Transitioning from waterfall to iterative development

Per Kroll

April 16, 2004

from The Rational Edge: The model for a perfect iterative development methodology is in many ways radically different from the perfect model for waterfall development. But, in practice, no team applies either approach strictly according to its model. This article explains why teams might decide to move gradually from a waterfall-like approach to a more iterative one, while outlining steps that can help.

Most software teams still use a waterfall process for development projects. Taking an extreme waterfall approach means that you complete a number of phases in a strictly ordered sequence: requirements analysis, design, implementation/integration, and then testing. You also defer testing until the end of the project lifecycle, when problems tend to be tough and expensive to resolve; these problems can also pose serious threats to release deadlines and leave key team members idle for extended periods of time.

In practice, most teams use a modified waterfall approach, breaking the project down into two or more parts, sometimes called phases or stages. This helps to simplify integration, get testers testing earlier, and provide an earlier reading on project status. This approach also breaks up the code into manageable pieces and minimizes the integration code, in the form of stubs and drivers, required for testing. In addition, this approach allows you to prototype areas you deem risky or difficult and to use feedback from each stage to modify your design. However, that runs counter to the thinking behind the waterfall approach: Many design teams would view modifying the design...
after Stage 1 as a failure of their initial design or requirements process. And although a modified waterfall approach does not preclude the use of feedback, it does not facilitate, accommodate, or encourage it. And finally, the desire to minimize risk does not typically drive a waterfall project. This article will explore the improvements that an "iterative" approach to the software development process offers over the traditional waterfall approach.

**Advantages of an iterative approach**

In contrast, an iterative approach -- like the one embodied in IBM Rational Unified Process® or RUP® -- involves a sequence of incremental steps, or iterations. Each iteration includes some, or most, of the development disciplines (requirements, analysis, design, implementation, and so on), as you can see in Figure 1. Each iteration also has a well-defined set of objectives and produces a partial working implementation of the final system. And each successive iteration builds on the work of previous iterations to evolve and refine the system until the final product is complete.

Early iterations emphasize requirements as well as analysis and design; later iterations emphasize implementation and testing.

**Figure 1: Iterative development with RUP.** Each iteration includes requirements, analysis, design, implementation and testing activities. Also, each iteration builds on the work of previous iterations to produce an executable that is one step closer to the final product.

The iterative approach has proven itself superior to the waterfall approach for a number of reasons:

- **It accommodates changing requirements.** Changes in requirements and "feature creep" -- the addition of features that are technology- or customer-driven -- have always been primary sources of project trouble, leading to late delivery, dissatisfied customers, and frustrated developers. To address these problems, teams who use an iterative approach focus on producing and demonstrating executable software in the first few weeks, which forces a review of requirements and helps to pare them down to essentials.

- **Integration is not one "big bang" at the end of a project.** Leaving integration to the end almost always results in time-consuming rework -- sometimes up to 40 percent of the total project effort. To avoid this, each iteration ends by integrating building blocks; this happens progressively and continuously, minimizing later rework.

- **Early iterations expose risks.** An iterative approach helps the team mitigate risks in early iterations, which include testing for all process components. As each iteration engages many aspects of the project -- tools, off-the-shelf software, team members' skills, and so on -- teams can quickly discover whether perceived risks are real and uncover new risks they did not suspect, at a time when these problems are relatively easy and less costly to address.

- **Management can make tactical changes to the product.** Iterative development quickly produces an executable architecture (albeit of limited functionality) that can be readily translated into a "lite" or "modified" product for quick release to counter a competitor's move.

- **It facilitates reuse.** It is easier to identify common parts as you partially design or implement them in iterations than to recognize them during planning. Design reviews in early iterations
allow architects to spot potential opportunities for reuse, and then develop and mature common code for these opportunities in subsequent iterations.

• **You can find and correct defects over several iterations.** This results in a robust architecture and a high-quality application. You can detect flaws even in early iterations rather than during a massive testing phase at the end. And you can discover performance bottlenecks when you can still address them without destroying your schedule -- or creating panic on the eve of delivery.

• **It facilitates better use of project personnel.** Many organizations match their waterfall approach with a pipeline organization: Analysts send the completed requirements to designers, who send a completed design to programmers, who send components to integrators, who send a system for test to testers. These multiple handoffs not only create errors and misunderstandings; they also make people feel less responsible for the final product. An iterative process encourages a wider scope of activities for team members, allowing them to play many roles. Project managers can better use available staff and eliminate risky handoffs.

• **Team members learn along the way.** Those working on iterative projects have many opportunities during the development lifecycle to learn from their mistakes and improve their skills from one iteration to another. By assessing each iteration, project managers can discover training opportunities for team members. In contrast, those working on waterfall projects are typically confined to narrow specialties and have only one shot at design, coding, or testing.

• **You can refine the development process along the way.** End-of-iteration assessments not only reveal the status of the project from a product or scheduling perspective; they also help managers analyze how to improve both the organization and the process in the next iteration.

