

The Addison-Wesley Signature Series



A KENT BECK SIGNATURE
BOOK

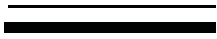
LEADING LEAN SOFTWARE DEVELOPMENT

RESULTS ARE NOT THE POINT

MARY AND TOM
POPPENDIECK



*Foreword by Dottie Acton,
Senior Fellow, Lockheed Martin*



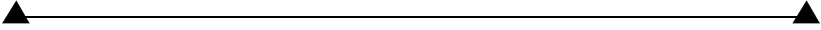
Leading Lean Software Development

would spread their knowledge around and speed things up at the same time,” I suggested.

“No, that’s impossible. Our system grew up over time and now it’s a complex web of interlocking applications,” he said. And so was his organization’s structure, apparently.

Implementing Scrum in this organization was not helping much. The managers had been hoping Scrum would be an easy way to speed up development, but they had ignored the underlying need to rationalize their complex technical and organizational architecture. I suspected that until they addressed the high-dependency architecture, they would not be able to form feature teams that could rapidly and independently deliver customer-focused features.

Mary Poppendieck



It is difficult for agile development to flourish in a highly interdependent environment, because teams cannot independently make and meet commitments. Agile development works best with cross-functional teams in which a single team or small group of teams has the skill and authority necessary to deliver useful feature sets to customers independent of other teams. This means that whenever possible, teams should be formed along the lines of features or services. Of course, this is most useful if those features or services are relatively small and have clean interfaces. Conway’s Law is difficult to escape.

Another big mistake we see in organizational structures is the separation of software development from the larger system supported by the software. For example, when hardware and software are involved, we often find separate hardware and software (and sometimes firmware) teams. This is similar to forming teams along technology layer lines. Conway’s Law says that it might be better to organize teams (or groups of teams) that are responsible for both the hardware *and* the software of a subsystem of the product.

Consider, for example, a digital camera with a picture-capture subsystem to set exposure, capture, and process pictures; a storage subsystem to store and retrieve pictures to a memory card; an electronic interface subsystem to transfer pictures to a computer; and possibly a smart lens subsystem. Each subsystem is software-intensive, but the camera should not be divided into a hardware subsystem and a software subsystem. Why? Suppose there were a change to the camera’s focusing system. With subsystem teams, only the picture-capture team would be involved in the change. With hardware and software teams, both teams would be involved and the change would not be obviously limited to a few people. When changes can be limited to a smaller subsystem, the cost and risk of change are significantly reduced.

Similarly, when business processes are being changed, it is often best to place software developers on the business process redesign team, rather than have separate software development teams. This is not to imply that the software developers would no longer have a technical home with technical colleagues and management. As we will see later, maintaining technical competencies and good technical leadership is essential. The organizational challenge is to provide people with a technical base even as they work on cross-functional teams. We will take up this challenge again at the end of the chapter.

Frame 6: Quality by Construction

*If you want more effective programmers, you will discover that they should not waste their time debugging—they should not introduce bugs to start with.*³⁸

—Edsger W. Dijkstra

Virtually all software development methodologies share the same objective: Lower the cost of correcting mistakes by discovering them as soon as they are made and correcting them immediately. However, as we demonstrated earlier, various methodologies differ dramatically in the way they frame this goal. The sequential life cycle approach is to create a complete, detailed design so as to find defects by inspection before any coding happens—because fixing design defects before coding is supposedly less expensive than fixing design defects later on. This theory has a fundamental problem: It separates design from implementation. A careful reading of the 1968 NATO conference minutes shows that separating design from implementation was regarded as a bad idea even as the term *software engineering* was coined.³⁹

But it gets worse. Sequential development leaves testing until the end of the development process—exactly the wrong time to find defects. There is no question that finding and fixing defects late in the development cycle is very expensive; this is a premise of sequential development in the first place. And yet, sequential approaches have brought about exactly the result they are trying to prevent: They delay testing until long after defects are injected into the code. In fact, people working in sequential processes *expect* to discover defects during final testing; often a third or more of a software release cycle is reserved to find

38. Dijkstra, “The Humble Programmer,” 1972.

39. Naur and Randell, *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee*, 1968.

and fix defects after coding is theoretically “complete.” The irony is that sequential processes are designed to find and fix defects early, but in most cases they end up finding and fixing defects much later than necessary.

