# Testing: The RUP Philosophy[1]

by **Paul Szymkowiak**
Process Engineer
Rational Software
Canada

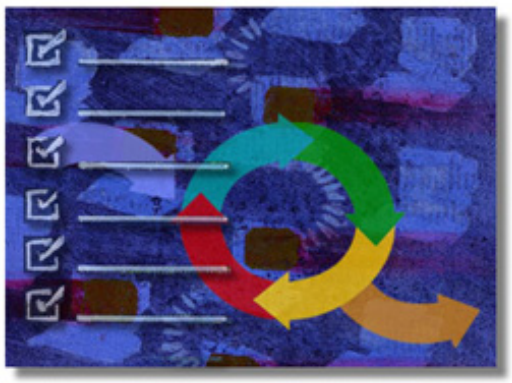
and

**Philippe Kruchten**

Director of Process Development
Rational Software Canada

And what is good, Phaedrus,
And what is not good --
Need we ask anyone to tell us these things?

*--Plato, circa 370 BC*



*Testers and quality engineers sometimes look at the Rational Unified Process,® or RUP,® with a great deal of suspicion. It is true that over the last two years, with the help of many people, both internally (Sam Guckenheimer, most notably) and externally (in particular James Bach, Cem Kaner, and Brian Marick), the RUP approach to testing has taken a bold departure from the traditional approach. It has become more attuned to iterative development, somewhat less focused on high ceremony, and closer to harmonizing with the XP mantra of "test first." In this article, we bring you up to date on our philosophy of testing and provide definitions for major concepts we currently use in RUP to describe this discipline.*

## The Mission of the Tester

The focus of the tester is primarily on *Objective Assessment*. Testers offer their services to other parts of the development organization to help assess the *software product* based on appropriate criteria, such as perceived quality, conformance to standards, and defect discovery. The

evaluations provided by this service help other team players (developers, managers, even customers) to make appropriate decisions about their next actions, based on demonstrable and objective information.

Anyone who is serious about producing an excellent product faces two problems in particular:

- **Effective assessment**: How do you know when the product is "good enough"?
- **Effective communication**: If the product is not yet good enough, how do you ensure that your teammates know it?

The answer to the first question determines when an organization releases a product. The answer to the second question prevents the organization from releasing a poor-quality product.

## The Concept of Product Quality in the RUP

You may be thinking, "I don't want to ship a merely satisfactory product; I want to ship a great product." Let's explore that. What happens when you tell your coworkers, your management, or your investors that your quality standards are high, and that you intend to ship a great product? If it's early in the project lifecycle, they probably nod and smile. Everyone likes quality. However, if it's late in the project lifecycle, you're probably under a lot of pressure to complete the project, and creating a great product may require that you engage in extensive testing, fix many problems (even small ones), add features, or even scrap and rewrite a large part of the code. You will also have to resolve disputes over different visions of "good" quality. Greatness is hard work. Perfection is even harder. Eventually, the people who control the project will come to you and say something like: "Perfection would be nice, but we have to be practical. We're running a business. Quality is good, but not quality at any cost. As you know, all software has bugs." Greatness can be a motivating goal. It appeals to the pride you have in your work. But there are problems with using what amounts to "if quality is good, more quality must be better" to justify the pursuit of excellence. For one thing, to make such an argument can make you seem like a quality fanatic, rather than a balanced thinker. For another thing, it ignores the cost factor. In leaving cost out of the picture, the "more is better" argument also ignores diminishing returns. The better your product, the harder it is to justify further improvement. While you labor to gold plate one aspect of a product, you must ignore other aspects of the product, or the potential opportunities presented by other projects. Every day, the business has to make choices about the best use of resources, and it must consider factors other than quality.

The RUP approach to testing incorporates the concept of *Good Enough Quality* (GEQ) from the work of James Bach.[2] This GEQ concept provides, paradoxically, a more effective argument than "more is better," because it provides a target that is either achievable or not achievable, in which case it becomes a *de facto* argument for canceling or rechartering the project.

## Paradigms of "Good Enough"

Most businesses practice some form of "good enough reasoning" about their products. The only ones that don't are those that believe they have achieved perfection, because they lack the imagination and skill to see how their products might be improved.