Some project managers resist adopting an iterative approach, seeing it as a form of endless, uncontrolled hacking. However, in RUP the entire project is tightly controlled. The number, duration, and objectives of iterations are carefully planned; and the tasks and responsibilities of participants are well defined. In addition, objective measures of progress are captured. Although the team does rework some things from one iteration to the next, this work, too, is carefully controlled.

**Four steps for a transition**

Most waterfall projects divide the development work into phases or stages; we can also view this as a first step toward iterative design. But then, to move to an iterative approach, we would apply different process principles, using the following four steps:

2. Divide the detailed design, implementation and test phases into iterations.
3. Baseline an executable architecture early on.
4. Adopt an iterative and risk-driven management process.

Let's examine each of these steps more closely.
Step 1: Build functional prototypes early

As a first step toward iterative development, consider one or more functional prototypes during the requirements and design phases. The objectives of these prototypes are to mitigate key technical risks and clarify stakeholders’ understanding of what the system should do.

Start by identifying the top three technical risks and the top three functional areas in need of clarification. The technical risks might relate to new technology, pending technology decisions that will greatly affect the overall solution, or technical requirements that you know will be hard to meet. Functional risks might relate to areas in which stakeholders have provided fuzzy requirements for critical functionality, or to several requirements that are core to the system.

For each of the key technical risks, outline what prototyping you need to do to mitigate the risks. Consider the following examples:

**Technical risk**: The project requires porting an existing application to run on top of IBM WebSphere Application Server. Users are already complaining about the application's performance, and you are concerned that porting it might slow performance even more.

**Prototype**: Build an architectural prototype to try out different approaches for porting your application. Ask an expert WebSphere architect to help you. Evaluate the results and write architectural and design guidelines providing the team with *dos* and *don'ts*. This will increase the likelihood that your ported application’s performance will be good enough to avoid rework late in the project.

**Technical risk**: You are building a new application for scheduling and estimating software projects. You know that a key differentiator for this application versus off-the-shelf products will be how well it supports iterative planning. However, that is also one of the fuzziest areas in your requirement specification.

**Prototype**: Build a functional prototype based on your assumptions about how to support iterative project planning. By demonstrating the prototype to various stakeholders, you will encourage them to pay more attention to planning and tell you which of your assumptions they agree or disagree with. The prototype will help you clarify the planning requirements and also provide you with useful information about the user experience and look and feel for your application. It might even yield some reusable code.

Step 2: Divide the detailed design, implementation and test phases into iterations.

Many project teams find it hard to divide a project into meaningful iterations before they know what the project is really about. But when you are ready to enter the detailed design phase, you typically have a good understanding of what the requirements are, and what the architecture will look like. It's time to try out iterative development!

You can use two main approaches to determine what should be done in what iteration. Let's discuss the pros and cons of each approach.
Approach 1: Develop one or more subsystems at a time. Let's assume that you have nine subsystems, each with increasingly larger numbers of components. You can divide the detailed design, implementation and test phase into three iterations, each one aiming at implementing three of the nine subsystems. This will work reasonably well if there are limited dependencies among the different subsystems. For example, if your nine subsystems each provided a well-defined set of capabilities to the end user, you could develop the highest priority subsystems in the first iteration, the second most important subsystems in the second iteration, and so on. This approach has a great advantage: If you run out of time, you can still deliver a partial system with the most important capabilities up and running. However, this approach does not work well if you have a layered architecture, with subsystems in the upper layers dependent on the capabilities of subsystems in the lower layers. If you had to build one subsystem at a time, such an architecture would force you to build the bottom layer subsystems first, and then go higher and higher up. But to build the right capabilities in the bottom layers, you typically need to do a fair amount of detailed design and implementation work on the upper layers, because they determine what you need in the lower layers. This creates a "catch-22"; the second approach explains how to resolve it.

Approach 2: Develop the most critical scenarios first. If you use Approach 1, you develop one subsystem at a time. With Approach 2, you focus instead on key scenarios, or key ways of using the system, and then add more of the less essential scenarios. How is this different from Approach 1? Let's look at an example. Suppose you are building a new application that will provide users the ability to manage defects. It is a layered application built on top of WebSphere Application Server, with DB2 as the underlying database. In the first iteration, you develop a set of key scenarios, such as entering a simple defect, with no underlying state engine. In the second iteration, you add complexity to these scenarios -- for example, you might enable the defect to handle a workflow. In the third iteration, you complete the defect entry capability by providing full support for atypical user entries, such as capability to save a partial defect entry and then come back to it, and so forth.

With this approach, you work on all the subsystems in all iterations, but you still focus in the first iteration on what is most important and save what is least important or least difficult for the last iteration.

Approach 1 is more appropriate if you are working on a system with a well-defined architecture -- on an enhancement of an existing application or developing a new application with a simple architecture, for example. Most projects building complex applications should use Approach 2, but they should plan the iterations in such a way that they can cut the scope of the last iterations to make up for possible schedule delays.

Step 3: Baseline an executable architecture early on.