Where did we get the idea that finding and fixing defects at the end of the development process is appropriate? Just about every quality approach we know of teaches exactly the opposite. We should know that it is best to avoid defects in the first place, or at least to find defects immediately after they have been inserted into the code. It’s the same as losing your keys: If you lost your keys a few seconds ago, look down. If you lost them a few minutes ago, retrace your steps. If you lost them a week ago, change the locks.⁴⁰ And yet we wait until the end of the development process to test our code, and by that time we need a lot of locksmiths. Why do we do this?

Part of the reason why we delay testing is because testing has historically been labor-intensive and difficult to perform on an incomplete system. With a lean frame of mind, we can no longer accept this historical mental model; we must change it. In fact, changing the historical mindset regarding testing is probably the most important characteristic we have seen in successful agile development initiatives. You can put all of the agile project management practices you want in place, but if you ignore the disciplines of test-driven development and continuous integration, your chances of success are greatly diminished. These so-called engineering disciplines *are not optional* and should not be left to chance.

Test-Driven Development

So how do you keep from introducing defects into code? Dijkstra was clear on that; you have to construct programs “from the point of view of provability.” This means that you start by stating the conditions that must hold if the program is to perform correctly. Then you write the code that makes the conditions come true.⁴¹ In fact, formal methods for constructing correct code have been around a long time, but they have been historically difficult to use and have not found practical widespread application. So what has changed?

xUnit Frameworks

In 1994, Kent Beck published “Simple Smalltalk Testing” in the *Smalltalk Report*.⁴² In three pages he outlined a framework for testing Smalltalk code along with suggestions for how to use the testing framework. In 1997, Kent

40. Thanks to Tomo Lennox for this analogy.

41. Freeman, *Software Systems Principles: A Survey*, 1975, p. 489.

42. Beck, “Simple Smalltalk Testing,” 1994, pp. 16–18.

Beck and Erich Gamma wrote a version of the same testing framework for Java and called it JUnit. Many more xUnit frameworks have been written for other languages, and these have become the first widely used—and practical—tools for constructing correct programs.

How can testing be considered a tool for constructing a correct program? Isn't testing after-the-fact? It is a bit of a misnomer to call the xUnit frameworks testing frameworks, because the recommended approach is to use these frameworks to write design specifications for the code. xUnit “tests” state what the code must do in order to perform correctly; then the code is written that makes these “tests” pass. While specifications written in xUnit frameworks may be less rigorous than formal methods such as Design by Contract, their simplicity and flexibility have led to widespread adoption. A well-considered xUnit test harness, developed and run wisely, makes quality by construction an achievable goal.

xUnit “tests,” or their equivalent using some other tool, are the software specification; they specify, from a technical perspective, how the smallest units of code will work, and how these units will interact with each other to form a component. These specifications are written by developers as a tool for designing the software. They are not—*ever*—written in a big batch that is completed before (or after) the code is written. They are written one test at a time, just before the code is written, by the developer writing the code.

There are tools to measure what percentage of a code base is covered by xUnit tests, but 100% code coverage is not the point; good design is the point. You should experiment to find out what level of code coverage generates good design in your environment. On the other hand, 100% of the xUnit tests you have in a harness should *always* pass. If they don't pass, the policy must be: *Stop*. Find and fix the problem, or else back out the code that caused the failure. A unit test harness with tests that always pass means that when there is a failure, the fault was triggered by the last code checked into the system. This eliminates large amounts of time wasted on debugging; Dijkstra's challenge has been met.

Unit tests are not universally applicable. For example, code produced by code generators—perhaps fourth-generation languages or domain-specific languages—is often not amenable to unit testing; you will probably have to use some form of acceptance testing instead. In addition, it is usually not practical to write unit tests for existing code bases, except for code that you are changing.

For new code, or for changes to existing code, xUnit tests or their equivalent should be considered *mandatory*. Why? Because writing tests first, then writing the code to pass the tests, means that the code will be *testable* code. If you want to change the code in the future, you will want testable code so you can make changes with confidence. Of course, this means that the tests have to be maintained over time, just like other code.

Acceptance Tests

In many organizations, a relatively detailed specification of what software is supposed to do from a customer's perspective is written before code is written. This specification should be written by people who truly understand the details of what the system is supposed to do, assisted by the members of the development team who will implement the features. The details of the specification should not be written long before the code; no doubt they will change, and the effort will be wasted. The detailed specification should be written iteratively, in a just-in-time manner.