Here are some examples of models of the "good enough" approach. Some of them are more effective than others, depending on the situation, but all have their weaknesses:

- **Not Too Bad ("We're not dead yet").** Our quality only has to be good enough so we can continue to stay in business. Make it good enough so that we aren't successfully sued.
- **Positive Infallibility ("Anything we do is good")**. Our organization is the best in the world. Because we're so good, anything we do is automatically good. Think about success. Don't think about failure, because "negative" thinking makes for poor quality.
- **Righteous Exhaustion ("Perfection or bust").** No product is good enough; it's effort that counts. And only our complete exhaustion will be a good enough level of effort. Business issues are not our concern. We will do everything we possibly can to make it perfect. Since we'll never be finished improving, someone will have to come in and pry it from our fingers if they want it. Then they will bear the blame for any quality problems, not us.
- **Customer Is Always Right ("Customers seem to like it").** If customers like it, it must be good enough. Of course, you can't please everybody all the time. And if a current or potential customer doesn't like the product, it's up to them to let us know. We can't read their minds. Quality by market share?
- **Defined Process ("We follow a good process").** Quality is the result of the process we use to build the product. We have defined our process and we think it's a good process. Therefore, as long as we follow the process, a good enough product will inevitably result.
- **Static Requirements ("We satisfy the requirements").** We have defined quality in terms of objective, quantifiable, noncontroversial goals. If we meet those goals, then we have a good enough product, no matter what other subjective, nonquantifiable, controversial goals might be suggested.
- **Accountability ("We fulfill our promises").** Quality is defined by a contract. We promise to do certain things and achieve certain goals. If we fulfill our contract, that is good enough.
- **Advocacy ("We make every reasonable effort").** We advocate for excellence. Throughout the project, we look for ways to prevent problems, and to find and fix the ones we couldn't prevent. If we work faithfully toward excellence, that will be good enough.
- **Dynamic Tradeoff ("We weigh many factors").** With respect to our mission and the situation at hand, a product is good enough when it has sufficient benefits and no critical problems, its benefits

> sufficiently outweigh its noncritical problems, and it would cause more harm than good to continue improving it.

However, for certain types of applications -- safety-critical systems, which may endanger human life -- failure (or more precisely, certain types of failure) is simply not an option.

## The Cost of Quality

Is a high-quality product necessarily more expensive? Depending on a lot of factors, such as process, skill, technology, tools, environment, and culture, you may be able to produce a much higher-quality product for the same cost than would otherwise be possible. A more testable and maintainable product will cost less to improve over the long run. Conversely, there are costs associated with poor quality, such as support costs and costs to the customer. Quality is just like any other part of engineering -- it is a trade-off, an optimization. More is not better at *any* cost. But on the other hand, there are minimum standards.

The cost of quality is a complex issue, and it is difficult to make broad generalizations. The only thing we can say with certainty is that we can always spend more time on much better tests, much more error handling, and fixing or rewriting every part of the product. No matter how good you are, quality does cost something. And if you can't think of more improvements to make, it's more likely that you've reached the upper limit of your imagination, not of quality. There must be a point of "diminishing returns," a point where your "test ROI" becomes negative. In the software industry, GEQ is viewed more as a response to one particular cost over any other -- the cost of not releasing the product soon enough. The specter of a closing market window, or an external deadline, threatens us with tangible penalties if we can't meet the challenge to deliver "on time." That's why project endings are so often characterized by frenzied triage. If you want to know what an organization really believes is good enough, and how well prepared they are for achieving it, witness the last three days of any six-month software project; see what happens when a new problem is reported on the last day.

## Wouldn't Quantification Help?

It can be tempting to reduce quality to a number, and then set a numerical threshold that represents good enough quality. The problem with that is you can only measure factors that *relate* to quality. But you can't measure quality itself. This is partly because the word "quality" is just a label for a relationship between a person and a thing. The statement "This product is high in quality" is just another way of saying,"Somebody values this product." It's a statement not only about the product, but also about people and the surrounding context. Even if the product stays the same, people and situations change, so there can be no single, static, true measure of quality. Read (or reread) Robert Pirsig's *Zen and the Art of Motorcycle Maintenance*, which states, "*At the leading edge there are no subjects, no objects, only the track of Quality ahead, and if you have no formal way of evaluating, no way of acknowledging this Quality, then the train has no way of knowing where to go.*"[3]

There are many measures you might use to get a sense of quality, even if you can't measure it completely and objectively. Even so, the question of what quality is good enough requires sophisticated judgment. You can't escape from the fact that, in the end, people have to think it through and make a judgment. For a simple product, that judgment may be easy. For a complex, "high-stakes" product, it's very hard. Making this call is not the responsibility of the tester, but the tester does provide much of the information that is used to make this decision. Finally, the definition of quality also depends upon the problem space; quality for NASA or for surgical automation has a very different meaning from quality for a customer service rep's workstation, and a different meaning from quality for a stock exchange.

