Three Essential Tools for Stable Development©

Andy Hunt and Dave Thomas
The Pragmatic Programmers, LLC

Three basic practices make the difference between a software project that succeeds and one that fails. These practices support and reinforce each other; when done properly, they form an interlocking safety net to help ensure success and prevent common project disasters. However, few development teams in the United States use these proven techniques, and even fewer use them correctly.

Many software projects that fail seem to fail for very similar reasons. After observing – and helping – many of these ailing projects over the past couple of decades, it seems clear to us that a majority of common problems can be traced back to a lack of three very basic practices. Fortunately, these three practices are easy and relatively inexpensive to adopt. It does not require a large-scale, expensive, or bureaucratic effort; with just these practices in place, your team can work at top speed with increased parallelism. You will never lose precious work, and you will know immediately when the development starts to veer off-track in time to correct it, cheaply and easily.

The three basic practices that we have identified as being the most crucial are *version control*, *unit testing*, and *automation*. Version control is an obvious *best practice*, yet nearly 40 percent of software projects in the United States do not use any form of version control for their source code files [1]. The motto of these shops seems to be *last one in wins*. That is, they will use a shared drive of some sort and hope that no one overwrites their changes as the software evolves. Hope is a pretty poor methodology, and these teams regularly lose precious work. Developers begin to fear making any changes at all, in case they accidentally make the system worse. Of course, this fear becomes a self-fulfilling prophecy as necessary changes are neglected and the system begins to degrade.

Unit testing is a coding technique for programmers so they can verify that the code they just wrote actually does something akin to their intent. It may or may not fulfill the requirements, but that is a separate question: If the code does not do what the programmer thought it did, then any further testing or validation is both meaningless and a large waste of time and money (two items that are in short supply to begin with). Developer-centric unit testing is a great way to introduce basic regression testing, create more modularized code that is easier to maintain, and ensure that new work does not break existing work. Despite the effectiveness of this technique in both improving design and identifying and preventing defects (aka bugs), 76 percent of companies in the United States do not even try it [2].

Automation is a catchall category that includes regular, unattended project builds, including regression tests and push-button convenience for day-to-day activities. Regular builds ensure that the product can be built to catch simple mistakes early and easily, when fixing them is the cheapest. When implemented properly, it is as if you have an ever-vigilant guardian looking over your shoulder, warning you as soon as there is a problem. Incredibly, some 70 percent of projects in the United States do not have any sort of daily build [2]. By the time they discover a problem, it has metastasized into a much larger and potentially fatal problem.

We will briefly examine each of these areas, with an in-depth look at unit testing in particular. We will outline the important ideas, synergies, and caveats for each of these practices so your team can either begin using them or improve your current use of them.

Version Control

Everyone can agree that version control is a best practice but even with it in place, is it being used effectively? Ask yourself these questions: Can you re-create your software exactly as it existed on January 8? When a bug is found that affects multiple versions of your released software, can your team fix it just once, and then apply that fix to the different versions automatically? Can a developer quickly back out of a bad piece of code?

There is more to version control than just keeping track of files. But before we proceed, we need to define some simple terminology: We use *check-in* to mean that a programmer has submitted his or her changes to the version control system. We use *checkout* to refer to getting a personal version of code from the version control system into a local working area.

When a programmer checks in code, it is now potentially available to the rest of the team. As such, it is only polite to ensure that this new code actually compiles successfully; it should be accompanied by unit tests (more on this later), and those tests should pass. All the other passing tests in the system should continue to pass as well – if they suddenly fail, then you can easily trace the failure to the new code that was introduced.

It is far easier to track down these sort of problems right at the point of creation instead of days, weeks, or even months later. To exploit this effect, you must allow and encourage frequent check-ins of code multiple times per day. It is not unusual to see team members check-in code 10-20 times a day. It *is* unusual – and very dangerous – to allow a programmer to go a few days or a week or more without checking in code.

Because check-ins occur so frequently, these and other day-to-day operations must be very fast and low ceremony. A check-in or checkout of code should not take more than five to 15 seconds in general. If it takes an hour, people will not do it, and you have lost the advantage.

