As a complete software development process framework, the RUP has obvious benefits for software engineers in the Analysis, Design, and Implementation disciplines. This article details its benefits for those in the Test discipline.

In many organizations, testing begins late in the software development process, often so late that the product gets released with little testing done. (This has sardonically been referred to as “depending on end-user testing.”) To solve this problem, many companies are beginning to mandate testing as a necessary evil. By contrast, the IBM® Rational Unified Process® changes the testing game by distributing testing throughout the software development process and thereby adding value rather than cost. This article looks at the many benefits of adopting the RUP® that accrue to the testing team.

Why test early and often?

Most testers have seen the chart shown in Figure 1 before -- the later you discover a problem, the more it costs to fix.
So why is testing still so often ignored or pushed until last? The answer is because it seems to take more time, it often doesn't seem to add value early in a software development effort, and it isn't tightly integrated into our software development effort. At the end of the effort, we don't have any time left, so testing gets pushed to our user community.

But we can't expect a quality product without involving our testing/quality teams earlier in the process. Distributing testing throughout the development process by employing the RUP stands to decrease costs and result in a higher quality product at the same time. The RUP encourages testing early by offering a number of mechanisms to integrate testing more closely with the software development effort:

- Making Test a distinct discipline
- Using an iterative development approach
- Continuously verifying quality
- Letting use cases drive development
- Scheduling implementation based on risk
- Managing change strategically
- Using the right-sized process

We discuss each of these mechanisms in the remainder of this article.

**Making Test a distinct discipline**

The RUP organizes activities into similar groups called disciplines. In this model, testing is a discipline in itself. The Test discipline participates in the core development process as both a consumer and a supplier of artifacts and activities -- each of which is clearly defined in the RUP. The instructions on how to do testing in the RUP define the artifacts, templates, tool mentors, guidelines, and checkpoints to use. Figure 2 shows the Test discipline workflow.
Just as important, the interface between other disciplines and testing is clearly defined. The artifacts and activities that other disciplines provide are tailored and built specifically to provide value to testing. There's no need to reformat documents or revalidate their quality, because testing was considered in these artifacts and activities all along.

The RUP defines not just one role specifically for the Test discipline, but four, as shown in Figure 3. Testing staff can undertake from one to all four roles at any one time, but the roles are clearly defined and have defined responsibilities for activities and artifacts. This means that the RUP Test discipline can scale well from a business with one tester to a business with test team members distributed across multiple continents.
Using an iterative development approach

"In the beginning," when software was simpler and teams were smaller, software was developed using a "waterfall" model. First, we'd try to understand everything the users could possibly need a system to do, write it all down, verify it with the users (maybe!), and then move on to the next step -- design. After we'd spent enough time in design and thought we understood how to implement everything, we'd build the system. Once the system was built, we'd ask testing to get involved. It was difficult to move backward, just as it's difficult to go back up a waterfall -- if we couldn't make something work in design, it wasn't a simple matter to go back to requirements and fix it.

The waterfall model is simple to understand, and a project schedule can be created from it relatively easily. The problem is that it just doesn't work well with a large project where some portion of the project may be poorly understood when the design activity begins. Also, with testing not beginning until the system is complete or nearly so, any project schedule slips are usually paid for by reducing the testing cycle (rather than slipping the release date), which often means that very little testing is done.

With the RUP's iterative approach, we scope the project without completely understanding the detailed requirements. We prioritize implementation based on risk, end-user needs, and architectural significance, and then proceed to implement some smaller portion of the system, taking it all the way from requirements through test to deployment. We then do this again with the next piece.

With this approach, the test team gets portions of the system to test a long time before the project is actually complete. For example, a nine-month project might execute in six-week iterations, which
would give six separate releases of partial functionality. Each of these releases gives testers an opportunity to evaluate the quality of the final product.

As shown in Figure 4 (from Walker Royce's book *Software Project Management*), the iterative approach gives the development team the opportunity to make smaller corrections early so the cost of the corrections is smaller. This reduces risk, increases the quality of each deliverable, and allows developers to deliver a working product earlier.

![Figure 4: Profile of the iterative (modern) approach compared with the waterfall approach](image)

**Continuously verifying quality**

Continuously verifying quality is another RUP best practice, and it's achieved by having quality checkpoints for each artifact, activity, and deliverable in the process. The RUP gives clear guidance on how to execute each activity so there's consistency across projects and so all members of the test team understand what their goals are. This gives each team member responsibility for quality, and just as important, it gives team members the tools they need to build quality into their work.

Without clear guidance on what goals the test team members are working toward and without a way of measuring the quality of their work, we get a wide variance in results and team members who don't know what they're trying to do. Just completing a document, for example, isn't enough, because if that document is so poorly written that it fails to communicate to other stakeholders, the document adds little or no value to the project. This frustrates everyone on the project, including the individual who produced the document and especially project management.

