



# Myths and realities of iterative testing

Level: Introductory

Laura Rose, Quality Engineering Manager, IBM

15 Apr 2006

from The Rational Edge: Software testing expert Laura Rose questions and debunks some widely held myths pertaining to iterative development in general and iterative testing in particular. She explains how iterative development principles can address these common misunderstandings and lead to a pragmatic testing methodology.

*Myths arise from a lack of direct experience. In the absence of information, we form beliefs based on what we think we know, often with a skeptical feeling towards what we don't know. In the realm of software development, myths can make it difficult to approach real-world problems objectively, thus putting budgets and schedules at increased risk.*

*In my career as a quality assurance manager, I have gained experience with a number of software development practices, including iterative development and the so-called "waterfall" approach. The former is generally presumed to be a more modern method than the latter. But this is, in fact, a myth: both approaches have been around since the 1960s. Another myth led to the popularity of the waterfall method in the 1970s. The thinking of Winston Royce, often cited as the father of the waterfall method, was actually misinterpreted. He recommended the single-pass waterfall method only for legacy maintenance projects. Royce actually suggested an iterative "do it twice" approach for software applications being developed for the first time.*



*From these inauspicious beginnings, the world of software development has given rise to numerous other myths. This article will question and debunk some of the most widely held myths pertaining to iterative development in general and iterative testing in particular. I'll explain how iterative development principles can address these common misunderstandings and lead us to a pragmatic testing methodology that mitigates or avoids altogether many common software development pitfalls, some of which our mythology holds as inevitable.*

**Myth:** In a majority of software development projects, we quickly code a prototype application to reduce risk and prove a concept, with the intention of throwing this code away.

**Reality:** There is nothing wrong with this approach. But, because of schedule pressure or because the results are encouraging, we do not discard that code. The reality is that our *prototyping* is actually *early coding*. That code becomes the foundation and framework for our new application. However, because it was created under the presumption that it would be thrown away, it bypassed requirements reviews, design reviews, code reviews, and unit testing. Our next-generation application is now built on an indeterminate foundation.

In an iterative development lifecycle, continuous experimentation is viewed as a good thing, and early prototyping is encouraged in each development iteration. But any code that is submitted into the product needs to follow best practices to assure its stability, reliability, and maintainability. One way to encourage proper software development practices from the very start of a project is to use the initial coding effort to plan, prove, and present the processes you intend to use throughout the various iterative development phases. As part of the iterative testing method, you can use your early coding cycles to test your product's concept and flush out glitches in your development processes at the same time.

**Myth:** Initiating testing activities earlier in the development cycle increases delivery time while reducing the number of features in the product.

**Reality:** Testing is *not* the time-consuming activity in a development lifecycle. Diagnosing and fixing defects are the time-consuming, bottleneck activities.

Testing is not the obstacle that shipwrecks our projects -- testing is the lighthouse that keeps our projects off the rocks.

Defects are in the product whether we look for them or not. Iterative testing moves the detection of problems closer to where and when they were created. This minimizes the cost of correcting the bug as well as its impact on schedules.

**Myth:** You can't test if you don't have a product to test.

**Reality:** Iterative testing isn't limited to testing code.

Every artifact your team produces can be verified against your success criteria for acceptance. Likewise, each process or procedure you use to produce your deliverables can be validated against your success criteria for quality. This includes the product concepts, the architecture, the development framework, the design, the customer usage flows, the requirements, the test plans, the deployment structure, the support and service techniques, the diagnostic and troubleshooting methods, and even the procedures you follow to produce the product.

As you can see from this partial list, the majority of these items don't involve code. Therefore, you miss opportunities to support quality and reduce risk by waiting until you actually have deliverable code. I disagree with the widely accepted notion that "you can't test quality into a product." You *can* test quality into a product. You just need to start early enough.