Now some people get a little nervous when they read this part. They fret that all of this code is being dumped into the system without being reviewed, tested by QA, audited, or whatever else their methodology or environment demands. They are rightfully concerned that this code is not yet ready to be part of a release. Nonetheless, it must still be in the version control system so that it is protected.

Most version control systems provide a mechanism to differentiate ongoing development changes from official release candidates. Some feature explicit *promotion*

© 2004 The Pragmatic Programmers, LLC. Portions of this article adapted from "Pragmatic Unit Testing in Java With JUnit," by Andy Hunt and Dave Thomas (Volume II of the Pragmatic Starter Kit), published by the Pragmatic Bookshelf and Copyright © 2003, 2004 The Pragmatic Programmers, LLC. Reprinted with permission.

commands to allow this. You can accomplish the same thing in other systems by using tags (or version labels) to identify stable release versions of source code as opposed to code that is in progress.

Regardless of the mechanism, it must be an easy operation to promote development changes to an official release status. On the other side of the coin, you need to be able to back out changes and any disastrous new code when needed.

Finally, you need to be able to re-create any product built at any previous point in time. This ability to go back in time is crucial for effective debugging and problem solving (just think of any developer who starts a discussion with, "Well, it used to work").

Commercial and freely available version control systems vary in complexity, features, and ease of administration. But one feature in particular is worth examining: whether it supports strict locking or optimistic locking. In systems under strict locking, only one person can edit a file at a time. While that sounds like a good idea, it turns out to be unduly restrictive in practice. We favor the Concurrent Version System <www.cvshome.org> described in [3].

You may find you can increase parallelism and efficiency in your team by using a system that features optimistic locking. In these systems, multiple people can edit the same source code file simultaneously. The system uses conflict-resolution algorithms to merge the disparate changes together in a sensible manner. Ninety-nine percent of the time it works perfectly without intervention. Occasionally, however, there is a conflict that must be addressed manually. At no point is anyone's work in danger of being lost, and it ends up being much more efficient to coordinate just these few conflicts by hand instead of having everyone coordinate *every* change with the rest of the team.

Unit Testing

When a developer makes a change to the code on your project, what feedback is available? Does the developer have any way of knowing if the new code broke anything else? Better still, how do *you* know if any developer has broken anything today? A system of automated unit tests will give you this information in real-time.

Programming languages are notorious for doing exactly what programmers say, not what they mean. Like a petulant child that takes your expressions completely literally, the computer follows our instructions to the letter, with no regard at all to our intent. Technology has yet to produce the compiler that implements with *do what I mean, not what I say.*

So in keeping with the idea of finding and fixing problems as soon as they occur, you want programmers to use unit tests (or checked examples) to verify the computer's literal interpretation of their commands. It is really no different from following through with a subordinate to verify that a delegated task was performed – except that instead of just checking once, automated unit tests will check and recheck every time any code is changed.

There are some requirements to using this style of development, however:

- The code base must be decoupled enough to allow testing. When code is tightly coupled, it is very difficult to test individual pieces in isolation, and harder to devise unit tests that exercise specific areas of functionality. Well-written code, on the other hand, is easy to test. If your team finds that the code is difficult to test, then take that as a warning sign that the code is in serious trouble to begin with.
- Only check-in tested code. As we mentioned above, checking-in foists a programmer's code onto the rest of the team. Once it is available to everyone, then the whole team will begin to rely on it. Because of this reliance, all code that is checked in must pass its own tests.
- In addition to passing its own tests, the programmer checking in the code must ensure nothing else breaks, either. This simple regression helps prevent that frustrating feeling of *one step forward, two steps back* that becomes commonplace when code fixes cause collateral damage to other parts of the code base. Usually these bugs then require fixes, which in turn cause more damage, and so on. The discipline of keeping all the tests running all the time prevents that particular death-spiral.
- There should be at least as much test code as production code. You might think that is excessive, but it is really just a question of where the value of the system resides. We firmly believe the code that implements the system is not where the value of your intellectual property lies. Code can be rewritten and replaced, and the new code (even an entirely new system) can be verified against the existing tests. Now the most precise specification of the system is in executable form – the unit tests. The learning and experience that goes into creating the unit tests is invaluable, and the tests themselves are the best expression we have of that knowledge.