## Conformance to Standards

Assessment implies the comparison of the product to some standard. One standard of reference that often comes to mind is the requirements -- the "specs" for the system. There may be other standards referred to, either formal standards, such as a usability or accessibility standard, or implicit standards, such as those related to look and feel (for example, we want it to match Windows 2000 conventions). There are also standards that are industry-specific: standards for the biomedical industry, the pharmaceutical industry, the aeronautical industry, and so on.

In practice, assessment is not only about comparing the software product against the requirements and other specification artifacts. We use -- implicitly or explicitly -- other standards for determining whether quality is acceptable. There are two important concerns that often require a broader consideration than the documented specifications:

- **Are the specifications themselves complete?** By their nature, specifications are somewhat abstract and evolve over the life of the project as a greater understanding of the problem and appropriate solutions are gained. Are there additional things we should assess that may be somewhat ambiguous or missing from the specifications?
- **What exceptional events or conditions might cause the software to break?** Specifications often overlook a number of exceptional concerns that are arguably more apparent to a tester. Sometimes that is because the requirements specifier-- the analyst -- is not familiar with certain aspects of the problem or solution domain, or because the exceptional requirements are regarded as implicit.

A diligent tester will consider tests that help to expose issues that arise from implied and omitted requirements.

The Quality Assurance Plan, which influences the Software Development Plan and, under the project manager's responsibility, defines the overarching approach to GEQ on the project and the techniques that will be used to assess whether an acceptable level of quality is being achieved.

## What Is Testing?

The Test discipline of the RUP acts in many respects as a service provider to the other disciplines. **Testing** focuses primarily on the evaluation or assessment of quality and is realized through a number of core practices:

- Finding and documenting gaps in GEQ.
- Generally advising team members about perceived software quality.
- Validating through concrete demonstration the assumptions made in design and requirement specifications.
- Validating that the software product functions as it was designed to.
- Validating that the requirements have been implemented appropriately.

An interesting, but somewhat subtle, difference between the Test Discipline and the other disciplines in RUP is that testing is essentially tasked with finding and exposing weaknesses in the software product. For this effort to be successful, it requires a somewhat negative approach rather than a positive one. It is driven by key questions such as: "How could this software fail?" The challenge is to find balance between two extremes: on one hand, avoiding both the approach that does not suitably and effectively challenge the software and expose its inherent problems and weaknesses, and, on the other hand, avoiding an approach that is so negative that it is unlikely to ever find the quality of the software product acceptable.

Based on information presented in various surveys and essays, software testing is said to account for 30 to 50 percent of total software development costs. It is, therefore, perhaps surprising to note that most people believe computer software is not well tested before it is delivered. This contradiction is rooted in a few key issues.

- Testing is usually done late in the lifecycle, keeping project risks and the number of unknown factors very high, for far too long, rather than assessment through testing with every iteration, as the RUP advocates.
- Testability is not considered in the product design (again contrary to the RUP), which thereby increases the complexity and effort required to perform tests, making test automation difficult, and in some cases making certain types of test impossible.
- Extensive test planning is done up front, in isolation from the system under test (SUT), when the least is known about the system under test. In contrast, the RUP advocates detailed test planning iteration by iteration, using the experience gained from actual testing in the previous iteration.

Beyond these issues, we also have to acknowledge that testing software is enormously challenging. The different ways a given program can behave are unquantifiable, and the number of potential tests for that program is

arguably limited only by the imagination of the tester.

Often, testing is done without a guiding methodology, resulting in a wide variance of success from project to project and organization to organization; success is primarily a factor of the quality, skills, and experience of the individual tester. Testing also suffers when insufficient use is made of productivity tools, to make the laborious aspects of testing manageable. A lot of testing is conducted without tools that allow the effective management of test assets such as extensive Test Data, without tools to evaluate detailed Test Results, and without appropriate support for automated test execution. While the flexibility of use and complexity of software make "complete" testing an impossible goal in all but the most trivial systems, an appropriately chosen methodology and the use of proper supporting tools can improve the productivity and effectiveness of the software testing effort.