**Myth:** You are more efficient if you have one developer (or a single source) specialized in each development area. In the simplest argument, if you have thirty developers and employ pair programming, you can code fifteen features. If you assign one developer per feature, you can code thirty features in the same amount of time. With thirty features, you have a fuller and more complete product.

**Reality:** The risk associated with having one developer per feature or component is that no one else can maintain and understand that feature. This strategy creates bottlenecks and delays. Defects, enhancement requests, or modifications are now queued to that single resource. To stay on schedule, your developers must work weekends and extra hours for extended periods, because they are the only ones who can continue new feature development and fix defects in this area. Your entire project schedule is now dependent on heroic efforts by a number of "single resources."<sup>1</sup> When that resource leaves the team, goes on vacation, or becomes overwhelmed, it causes delays in your schedule. Because of the way you've chosen to implement and manage your development strategy, you are now unable to provide your team with relief.

Pair programming, pair testing, code reviews, and design reviews are sound practices that not only increase the quality of the product, but also educate others on each feature or component, such that they increase your pool of resources to fix and maintain project code. The two members of a pair don't have to be equally sophisticated or knowledgeable in their area. They can be just knowledgeable enough to eliminate the inevitable bottlenecks created by specialization as discussed above.

Moreover, dividing development activities into logical, smaller, independent chunks (a.k.a. sprints) allows developers of different skill levels to work efficiently on the various pieces. With multiple capable resources, you can avoid actually assigning the different tasks to a specific developer. When we have more than one resource capable of accomplishing the task, assigning specific tasks to specific individuals creates a false dependency. Similar to multiple bank tellers servicing a single line of waiting customers, efficiency improves when developers instead sign up for the next task when they've completed the last task.

**Myth:** Producing code is the developer's primary task.

**Reality:** The primary task of everyone on the development team is to produce a product that customers will value. This means that during the requirements review activities, for example, the developer's primary task is "requirement review." During the design activities, the developer's primary task is creating and reviewing the design documents. During the coding activities, the developer's primary task is generating bug-free and customer-relevant code. During the documentation review activities, the developer's primary task is making sure the user assistance materials and error messages serve to flatten the customer's learning curve. During installation and setup, the developer's primary task is to make sure customers can set up and configure your product easily, so that they can get their "real" job done as efficiently as possible. The greater the effort required to use software to accomplish a task, the lower the customer's return on investment and the greater the abandonment rate for the application.

**Myth:** Requirements need to be stable and well defined early for software development and testing to take place efficiently.

**Reality:** If our ultimate goal was "efficient software development and testing," then having stable and well-defined requirements up-front might be a must. But our actual goal is to produce a product that customers will value. Most of us can readily admit that we don't always know what we want. Since there are constant changes in the marketplace, such as new products and options, we often change our minds after being introduced to new concepts and information. Acknowledging the above illustrates one key reason why it is rarely effective to keep requirements static. Frequent interaction with the product

during development continually exposes the customer to our efforts and allows us to modify our course to better meet customers' changing needs and values.

**Myth:** When the coding takes longer than originally scheduled, reducing test time can help get the project back on schedule.

**Reality:** Typically, coding delays occur because of unexpected difficulties or because the original design just didn't work. When it's apparent that we've underestimated project complexity, slashing test time is a very poor decision. Instead, we need to acknowledge that, since the test effort was probably based on the underestimated coding effort, the test effort was probably underestimated as well. We may therefore need to schedule more test time to correct the original estimation, not less.

Iterative testing increases test time without affecting the overall schedule by starting the testing earlier in each iteration. Also, the quality of the product determines the amount of testing that is required -- not the clock. For instance, if the product is solid and no new defects are being found in various areas, then the testing will go very smoothly and you can reduce testing time without reducing the number of tests or the test coverage. If the product is unstable, and many defects are being discovered and investigated, you'll need to add test cycles until the quality criteria are met. And don't forget that testing is not the project's time-consuming activity in the first place.

