

# extreme TESTING

Why aggressive software development  
calls for radical testing efforts

by Ronald E. Jeffries

**V**ery rapid development of high-quality applications software is quickly becoming an expected norm. How can you and your team ride that wave and deliver the goods? Extreme Programming, the rapid application development practice formulated by Kent Beck, may be the answer. This humanistic discipline combines simplicity, communication, feedback, and aggressiveness to produce high-quality software very quickly.

To develop software rapidly, you start simply, and evolve quickly from release to release. This is the standard “incremental development” approach recommended by many. But to go fast and win, you have to live through change after change after change. Feedback is vital, and the most basic and critical feedback is that of Extreme Testing—the best way to survive extreme change.

Testing isn't just for testers anymore. With Extreme Testing, you can develop software more quickly, with more confidence, and with higher quality.

We'll get to the details of Extreme Testing shortly, but first let me tell you a little story of how it works.

More than two-thirds of the way through the development of a famous Extreme payroll project, the customers asked for a very tricky change. We had been storing and displaying entitlements—the money you get—as positive, and deductions as negative. It made sense to us.

But the payroll people wanted both entitlements and deductions handled as positive—a lease car deduction of **\$110**, for example, not **(\$110)**. This was awful because the system saves hundreds of internal values and displays them on demand. Their request affected everything.

Part of the change wasn't very challenging. There were “bins” of money, and the system already knew which ones were entitlements and which were deductions. We just changed things so that deduction bins were always subtracted from entitlements when they were combined.

The devil, as always, was in the details. There were lots of places where entitlements and deductions were added, using procedural code rather than bins. Each of these additions had to be changed to include some subtraction. Similarly, there were tests for negative that had to be

changed to positive, and so on. After a couple of days of getting the base support in place, a team of two of us went ahead and started changing things.

Over two-thirds through the project, would you want to change this critical assumption? With Extreme Testing, it was almost easy!

We had, at that time, around 2,000 Extreme Tests. We had Unit Tests testing the behavior of each class, and we had Functional Tests computing hundreds of paychecks and validating thousands of values. We just started running all of the tests.

Most of them worked, but hundreds did not. Each of these failures was due to a comparison or arithmetic operation somewhere, and of course, each such operation was affecting a lot of tests.

## ▶▶ QUICK LOOK

- Applying extreme testing to Unit Tests
- Key dimensions of extreme Functional Tests
- Test reporting
- Questions and answers about feasibility

The two of us worked through the tests one at a time, while the rest of the team pretended to ignore the tests being wrong. In one week, we had all the tests back where they had been, and had stumbled across and fixed a number of unrelated problems as well.

That was when we knew that Extreme Testing was magic: we had fearlessly changed an essential assumption of the system, well into development, and in a week, *we were certain that it had worked!* Over the life of the system, we have found few defects, if any, that were due to this extensive change.

Extreme Testing is all about confidence. **Unit Tests** let developers evolve the system rapidly and with confidence, and **Functional Tests** give customers and developers confidence that the whole product is progressing in the right direction.

## Unit Tests

Rapid development implies rapid change to the code base. We need to build small increments of business value very quickly. This means that we can't invest up front in general-purpose support software: we have to let it evolve into being.

We're always learning. Developers have to learn how to build the system, and customers have to learn what they really want. We can connect and overlap these two learning processes by building the software and changing it quickly as we both learn.

This is "just in time" software development. To go this way, we embrace the fact that the things we build will need to change. We approach development to make it easy to transform yesterday's code into what we need tomorrow. To accomplish this, we make sure that our code is "well-factored."

While a complete description of good factoring is beyond the scope of this article (watch for Martin Fowler's forthcoming book on refactoring for excellent coverage of the subject), I'll just say that well-factored means that the

code does everything "once and only once." Because it only does one thing, it's easy to change that one thing. Because that thing is done in only one place, it's easy to find what to change. It's what in the olden days we used to call modularity.

Changing well-factored code is easy. If at some point in the future we see our needs more clearly, it is very likely that the change we'll need is localized if our code is well-factored.

Even so, all this rapid change adds risk. We need to be sure of two things: the new capability works, and we haven't broken anything that used to work. And that requires testing.

There are two things to be sure of, so Extreme Testing specifies two key actions:

1. To be sure that new features work, write Unit Tests for every feature. Write them before you release the code, preferably before you even write it. Save all the unit tests for the whole system.
2. To be sure that nothing else is broken, run all the Unit Tests in the entire system before any code is released—and ensure that those tests run at 100 percent!

Let me emphasize that last point. Whenever Extreme Programmers release any code at all, **every unit test in the entire system must be running at 100 percent!** That shows us not just that the new feature works, but that the changes haven't broken anything anywhere.

Set aside your doubts, for a moment, about whether your programmers would do it, or whether you could afford to do it. We'll come back to that.

Just bask in the essential truth: if you had tests for everything, and they always ran at 100 percent when new code was released, you would have incredible certainty that you hadn't broken anything—no matter how complex the change you had made. Would that be worth having? You're into testing, or you wouldn't be reading this. You *know* it would be worth having!

