eXtreme Programming (XP) is a software development discipline developed by Kent Beck in 1996. It is based upon four values: communication, simplicity, feedback, and courage. It stresses continual communication between the customer and development team members by having an on-site customer while development progresses. The on-site customer decides what will be built and in what order. You embody simplicity 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 with 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. You write tests before you write code.
- **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 of the code. One writes the code, or *drives*, while the other reviews the code for correctness and understandability at the same time.
- **Collective ownership** — Everyone owns all of the code. This means 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 for 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.

There are currently three books available that describe XP:

1. eXtreme Programming Explained
2. Extreme Programming Installed
3. Planning Extreme Programming

Several Web sites are available for further information on XP.

Rational®
the software development company

Dual Headquarters:

Rational Software
18880 Homestead Road
Cupertino, CA 95014
Tel: (408) 863-9900

Rational Software
20 Maguire Road
Lexington, MA 02421
Tel: (781) 676-2400

Toll-free: (800) 728-1212
E-mail: info@rational.com
Web: www.rational.com
International Locations: www.rational.com/worldwide