**Myth:** Finding and fixing all the defects will create a quality product.

**Reality:** Recent studies illustrate that only 10 percent of software development activities, such as creating customer-requested features, actually add value for the customer. A percentage of features, for example, are developed in order to stay competitive in the market, but are not necessarily used or desired by the customers. Finding and fixing defects related to these features also does not add customer value, because the customers might never have encountered these bugs.

Iterative testing, on the other hand, actually reduces defect inventory and customer wait time based upon what is of value to the customer. By involving customers in each iteration, iterative testing compresses the delivery cycle to the design partner customers, while maximizing the value of the application to this customer.

**Myth:** Continually regression-testing everything every time we change code is tedious and time consuming...but, in an ideal world, it should be done.

**Reality:** Regression testing doesn't mean "testing everything, every time."

Iterative regression testing means testing what makes sense in each phase and iteration. It also means modifying our coverage based on the impact of the change, the history of the product, and the previous test results.

If your regression tests are automated, then go ahead and run all of them all the time. If not, then be selective about what tests you run based on what you want the testing to accomplish. For instance, you might run a "sanity regression suite" or "set of acceptance tests" prior to "accepting" the product to the next phase of testing. Since the focus of each iteration is not necessarily the same, the tests don't need to be the same each time. Focus on features and tests that make sense for the upcoming deliverables and phase. For instance, if you're adopting components from a third party, like a contractor or an open source product, the sanity regression suites would focus on the integration points between the external and internal components. If, during your initial sanity regression testing of this third party module, you find defects or regressions, you may choose to alter your regression suites by adding additional tests based on the early results.

If, on the other hand, you're adopting a set of defect fixes that span the entire product that is within your control, the sanity regression suite would be focused and structured entirely on your end-to-end, high profile customer use cases. If the change is confined to just one area and the product has a stable quality track record, you could focus the regression suite on just that area. Likewise, in the end-game, you may want a very small sanity regression suite that covers media install, but not in-depth or end-to-end tests. Once again, the focus of the sanity or acceptance regression suite depends upon what was tested in the previous cycle, the general stability of the product, and the focus of the next iteration.

**Myth:** It's not a bug -- the feature is working as designed.

**Reality:** The over-explanation of why the product is doing what it's doing is a common trap. Sometimes we just know too much. When defects are triaged and reviewed, we often explain away the reasons for the defect. Sometimes we tag defects as "works as designed" or "no plans to fix" because the application is actually working as it was designed, and it would be too costly or risky to make the design change. Similarly, we explain many usability concerns as "it's an external component" or "it's a bell and whistle." Our widgets or UI controls may have known limitations. Or we may even tell ourselves that "once the user learns to do it this way, they'll be fine."

**In short:** We know the ins and outs of the code, and why it's working as it is. And at that component level, we're making very

logical and sensible decisions. But we're lacking the top-down view of the entire customer experience. We aren't appreciating the end effect of our coding trade-offs and workarounds. Our primary focus is getting the feature code completed on time. By focusing on just the individual feature or component, we are unintentionally making code decisions that negatively impact the overall flow and usability of the product. It's usability, after all, that most affects the customer's efficiency. It's also usability (or lack thereof) that drives user abandonment numbers.

Elevating our view of the project above those layers of detail allows us to see the product as the customer would -- in its entirety, rather than as individual components. This elevated view helps us to ignore all the reasons why the product works the way it does. Customers don't really care if a given design was quicker to code. It's of little concern to them if the UI controls are conflicting with the API of component X, which is outside your particular development task. They are only concerned about whether the application is assisting or obstructing them in the pursuit of their goals.

Table 1 illustrates some ways we can act as advocates for the customer's perspective as we test an application.