We will look at implementing unit tests (aka checked examples) in much greater detail later in this article.

Automation

An old saying goes the *cobbler's children have no shoes*. This saying is particularly appropriate for our use of software tools during software development. We see teams routinely waste time using manual procedures that could easily be automated.

Everyone clamors for software development to be more defined and repeatable. Well, the *design* and implementation of software probably cannot be made repeatable any more than you could make the process of making hit movies repeatable. But the *production* of software is another matter entirely.

The process of taking source code files, bits of eXtensible Markup Language, libraries, and other resources and producing an executable for the end user should be precisely repeatable. Given the same inputs, you want the same outputs, every time, without excuses. In combination with version control, you want to be able to go back in time and reproduce that same pile of bits that you would have produced on January 8 just as easily. That comes in very handy should the Department of Justice ask for it politely, or a frustrated customer asks for it somewhat less politely to work around some outstanding bug.

The rule we try to adopt is that any manual process that is repeated twice is likely to be repeated a third time – or more – so it needs to be encapsulated within a shell script, batch file, piece of Java code, Job Control Language, or whatever.

Unit tests, as well as functional and acceptance tests, should be run automatically as well as be part of the build process. You will probably want to run the unit tests (which should execute very quickly) with every build; automatic functional and acceptance tests might take longer and you may only want to run those once a week, or when convenient.

You see, not only does automation make developer's lives easier by providing push-button convenience, it helps keep the feedback coming by constantly checking the state of the software. Automated builds are constantly asking two questions: Does the software build correctly? Do all the tests still pass a basic regression? With the computer performing these checks regularly, developers do not have to. Problems can be identified as soon as they happen, and the appropriate developer or team lead can be notified immediately of the problem [4]. Problems can be fixed quickly, before they have a chance to cause any additional damage. That is the benefit we want from automation.

Finally, consider how the build communicates to the development team and its management. Does the team lead look at the latest results in some log file and then report status to management? Does not that constitute a manual process? It is relatively easy to set up visual display devices, ranging from liquid crystal display screens to bubbling lava-style lamps to the new and popular Ambient Orb [4].

Synergy

These three practices interlock to provide a genuine safety net for developers. Version control is the foundation. Unit tests and scripts for automation are under version control, but version control needs automation to be effective. Unit testing needs both version control and automation.

With the combination, developers can better afford to take chances, experiment, and find the best solutions. The Rule of Three says that if you have not proposed at least three solutions to a problem then you have not thought about it hard enough. With this set of practices in place, developers can realistically try out a number of different solutions to a problem: Version control will keep them separate, and unit testing will help confirm the viability of each solution. All this with plenty of automated support, including continuous, ongoing checks ensures that the team does not wander too far off into the woods. This is how modern, successful software development is done.

Unit Testing With Your Right-BICEP

You can strengthen your organization's testing skills by looking at six specific areas of code that may need unit tests. These areas are remembered easily using the mnemonic Right-BICEP [5]:

Right	Are the results *right*?
B	Are all the *boundary* conditions correct?
I	Can you check *inverse* relationships?
C	Can you *cross-check* results using other means?
E	Can you force *error* conditions to happen?
P	Are *performance* characteristics within bounds?

Are the Results Right?

The first and most obvious area to test is simply to see if the expected results are right – to validate the results. These are usually the *easy* tests, as they represent the answer to the key question: If the code ran correctly, how would I know? Here is an example of how being forced to think about testing helps developers code better: If this question cannot be answered satisfactorily, then writing the code – or the test – may be a complete waste of time.