You can view this step as a much more formal and organized way of doing Step 1: Build functional prototypes early on. But what is an "executable architecture"? 
An executable architecture is a partial implementation of the system, built to demonstrate that the architectural design will support the key functionality. Even more important, it demonstrates that the design will meet requirements for performance, throughput, capacity, reliability, scalability, and other "-ilities." Establishing an executable architecture allows you to build all the system's functional capability on a solid foundation during later phases, without fear of breakage. The executable architecture is an evolutionary prototype, intended to retain proven features and those with a high probability of satisfying system requirements when the architecture is mature. In other words, these features will be part of the deliverable system. In contrast to the functional prototype you would typically build in step 1, the evolutionary prototype covers the full breadth of architectural issues.

Producing an evolutionary prototype means that you design, implement, and test a skeleton structure, or architecture, of the system. The functionality at the application level will not be complete, but as most interfaces between the building blocks are implemented, you can (and should) compile and test the architecture to some extent. Conduct initial load and performance tests. This prototype also reflects your critical design decisions, including choices about technologies, main components, and their interfaces; it is built after you have assessed buy versus build options and after you have designed and implemented architectural mechanisms and patterns.

But how do you come up with the architecture for this evolutionary prototype? The key is to focus on the most important 20 to 30 percent of use cases (complete services the system offers to the end users). Here are some ways to determine what use cases are most important.

- **The functionality is the core of the application, or it exercises key interfaces.** The system's key functionality should determine the architecture. Typically an architect identifies the most important use cases by analyzing many factors: redundancy management strategies, resource contention risks, performance risks, data security strategies, and so on. For example, in a point-of-sale (POS) system, Check Out and Pay would be a key use case because it validates the interface to a credit card validation system -- and it is critical from a performance and load perspective.

- **Choose use cases describing functionality that must be delivered.** Delivering an application without its key functionality would be fruitless. For example, an order-entry system would be unacceptable if it did not allow users to enter an order. Typically, domain and subject-matter experts understand the key functionality required from the user perspective (primary behaviors, peak data transaction, critical control transactions, etc.), and they help define critical use cases.

- **Choose use cases describing functionality for an area of the architecture not covered by another critical use case.** To ensure that your team will address all major technical risks, they must understand each area of the system. Even if a certain area of the architecture does not appear to be high risk, it may conceal technical difficulties that can be exposed only by designing, implementing, and testing some of the functionality within that area.

The first and last criteria in the above list will be of greater concern to the architect; project managers will focus mainly on the first two.
For each critical use case, identify the most important scenario(s) and use them to create the executable architecture. In other words, design, implement and test those scenarios.

**Step 4: Adopt an iterative and risk-driven management process.**

If you were to follow Steps 2 and 3 as described above, then you would come very close to the model for "ideal" iterative development. Then, your next step would be to fuse Steps 2 and 3, adding a management lifecycle that supports risk-driven and iterative development. That is the iterative lifecycle described in RUP.

RUP provides a structured approach to iterative development, dividing a project into four phases: Inception, Elaboration, Construction, and Transition. Each phase contains one or more iterations, which focus on producing the technical deliverables necessary to achieve the business objectives of that phase. Teams go through as many iterations as they need to address the objectives of that phase, but no more. If they do not succeed in addressing those objectives within the number of iterations they had planned, they must add another iteration to the phase -- and delay the project. To avoid this, keep your focus sharply on what you need to achieve the business objectives for each phase. For example, focusing too heavily on requirements in Inception would be counterproductive. Below is a brief description of typical phase objectives.

- **Inception**: Establish a good understanding of what system to build by getting a high-level understanding of all the requirements and establishing the scope of the system. Mitigate many of the business risks, produce the business case for building the system, and get buy-in from all stakeholders on whether or not to proceed with the project.
- **Elaboration**: Take care of many of the most technically difficult tasks: design, implement, test, and baseline an executable architecture, including subsystems, their interfaces, key components, and architectural mechanisms (e.g., how to deal with inter-process communication or persistency). Address major technical risks, such as resource contention risks, performance risks, and data security risks, by implementing and validating actual code.
- **Construction**: Do a majority of the implementation as you move from an executable architecture to the first operational version of your system. Deploy several internal and alpha releases to ensure that the system is usable and addresses users' needs. End the phase by deploying a fully functional beta version of the system, including installation, supporting documentation, and training material; keep in mind, however, that the functionality, performance and overall quality of the system will likely require tuning.
- **Transition**: Ensure that the software addresses the needs of its users. This includes testing the product in preparation for release and making minor adjustments based on user feedback. At this point in the lifecycle, user feedback should focus mainly on fine-tuning the product, and on configuration, installation, and usability issues; all the major structural issues should have been worked out earlier in the project lifecycle.

**Many ways to apply these steps**

In this article, we have described how you can gradually transfer from a waterfall approach to an increasingly iterative approach, using four transitional steps. Each step will add tangible value to your development effort, with minimal disruption. Some teams may take on more than one step at
a time; others may run a few projects based on one step and then take the next step. However you choose to use this step-wise approach, it can help you reduce the risks associated with process changes in a development organization.

**Notes**


© Copyright IBM Corporation 2004

Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)