These are generic testing concerns, but certain aspects of software testing are more context-specific. For "safety-critical" systems where a failure can harm people (such as air-traffic control, missile guidance, or medical delivery systems), high-quality software is essential for the success of the system. For a typical MIS system, the criticality of the system may not be as immediately obvious as in a safety-critical system, but it's likely that a serious defect could cost the business using the software considerable expense in lost revenue or possible legal costs. In this "information age" of increasing demand on the provision of electronically delivered services over media such as the Internet, many MIS systems are now considered "mission-critical"-- that is, when software failures occur in these systems, companies cannot fulfill their functions and experience massive financial losses.

Another specific testing concern is projects that do not pay enough attention to performance testing until very late in the development cycle. For systems that will be in continuous use (24x7), for distributed systems, and for systems that must scale up to a large number of simultaneous users, it is important to assess early and continuously to verify that the expected performance will be met. This can start in the Elaboration phase when enough of the architecture is in place to start exercising the system under various load conditions.

A continuous approach to quality, initiated early in the software lifecycle, can significantly lower the cost of completing and maintaining the software. This greatly reduces the risk associated with deploying poor-quality software.

## The RUP Testing Philosophy

In a traditional, waterfall approach, up to 80 percent of the test project time can be spent planning the test effort and defining test cases, but not actually conducting any testing at all. Then towards the end of the lifecycle, 20 percent of the effort is typically spent running and debugging tests. Often an additional 20 percent (yes, we are now over-budget!) is then required to fix anything in the product that did not pass the tests.

The test philosophy in the RUP takes a different approach and can be summarized as a small set of principles:

- **Iterative development**. Testing does not start with just test plans. The tests themselves are developed early and conducted early, and a useful subset of them is accumulated in regression suites, iteration after iteration. This enables early feedback of important information to the rest of the development team, permits the tests themselves to mature as the problem and solution spaces are better understood, and enables the inclusion in the evolving software design of required testability mechanisms. To account for the changing tactical objectives of the tester throughout the iterative lifecycle, we will introduce the concept of **mission**.
- **Low up-front documentation**. Detailed Test planning is defined iteration by iteration, based on a governing master Test plan, to meet the needs of the team and match the objectives of the iteration. For example, during the Elaboration phase, we focus on architecture, and so the test effort should focus on testing the key architectural elements as they evolve in each iteration. The performance of key end-to-end scenarios should be assessed -- typically under load -- even though the user interface may be rudimentary. But beyond this semiformal artifact that defines the test plan for each iteration, there is not a lot of upfront specification paperwork developed.
- **Holistic approach**. The approach to identifying appropriate tests is not strictly and solely based on deriving tests from requirements. After all, the requirements rarely **specify** what the system should *not* do; they do not enumerate all the possible crashes and sources of errors. The tests for these issues have to come from elsewhere. Tests in the RUP are derived from the requirements *and* from other sources.
- **Automation**. Testing starts early in, and continues throughout, the RUP lifecycle; it is impractical to work on many aspects of testing without appropriate tool support. In particular, tools can help **generate** test data conditions, run tests, and analyze results.

## Mission

The concept of an evaluation **mission**, as used in the RUP approach to testing, has been derived from the work of James Bach in identifying different missions commonly adopted by software test teams. Bach advocates using a simple heuristic model for test planning. This model recognizes that different missions govern the activities and deliverables of the testing effort, and that selecting an appropriate mission is a key aspect of test planning.

The evaluation mission identifies a simple statement the test team can remember in order to stay focused on their overall goal and appropriate deliverables for a given iteration. This is especially important in situations where the team is faced with a number of possibly conflicting goals. A test team without an evaluation mission often describes their goal with

statements such as "We test everything" or "We just do testing." Such teams are concerned with simply performing the test activities and overlook how those activities should be adjusted to suit the current project context or iteration context to achieve an appropriate goal.

Mission statements shouldn't be too complex or incorporate too many conflicting goals. The best mission statements are simple, short, succinct -- and achievable. Here are some ideas for mission statements you might adopt for a given iteration:

- Find as many defects as possible.
- Find important problems fast.
- Assess perceived quality risks.
- Advise about perceived project risks.
- Advise about perceived quality.
- Certify to a given standard.
- Assess conformance to a specification (requirements, design, or product claims).