## The Four Commandments of Extreme Programming (XP)

**SIMPLICITY** means that the software is developed using the simplest possible design and constructs, but it means more: simplicity pervades the entire process. XP eliminates, as much as possible, the unnecessary elements of building software. One of our rules is "You aren't going to need it," which reminds us to add software or process only when we really need them, not in anticipation of need.

**COMMUNICATION** is key to rapid development and to customer satisfaction. XP embodies communication with a focus on simplicity: Use person-to-person communication instead of written documents wherever possible.

**FEEDBACK** is important to any development process, but when you are trying to eliminate everything you can, you need feedback to be sure you're on track. Software testing is a major source of quality feedback in XP, but we include resource, scope, and time feedback as well.

**AGGRESSIVENESS** in moving forward is possible when you take the simplest possible approach and employ a process high in communication and feedback.

## Functional Tests

For rapid incremental development to be successful, team confidence must be high. Unit Tests give the developers confidence. Customers and management need confidence too, to keep putting up the money and time. Their confidence comes from seeing the system actually do what is being paid for.

Since each increment adds specific business value, include tests that check each increment to see whether the value is there. In Extreme Programming these are called Functional Tests.

In a payroll system, for example, an early increment might call for giving employees their regular base pay for days worked, with no special entitlements, and taking a straight tax deduction. These employees still have to be read in from the database, their pay

rate determined, and so on.

Functional Tests for this can just check a small number of employees. Each employee has his pay rate checked for correctness, with the pay amount calculated for the days worked, the tax taken, and the net amount.

In a later increment—with overtime, shift premium, sick days, comp time, federal, state, and local taxes included—we have to check many more employees and more values for each employee in order to cover all the cases.

Here are the key dimensions of Extreme Functional Tests:

**Customer-owned** To get confidence from the tests, the customers must understand them, and should provide the values tested.

**Comprehensive** We tell our customers: you only have to test it if you want it to work. But invest wisely. Don't just shotgun a million values: pick values where the test will have meaning if it succeeds *and* if it fails.

**Repeatable** The rules of pay won't change—except to get more complex. Every time you do a new increment, write new tests; but upgrade all the old tests, and make *them* work too.

**Automatic** You can't have comprehensive repeatable tests if you have to manually check the results. Have a testing facility to set up and run the tests, check the results, and report them.

**Timely** If the tests don't show up until weeks after the development is done, it just slows things down. Everyone's mind is on something else by then, and it's disruptive to turn back time. Get Functional Tests for an increment running (although probably failing) by halfway through the increment. Developers improve and release code during the increment, and get feedback as they go. Rapid feedback enables rapid development.

**Public** The developers need rapid feedback, as we have seen. But customers and management need to see progress as well. With all this testing, things will probably go well, and everyone will know it. But if things aren't going so well, declines in test scores will alert everyone to the problem. Let's face it: under pressure, developers may start to ignore test scores. The testing group, the customers, or management may need to step in and put things back on track.

## Test Reporting

Now that we have all these tests, how do we report on the results? Here, as everywhere, Extreme Programming offers a simple answer. No reporting has to be done on Unit Tests—they are always at 100%. For Functional Tests, the interesting questions are:

- How many Functional Tests are there?
- Have the customers validated all answers?
- How many tests are running correctly?
- What's the trend?

There are hundreds of Functional

Tests. An Extreme test runs all of them, every day, on the integrated system. We can display the scores in a graph, as shown in Figure 1.

Here we see test status at the end of each development increment. This represents about three weeks' work, with perhaps ten or twenty user stories (think mini-use cases or scenarios) implemented by ten developers. A story should encompass about three days of "perfect engineering time," spread over six to nine real days.

In one simple graph, we get the overall picture of quality for the project. Green means the test is getting the right answer; red means wrong; yellow means the customers have told us what system output to test, but have not yet provided a validated answer to compare against.

A single graph like this can tell almost the complete story of quality, in a single glance. Early on, the widening yellow band says we were having trouble getting validated answers. Then, as validated answers started to come along, we learned that the system was making a lot of errors. In increment eight, quality took a big step upward, and has improved ever since. Finally, in increment twelve, all the data values are validated, and there are just a few tests to fix to be at 100%. We see that this has been a healthy project, and we can have great confidence in the quality of the system.

Extreme projects update graphs like these on a daily basis, posting them prominently on the wall for everyone to see. Developers, customers, and management all see the truth of the project's progress, all the time. While the project may keep detailed graphs for various suites of tests, we find that the bulk of the benefit comes from this one graph.

## But Can You Do It?

Your reaction to all this is probably that it sounds good, but *could* you do it and could you *afford* to do it? Here are some questions and answers based on actual experience with Extreme Testing in development.