**Table 1: Ways to advocate for the customer's perspective**

When we encounter this response....	... we should ...
We already know this isn't right.	Identify explicit things that need to be fixed by the release date and what needs to be fixed in a subsequent release. Identify owners and deadlines.
It's on our list for future consideration (with no scheduled date).	Realize that if it has a "future consideration" tag (or similar), it's not real. Work with tech support, field consultants, and customers to illustrate the importance of the bug or feature. Once the team is assured of the customer value, schedule a release date and owner.
It's in someone else's code outside our department.	Create a SWAT team that includes "outside department" staff in addition to development staff. Schedule a release date and owner of the change.
Customer or pilot error.	Study the documentation or usage flow. Customer errors are often the result of some unclear and unintuitive step. Explicitly identify the change that needs to occur, schedule it, and set an owner.
It's working as designed.	Request a full review of the design in the workflow perspective. Walk through how the customer or role will play out from beginning to end.
It's working this way because XXX.	Realize that if XXX isn't "because the customer wants it this way," then XXX is irrelevant. Eliminate that reason from the discussion and move on to the next point.

**Myth:** A tester's only task is to find bugs.

**Reality:** This view of the tester's role is very limited and adds no value for the customer. Testers are experts with the system, application, or product under test. Unlike the developers, who are responsible for a specific function or component, the tester understands how the system works as a whole to accomplish customer goals. Testers understand the value added by the product, the impact of the environment on the product's efficiency, and the best ways to get the most out of the product.

Taking advantage of this product knowledge expands the value and role of our testers. Expanding the role of the tester to include customer-valued collateral (like tips, techniques, guidelines, and best practices for use) ultimately reduces the customer's cost of ownership and increases the tester's value to the business.

**Myth:** We don't have enough resources or time to fully test the product.

**Reality:** You don't need to fully test the product -- you need to test the product sufficiently to reduce the risk that a customer will be negatively affected.

The reality of changing market demands generally means that, indeed, it's actually not possible to exhaustively test a product in the specified timeframe. This is why we need a pragmatic approach to testing. Focus on your customers' business processes to identify your testing priorities. Incorporate internal customers to system test your product. These steps increase your testing resources, while providing real-world usability feedback. You can also do your system testing at an external customer lab to boost your real-world environment experience without increasing your maintenance or system administration activities.

**Myth:** Testing should take place in a controlled environment.

**Reality:** The more the test environment resembles the final production environment, the more reliable the testing. If the customer's environment is very controlled, then you can do all your testing in a controlled environment. But if the final production environment is not controlled, then conducting 100 percent of your testing in a controlled environment will cause you to miss some important test cases.

While unpredictable events and heterogeneous environments are difficult to emulate, they are extremely common and therefore expected. In our current global market, it is very likely that your application will be used in flexible, distributed, and diverse situations. In iterative testing, we therefore schedule both business usage model reviews and system testing activities with customers whose environments differ. The early business usage reviews identify the diversity of the target customer market, prior to coding. System testing at customer sites exercises our product in the real world. Although these "pre-released" versions of the product are still in the hands of our developers and running on our workstations, they are tested against customer real-world office (or lab) environments and applications. While this strategy doesn't cover every contingency, it acknowledges the existence of the unexpected.

**Myth:** All customers are equally important.

**Reality:** Some customers are more equal than others, depending upon the goal of a particular release. For example, if the release-defining feature for the January release is the feature that converts legacy MyWidget data to MyPalmPilot data, then the reactions of my customers that use MyWidget and MyPalmPilot are more important for this particular release than the input of other customers.

All our customers are important, of course. But the goal of iterative testing is to focus on testing the most important features for this particular iteration. If we're delivering feature XYZ in this iteration, we want expert customer evaluation of XYZ from users familiar with prior XYZ functionality. While we welcome other feedback, such as the impressions of new users, the XYZ feature takes precedence. At this stage of development, users new to the market cannot help us design the "right XYZ feature."

**Myth:** If we're finding a lot of bugs, we are doing important testing.

**Reality:** The only thing that finding a lot of bugs tells us is that the product has a lot of bugs. It doesn't tell us about the quality of the test coverage, the severity of the bugs, or the frequency with which customers will actually hit them. It also doesn't tell us how many bugs are left.