## Test Cycles

A **test cycle** is a period of independent test activity that includes, among other things, the execution and evaluation of tests. Each iteration can contain multiple test cycles -- the majority of iterations contain at least one. Each test cycle starts with the assessment of a software build's stability before it's accepted by the test team for more thorough and detailed testing.

The RUP recommends that each build be regarded as potentially requiring a new cycle of testing (that is, a test cycle), but there is no strong coupling between build and test cycle. Typically, Inception iterations of new projects don't produce builds, but iterations in all other phases do. Although each build is a potential candidate for a cycle of testing, there are various reasons why you might not decide to test every software build. For example, in situations where a build is created daily, a useful test cycle may take longer than the available time between builds. In this case it may be appropriate to test every nth software build.

## The Test Discipline in the RUP

In the RUP, you'll find the implementation of this approach to testing. In particular, you'll find details on the following artifacts:

- Test Evaluation Summary
- Test Plan (and sometimes a Master Test Plan)
- Test Ideas, and Test-Idea List
- Test Suites and Test Cases

- Defect and Defect List
- Workload Model

The RUP has four roles focusing on test-related activities:

- Test Manager
- Test Analyst
- Test Designer
- Tester

These are roles (not necessarily job titles) that a software developer may be called upon to play.

## Conclusion

Testing in the RUP is about continuously providing the management and the development teams with an objective assessment of the quality of the product. It is not about ticking all the checkmarks for all the *requirements* (even if this plays a role) in one massive shot toward the end of the project lifecycle.

The expected level of quality and the requirements may evolve during the lifecycle, and testing should evolve with them. The definition of the level of quality is an engineering trade-off -- an optimization -- highly dependent on the business context in which the software product will be deployed. Testing can start early, since iterative development produces testable code early, and can therefore provide a crucial feedback, both on the product and the process, so that they can evolve as required - feedback that is sorely missing in more traditional approaches.

The role of the tester or the quality engineer in the RUP is not primarily to find defects, but to provide other team members -- developers and managers -- with an objective assessment of the quality of the product. Testing is not a complete, separate workflow, parallel to or appended to software development, but fully integrated in the iterative cycle of the RUP. Taking advantage of iterations, testing becomes an almost-continuous process that continually feeds back into the requirements, the design, and the management of the project. It is then a collaborative activity, no longer an adversarial or rubber-stamping activity. It is also more effectively automated by tools. Testing in the RUP embraces a practical and flexible notion of Good Enough Quality that takes into account the balance between the cost of testing and the desired level of quality based on the context.

## Acknowledgments

Many thanks to the usual set of culprits for making our prose and style more focused, and for setting us straight: Sam Guckenheimer, Vincent Encontre, Joe Marasco, Lee Thomas, and Catherine Southwood.

## References

James Bach, "Good Enough Quality: Beyond the Buzzword." *IEEE Computer*, 30 (8), August 1997, pp. 96-98.

Rex Black, *Managing the Testing Process*. Microsoft Press,1999.

Cem Kaner, Jack Falk, and Hung Quoc Nguyen, *Testing Computer Software*, 2nd Edition. John Wiley & Sons,1999.

Cem Kaner, James Bach, and Bret Pettichord, *Lessons Learned in Software Testing*. John Wiley & Sons, 2001.

Jim Heumann, "Generating Test Cases from Use Cases." *The Rational Edge*, June 2001.
http://www.therationaledge.com/content/jun_01/m_cases_jh.html

Brian Marick, "Faults of Omission." *Software Testing and Quality Engineering Magazine*, March-April 2000.

Glenford J. Myers, *The Art of Software Testing*. John Wiley & Sons, 1979.

Robert Pirsig, *Zen and the Art of Motorcycle Maintenance*. Bantam Books,1977. Connie U. Smith and Lloyd Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2001.

---

## Notes

[1] This article is part of a chapter on testing in Per Kroll and Philippe Kruchten's upcoming book: *RUP Made Easy*, to be published this spring by Addison Wesley Longman (ISBN 0-32-116609-4).

[2] See James Bach, "Good Enough Quality: Beyond the Buzzword," *IEEE Computer*, 30 (8), August 1997, pp.96-98.

[3] Robert Pirsig, *Zen and the Art of Motorcycle Maintenance*. Bantam Books, 1977, p. 277.

---

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