"But wait," you cry out, "that does not sound very agile! What if the requirements are vague or incomplete? Does that mean we can't write code until all the requirements are firm?" No, it does not at all. If the requirements are truly not yet known, or not yet complete, you can always make some assumptions as a stake in the ground. They may not be correct from the user's point of view (or anyone else on the planet), but they let the team continue to develop. And, because you have written a test based on your assumption, you have now documented it – nothing is implicit.

Of course, you must then arrange for feedback with users or sponsors to fine-tune your assumptions. The definition of *correct* may change over the lifetime of the code in question, but at any point, you should be able to prove that it is doing what you think it ought.

Boundary Conditions

Identifying boundary conditions is one of the most valuable parts of unit testing because this is where most bugs generally live – at the edges. Some conditions you might want to think about include the following:

- Totally bogus or inconsistent input values such as a file name of !*W:X\\{\&Gi/w$>$g/h\#WQ@.
- Badly formatted data such as an e-mail address without a top-level domain <fred@foobar>.
- Empty or missing values such as 0, 0.0, "", or null.
- Values far in excess of reasonable expectations such as a person's age of 10,000 years.
- Duplicates in lists that should not have duplicates.
- Ordered lists that are not in order and vice-versa. Try handing a pre-sorted list to a sort algorithm, for instance, or even a reverse-sorted list.
- Things that arrive out of order, or happen out of expected order such as trying to print a document before logging in, for instance.

An easy way to think of possible boundary conditions is to remember the acronym CORRECT. For each of these items, consider whether or not similar conditions may exist in your method that you want to test, and what might happen if these conditions were violated [4]:

- **Conformance.** Does the value conform to an expected format?
- **Ordering.** Is the set of values ordered or unordered as appropriate?
- **Range.** Is the value within reasonable minimum and maximum values?
- **Reference.** Does the code reference anything external that is not under direct control of the code itself?
- **Existence.** Does the value exist (e.g., is non-null, non-zero, present in a set, etc.)?
- **Cardinality.** Are there exactly enough values?
- **Time (absolute and relative).** Is everything happening in order? At the right time? In time?

Check Inverse Relationships

Some methods can be checked by applying their logical inverse. For instance developers might check a method that calculates a square root by squaring the result, and testing that it is tolerably close to the original number. They might also check that some data was successfully inserted into a database by then searching for it, and so on.

Be cautious when the same person has written both the original routine and its inverse, as some bugs might be masked by a common error in both routines. Where possible, use a different source for the inverse test. In the square root example, we might use regular multiplication to test our method. For the database search, we will probably use a vendor-provided search routine to test our insertion.

Cross-Check Using Other Means

Developers might also be able to cross-check results of their method using different means. Usually there is more than one way to calculate some quantity; we might pick one algorithm over the others because it performs better or has other desirable characteristics. That is the one we will use in production, but we can use one of the other versions to cross-check our results in the test system. This technique is especially helpful when there is a proven, known way of accomplishing the task that happens to be too slow or too inflexible to use in production code.

Another way of looking at this is to use different pieces of data from the code itself to make sure they all *add up*. For instance, suppose you had some sort of system that automated a lending library. In this system, the number of copies of a particular book should always balance. That is, the number of copies that are checked out plus the number of copies sitting on the shelves should always equal the total number of copies in the collection. These are separate pieces of data, and may even be managed by different pieces of code, but they still have to agree and so can be used to cross-check one another.

Force Error Conditions

In the real world, errors happen. Disks fill up, network lines drop, e-mail goes into a black hole, and programs crash. You should be able to test that code handles all of these real-world problems by forcing errors to occur.

That is easy enough to do with invalid parameters and the like, but to simulate specific network errors – without unplugging any cables – takes some special techniques, including using mock objects.

In movie and television production, crews will often use *stand-ins*, or doubles, for the real actors. In particular, while the crews are setting up the lights and camera angles, they will use *lighting doubles*: inexpensive, unimportant people who are about the same height and complexion as the very expensive, important actors who remain safely lounging in their luxurious trailers.

The crew then tests their setup with the lighting doubles, measuring the distance from the camera to the stand-in's nose, adjusting the lighting until there are no unwanted shadows, and so on, while the obedient stand-in just stands there and does not whine or complain about *lacking motivation* for their character in this scene.