The only certain way to stop finding bugs is to stop testing. It seems ridiculous, but the thought has merit. The crux of this dilemma is to figure out what features in the product actually need to work. I've already mentioned that there are many workflows in a product that aren't actually used -- and if they aren't used, they don't need to work. Incorporating customer usage knowledge directly into your test planning and defect triage mechanism improves your ability to predict the customer impact and risk probability associated with a defect. Incorporating both risk- and customer-based analysis into your test plan solution will yield a more practical and pragmatic test plan. Once you're confident in your test plan, you can stop testing after you've executed the plan.

How do you build that kind of confidence? Start, in your test planning, by identifying all the areas that you need to test. Get customer review and evaluation on business processes and use cases so that you understand the frequency and importance of each proposed test case. Take special care to review for test holes. Continually update and review your test plans and test cases for each iteration. Your goal is to find what's not covered. One way to do this is to map bug counts by software areas and categories of tests. If a software area doesn't have defects logged against it, it could mean that this area is extremely solid or that it hasn't been tested. Look at the timestamps of the defect files. If the last defect was reported last year, maybe it hasn't been tested in awhile. Finding patterns of missing bugs is an important technique to verifying test coverage.

**Myth:** Thorough testing means testing 100 percent of the requirements.

**Reality:** Testing the requirements is important, but not sufficient. You also need to test for what's missing. What important requirements aren't listed?

Finding what's not there is an interesting challenge. How does one see "nothing?" Iterative testing gets customers involved early. Customers understand how their businesses work and how they do their jobs. They can tell you what's missing in your application and what obstacles it presents that stops them from completing their tasks.

**Myth:** It's an intermittent bug.

**Reality:** There are no intermittent bugs. The problem is consistent -- you just haven't figured out the right conditions to reproduce it. Providing serviceability tools that continually monitor performance, auto-calibrate at the application's degradation thresholds, and automatically send the proper data at the time of the degradation (prior to the application actually crashing) reduces both in-house troubleshooting time and customer downtime. Both iterative testing and iterative serviceability activities reduce the business impact of undiscovered bugs.

Better diagnostic and serviceability routines increase the customer value of your product. By proactively monitoring the

environment when your product starts to degrade, you can reduce analysis time and even avoid shutdown by initiating various auto-correcting calibration and workaround routines. These types of autonomic service routines increase your product's reliability, endurance, and run-time duration, even if the conditions for reproduction of a bug are unknown.

In a sense, autonomic recovery routines provide a level of continuous technical support. Environment logs and transaction trace information are automatically collected and sent back to development for further defect causal analysis, while at the same time providing important data on how your product is actually being used.

If we acknowledge that bugs are inevitable, we also need to realize the importance of appropriate serviceability routines. These self-diagnostic and self-monitoring functions are effective in increasing customer value and satisfaction because they reduce the risk that the customer is negatively affected by bugs. Yet even though these routines increase customer value, few development cycles are devoted to putting these processes in place.

**Myth:** Products should be tested under stress for performance, scalability, and endurance.

**Reality:** The above is true. But so is its opposite. Leaving an application dormant, idle, or in suspend mode for a long period emulates customers going to lunch or leaving the application suspended over the weekend, and often uncovers some issues.

I recommend including sleep, pause, suspension, interrupt, and hibernating mode recovery in your functional testing methods. Emulate a geographically-distributed work environment in which shared artifacts and databases are changing (such as when colleagues at a remote site work during others' off-hours) while your application is in suspend or pause mode. Test what occurs when the user "wakes it up," and the environment is different from when they suspended it. Better yet, put your product in a real customer environment and perform system testing that emulates some of the above scenarios.