There are traditionally two uses of a software specification: Developers write code to meet the specification, and testers create various types of tests and write test scripts to assure that the code meets the specification. The problem is, the two groups usually work separately, and worse, the testers don't write the tests until after the code is written. Under this scenario, the developers invariably interpret the details of the specification somewhat differently than the testers, so the code and tests do not match. Well after the code is written, the tests are run and these discrepancies are discovered. This is not a mistake-proofing process; this is a defect-injection process (see Figure 2-3).

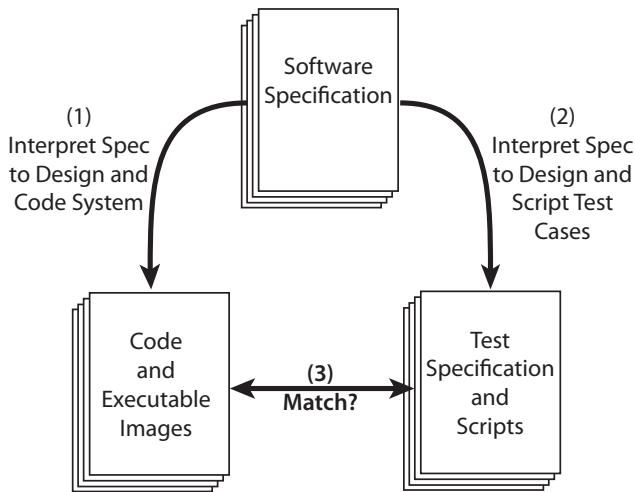


Figure 2-3 A defect-injection process

To mistake-proof this process, the order of development needs to be changed. Tests should be created first—before the code is written. Either the tests are derived from the specification, or better yet, the tests *are* the specification. In either case, coding is not done until the tests for that section of code are available, because then developers know what it is they are supposed to code. This is not cheating; it is mistake-proofing the process (see Figure 2-4).

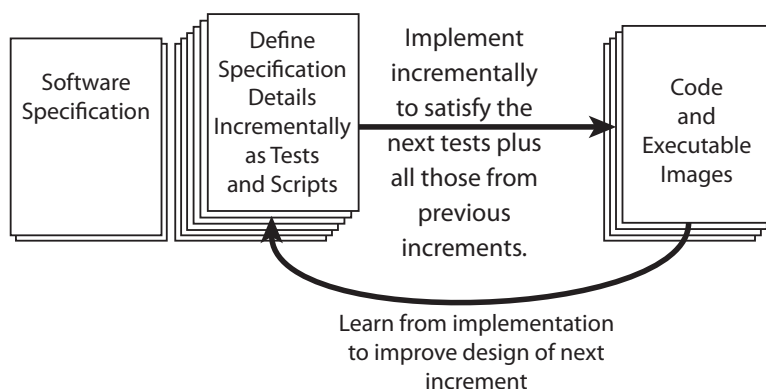


Figure 2-4 *A mistake-proofed process*

The preferred approach to specifying the details of a software product or application is to write acceptance tests. These tests are end-to-end tests that specify what the application is supposed to do from a business perspective, so they are quite different from unit tests. When the specification is written as executable tests, it can be run repeatedly as small parts of the code are written, and defects can thus be detected and fixed the moment they are injected into the system.

Of course, not all specifications can be written as automated tests, and automated tests must be created with insight or they will become a complex burden just as fast as any other software. However, creating an intelligent harness for testing that the code does what it is supposed to do is the best way we know of to produce code that—by definition—meets the specification.

Test Automation

Executable tests (specifications, actually), whether they are acceptance tests or xUnit tests or other tests, are the building blocks of mistake-proofing the software development process. These tests are put into a test harness, and then every time new code is written, the code is added to the system and the test harness is run. If the tests pass, you know not only that the new code works, but also that everything that used to work is still working. If the tests don't pass, it's clear that the last code you added has a problem or at least has exposed a problem, so it's backed out and the problem is immediately addressed.

If this sounds easy, rest assured that it is not. Deciding what to automate, writing good tests, automating them, running them at the right time, and maintaining them is every bit as challenging as writing good code. It requires a good test architecture, sensible choices as to what to include in various test suites, and ongoing updating of tests so that they always match the current code.