What you can do in unit testing is similar to the use of lighting doubles in the movies: Use a cheap stand-in that is kind of close to the real thing, at least superficially, but that will be easier to work with for your purposes.

Performance Characteristics

One area that might prove beneficial to examine is performance characteristics – not performance itself, but trends as input sizes grow, as problems become more complex, and so on. Why? We have all experienced applications that work fine for a year or so, but suddenly and inexplicably slow to a crawl. Often, this is the result of a silly error or oversight: A database administrator changed the indexing structure in the database, or a developer typed an extra zero into a loop counter.

What we would like to achieve is a quick regression test of performance characteristics. We want to do this regularly, every day at least, so that if we have inadvertently introduced a performance problem we will know about it sooner rather than later (because the nearer in time you are to the change that introduced the problem, the easier it is to work through the list of things that may have caused that problem).

So, to avoid shipping software with unsuspected performance problems, teams should consider writing some rough tests just to make sure that the performance curve remains stable. For instance, suppose the team is working on a component that lets users browse the Web from within their application. Part of the requirement is to filter out access to Web sites that we wish to block. The code works fine with a few dozen sample sites, but will it work as well with 10,000? 100,000? We can write a unit test to find out.

This gives us some assurance that we are still meeting performance targets. But because this one test takes six to seven seconds to run, we may not want to run it every time. As long as we run it (say) nightly, we will quickly be alerted to any problems we may introduce while there is still time to fix them.

Getting Started

All of the software tools mentioned in this article are freely available on the Web. To get started using these practices effectively, we recommend following this sequence:

1. Get everything into version control.
2. Arrange for automatic, daily builds. Increase these to multiple times per day or continuously as soon as the process begins to work smoothly.
3. Start writing unit tests for new code. Where needed, add some unit tests to existing code (but be pragmatic about it; only add tests if they will really help, not just for the sake of completeness).
4. Add the unit tests to the scheduled builds.

You can begin right away. Fire up that Web browser and start downloading some software if you do not already have it. These ideas will not fix all the problems on your project, of course, but they will provide your project with a firm footing so you can concentrate on the truly difficult problems.◆

References

1. Zeichick, Alan. "Debuggers, Source Control Keys to Quality." Software Development Times 1 Mar. 2002.
2. Cusumano, Michael, et al. "A Global Survey of Software Development Practices." Paper 178. MIT Sloan School of Management, June 2003.
3. Thomas, Dave, and Andy Hunt. Pragmatic Version Control With CVS. Raleigh, NC: Pragmatic Bookshelf, 2003 <www.PragmaticBookshelf.com>.
4. Clark, Mike. Pragmatic Project Automation. Raleigh, NC: Pragmatic Bookshelf, 2004 <www.PragmaticBookshelf.com>.
5. Hunt, Andy, and Dave Thomas. Pragmatic Unit Testing in Java With JUnit. Raleigh, NC: Pragmatic Bookshelf, 2003. (Also available in a C# version) <www.PragmaticBookshelf.com>.

About the Authors

Andy Hunt is an avid woodworker and musician, but curiously, he is more in demand as a consultant. He has worked in telecommunications, banking, financial services, and utilities, as well as more exotic fields such as medical imaging and graphic arts. Hunt is author of many articles, columns and books, and co-author of "The Pragmatic Programmer."

The Pragmatic Programmers, LLC
9650 Strickland RD
STE 103-255
Raleigh, NC 27615
Phone: (800) 699-7764
E-mail: andy@pragmatic programmer.com

Dave Thomas likes to fly single-engine airplanes and pays for his habit by finding elegant solutions to difficult problems, consulting in areas as diverse as aerospace, banking, financial services, telecommunications, travel and transport, and the Internet. Thomas is author of many articles, columns and books, and co-author of "The Pragmatic Programmer."

The Pragmatic Programmers, LLC
P.O. Box 293325
Lewisville, TX 75029
Phone: (972) 539-7832
E-mail: dave@pragmatic programmer.com