### Will the developers actually write Unit Tests?

Teams who try working with Extreme Programming's

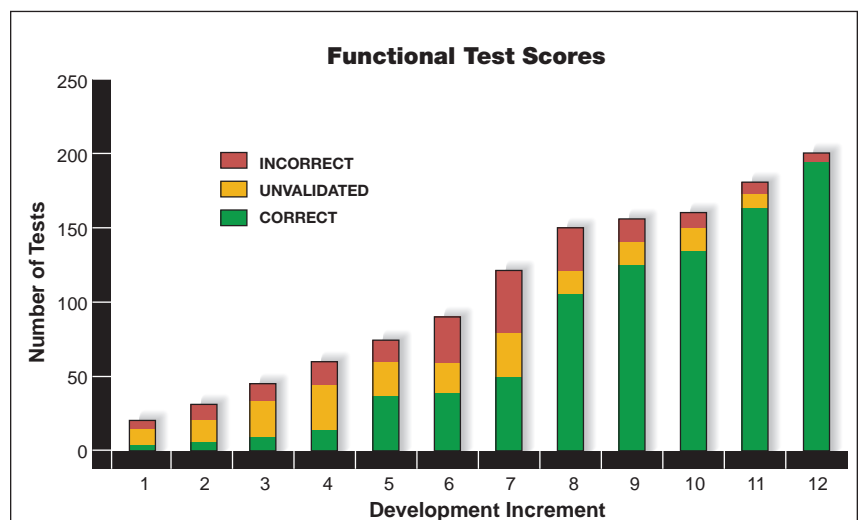


FIGURE 1: Functional test status at the end of each development increment.

extensive up-front Unit Tests will never go back. New work goes much faster when you have tests to show you when you are done. Refactoring and other revisions go much faster when you have tests to show you that you haven't broken anything. Software release goes much faster when you run the tests before every release, because if anything breaks you know almost exactly where the problem is. Developers who work with tests get to spend more time working with new code, and less time trying to find obscure bugs in old code.

**Yes, but how do I get them to try it?** This could be a challenge. Most teams are willing to try an experiment for a few iterations, and most will experience enough early success to keep going. You may find it desirable to have a coach with the team, at least during the early days of trying any of the Extreme Programming disciplines.

**Doesn't writing all that extra test code slow things down?** Only if you look at a very narrow window of time. Instead, picture the days you'll have to spend debugging when a problem only shows up in integration after everyone has been releasing code for days or weeks. Imagine your developers saying that some part of the system is so fragile that it can't be enhanced in the direction you need to go. And think of the fear of dealing with some part of the system that only one programmer understands, after he's gone.

**Won't the tests either take a long time to run, or not get run at all?** If you don't run your tests, you don't know if the system works. If the tests run too slowly, you have the wrong tests, or the wrong testing facility. Keep the tests comprehensive, but lean and efficient.

## Yes, You Can!

Extreme Programming is extreme in simplicity, in communication, and in aggressiveness. To make this work, we are also extreme in the amount of testing we do. This combination lets us develop software rapidly, confidently, and with a lot of fun in the process. It will work for you, too. **STQE**

---

*Ron Jeffries has been developing software since 1961, when he accidentally got a summer job at Strategic Air Command HQ, and they accidentally gave him a FORTRAN manual. He and his teams have built operating systems, language compilers, relational and set-theoretic database systems, manufacturing control, and applications software, producing about a half-billion dollars in revenue, and he wonders why he didn't get any of it. For the past few years he has been learning, applying, and teaching Extreme Programming, teamed with Kent Beck, Ken Auer, Ward Cunningham, and Martin Fowler. Ron can be reached at ronjeffries@acm.org.*

## Testing Frameworks

**O**ur payroll project example uses Kent Beck's Testing Framework as the basis for both Unit and Functional Tests. This framework is widely available for Smalltalk, Java, and C++ (see the Webinfolink section of the magazine's Web site for resource information).

Unit Tests belong to programmers, so the unit testing facilities are optimized for that purpose. We have our tests divided into "suites," which are interesting combinations of related tests. With smaller suites, we can run a few related tests while we're working in a specific area. When tests run quickly, developers test more frequently and get important feedback sooner.



We use a very simple GUI that lets us open a suite and run it. The GUI displays a running percentage of tests completed while they are running and a total score when completed. The GUI above is showing that the tests are complete and the score is only 91%. This is bad, which is why the screen is red. The developer needs to fix things up. Remember that every test in the entire system must

be brought up to 100% before the developer can release her code.

Now the GUI shows the tests complete, and the developer can release her software. We can say that with confidence because we required all the Unit Tests in the system to run at 100% before release, and single-threaded releases on a single integration machine. Success!

We built a more comprehensive framework for Functional Testing because it needs to



make sense to users rather than developers. Its GUI displays the result of paying an employee, with lines for each internal or exported value the customers want to check. Each line shows an expected value and the value computed by the system. Lines where the values differ are highlighted.

Although this tool is very powerful, we built it essentially from scratch in only a few weeks. The essence is simple: just compute values in the system and compare them to values saved on a file.