The RUP quality checkpoints provide a way for all interested parties on a project to measure the quality of artifacts being created. With these checkpoints, any interested reviewer can participate in evaluating the quality of a deliverable, even without going to a formal meeting. (How often are we given documents to evaluate on our own without having any idea how to measure their worth?) The initial planning for a project makes it easy to include members of the test (and/or quality assurance) team in the review process, and gives all members of the review team the same set
of tools for measuring the quality of a deliverable. Review points can be defined as formally as necessary for the team.

The best practice of continuously verifying quality is an obvious one that every team, every person, every process acknowledges as important. A team using the RUP has the huge benefit of knowing how to measure quality in their work, and also knowing how the consumers of their work will measure its quality as well. The "junction points" between suppliers and consumers are well defined, so no one has to ask "How much should we do here?" or "What do they need that for?"

**Letting use cases drive development**

The RUP is "use-case driven" -- that is, functional requirements are primarily captured in the form of use cases, and design and implementation efforts focus on delivering value as described in these use cases. The deliverables at the end of an iteration are focused on customer-perceived value, and this is what the test team concerns itself with, verifying the use-case model rather than testing internal components. (See Figure 5.)

![Figure 5: How the use-case model drives all phases of the RUP](image)

A use case is a dialog between an actor (someone outside the system) and the system itself, where value is provided to the actor by the system. Use cases don't call out the internal workings of the system unless they can be seen by the actor. Use cases simplify the development team's view of the system to be built; they organize the system requirements in a way that makes them understandable and usable by all members of the team, not merely those who have struggled through implementing them. Furthermore, they make it easy to notice and specify missing requirements and to notice and remove superfluous requirements.

Because the form of a use case (and particularly a scenario in a use case) is similar to that of a test case -- "The user does this . . . the system responds like this . . . the user does this . . . " -- a tester can take the functional requirements as outlined by the requirements team and easily convert them to test cases by adding specific data to a scenario. (*Generating Test Cases From Use Cases* by Jim Heumann describes this process in detail.) The fact that there's no need to translate the system requirements into any other form is an *enormous* benefit to the tester. In addition, as developers execute the use cases, they're in essence executing test cases as they write the software. This is a great benefit for any project team that wants to deliver a quality product.
Scheduling implementation based on risk

Often in software development, the flashiest features are implemented first -- to show progress and ensure continued funding, or because those things can often be the most fun to develop. The problem is that many times this approach leaves high-risk items to be implemented late in the project, which often means major architectural changes and unplanned work at the very end, resulting in a schedule slip. Which area is the first to get cut when a schedule slips? Testing is.

The RUP practice of scheduling implementation based on risk is beneficial to the entire development team, but it's of particular benefit to the testing team. Implementing high-risk items first means schedule changes can happen early in the project, when they're easiest to accommodate. This also means that the major architectural areas have already been tested early in the project, which makes the test cases and test scripts more likely to work later in the project as well. Finally, it means that management isn't faced with a choice between late delivery of a well-tested product or on-schedule delivery of an inadequately tested product.

Managing change strategically

The RUP advocates embracing change in a managed fashion. Late-project unmanaged change, where developers are adding major features right before a release, has been the downfall of many projects. A solid change management strategy improves quality in the project.

A solid change management strategy also gives the test team a way to communicate back to the development team when problems are found. This means fewer disagreements over proper priorities for the development team, and less time wasted expecting something to be fixed when it really hasn't been.

Using the right-sized process

All of this might sound like a lot of work -- a lot of artifacts, activities, and extra work to make things run smoothly. But the RUP isn't about creating unnecessary artifacts and activities; it's about delivering value to your stakeholders. Sometimes, in a very small project, an ad hoc informal process is sufficient to accomplish this. In a very large project, a more formal process is needed to ensure that communication is kept clear and open. The RUP is scalable to any size of project -- from a two-person "we do our own testing" project to a two-thousand-person "the sun never sets on our distributed development" project.

The RUP makes it easy to incorporate just enough process (neither too much nor too little) to make your project successful, as encapsulated in the RUP's "just enough process for your project" philosophy (see Figure 6). There's no "one size fits all" process, so each project does a bit of right-sizing throughout development to make sure the right process elements are in place to mitigate risk and guarantee success on that project.
Some process evangelists might advocate that a lot of process is good for every project every time. The RUP always remembers that the point is to build a product, and the process must add value to the project. If the project doesn't need some aspect of the process, the process should be adapted, not the project.

**In a nutshell**

The RUP is a complete software development process framework that fosters high-quality, on-time, on-budget, well-tested software projects. Adopting and using the RUP makes life easier not only for those who depend on the results of testing but also for members of the test team themselves.

**Resources**

- See *Generating Test Cases From Use Cases* by Jim Heumann (Rational Edge, June 2001) for tips on generating test cases from use cases (what else?).