**Myth:** The customer is always right.

**Reality:** Maybe it's not the right customer. You can't make everyone happy with one release. Therefore, be selective in your release-defining feature set. Target one type of customer with a specific, high-profile testing scenario. Then, for the next release or iteration, select a different demographic. You'll test more effectively; and, as your product matures, you'll increase your customer base by adding satisfied customers in phases.

**Myth:** Automate! Automate! Automate!

**Reality:** Automate judiciously and with ROI in mind. The more the test environment resembles the final production environment, the more reliable the testing. If the customer's product is 100 percent automated, then Automate! Automate! Automate! If your product isn't meant to be automated in the customer's environment, then you want a combination of automation, ad hoc, exploratory, and customer scenario testing.

It's also helpful to expand your definition of automation. Automated test cases can retest areas that you have already tested manually. But that doesn't increase your test coverage or your understanding of how users interact with the application. Instead, create automation that actually allows you to invest more effort in creative manual testing. Automate the things you need to do frequently and that take you a long time to accomplish, like:

- The breakdown and setup of clean environments for testing each build.
- Sanity acceptance testing for every build, integration point, or iteration.
- Testing the same suite across various platforms, operating systems, and languages.
- Unit and command line testing of low-level components.

For more ideas on increasing your return on investment in automation, visit <http://www-128.ibm.com/developerworks/rational/library/may05/rose/>.

**Myth:** Iterative development doesn't work.

**Reality:** It's human nature to be skeptical of the unknown and the untried. In the iterative development experience, benefits accrue gradually with each successive phase. Therefore, the full benefit of the iterative approach is appreciated only towards the end of the development cycle. For first-timers, that delayed gratification can truly be an act of faith. We have no experience that the approach will work, and so we don't quite trust that it will. When we perceive that time is running out, we lose faith and abandon the iterative process. In panic mode, we fall back into our old habits. More often than not, iterative development didn't actually fail. We just didn't give it a chance to succeed.

Iterative testing provides visible signs at each iteration of the cumulative benefits of iterative development. When teams share incremental success criteria (e.g., entrance and exit criteria for each iteration), it's easier to stay on track.

Because we are continually monitoring our results against our exit criteria, we can more easily adjust our testing during the iterations to help us meet our end targets. For instance, in mid-iteration, we might observe that our critical and high defect counts are on the rise and that we're finding bugs faster than we can fix them. Because we've recognized this trend early, we can redistribute our resources to sharpen our focus on critical-path activities. We might reassign developers working on "nice-to-have" features to fix defects in "release-defining" features or remove nice-to-have features associated with high defect counts.

## Conclusion

I've touched on just a few of the software development myths and assumptions that we encounter every day. The more assumptions we make, the less we're open to discovering the unexpected -- which is what software testing is all about.

Unfortunately, our myths are very seductive. They are disguised as answers, and they conveniently end the dialogue. When we're understaffed and under pressure, it's very tempting to accept assumptions as facts.

Because it's often difficult for us to distinguish myth from reality, we need to test our answers. This simple and effective mantra, which sums up everything I've talked about in this article, will help keep you on that path:

*Iterative testing is never being satisfied with the right answer.*

## Notes


<sup>1</sup> In the Capability Maturity Model (CMM, or CMMI), the maturity level described by depending upon heroic efforts by several individuals is symptomatic of the Level 1, better known as the Chaos Level.

## About the author



Laura Rose is a quality assurance expert and the product manager responsible for automated performance test tools at IBM Rational. In addition to leading projects in both software programming and testing environments, she has thirteen years of programming experience and ten in test management. She has been a member of the American Society for Quality, the Triangle Quality Council, and the Triangle Information Systems Quality Association. She is published and has presented at various test and quality conferences including IBM Test Symposium West, Practical Software Quality Conference (East and West), the American Quality Society Conference, Better Software Conference & EXPO and StarWest. You can reach her at [llrose@us.ibm.com](mailto:llrose@us.ibm.com).

## Share this....

 [Digg this story](#)

 [del.icio.us](#)

 [Slashdot it!](#)