



# *Integrating unit testing into the organization*

---

## ***This chapter covers***

- Becoming an agent of change
- Implementing change from the top down or from the bottom up
- Preparing to answer the tough questions about unit testing

As a consultant, I've helped several companies, big and small, integrate test-driven development and unit testing into their organizational culture. Sometimes this has failed, but those companies that succeeded had several things in common. This chapter draws on stories from both camps as it looks at the following topics:

- *Becoming the agent of change*—The initial steps you should take before introducing any changes
- *Ways to succeed*—Things that contribute to successful changes in a process
- *Ways to fail*—Things that can destroy what you're trying to do
- *Tough questions and answers*—The most frequently asked questions encountered when introducing unit testing to a team

In any type of organization, changing people's habits is more psychological than technical. People don't like change, and change is usually accompanied with plenty of FUD (fear, uncertainty, and doubt) to go around. It won't be a walk in the park for most people, as you'll see in this chapter.

## **9.1 Steps to becoming an agent of change**

If you're going to be the agent of change in your organization, you should first accept that role. People will view you as the person responsible for what's happening, whether or not you want them to, and there's no use in hiding. In fact, hiding can cause things to go terribly wrong.

As you start to implement changes, people will start asking the tough questions about what they care about. How much time will this waste? What does this mean for me as a QA engineer? How do we know it works? Be prepared to answer. The answers to the most common questions are discussed in section 9.4. You'll find that convincing others inside the organization before you start making changes will help you immensely when you need to make tough decisions and answer those questions.

Finally, someone will have to stay at the helm, making sure the changes don't die for lack of momentum. That's you. There are ways to keep things alive, as you'll see in the next sections.

### **9.1.1 Be prepared for the tough questions**

Do your research. Read the answers at the end of this chapter, and look at the related resources. Read forums, mailing lists, and blogs, and consult with your peers. If you can answer your own tough questions, there's a good chance you can answer someone else's.

### **9.1.2 Convince insiders: champions and blockers**

Loneliness is a terrible thing, and not many things make you feel more alone in an organization than going against the current. If you're the only one who thinks what you're doing is a good idea, there's little reason for anyone to make an effort to implement what you're advocating. Consider who can help and hurt your efforts: the champions and blockers.

#### **CHAMPIONS**

As you start pushing for change, identify the people you think are most likely to help in your quest. They'll be your *champions*. They're usually early adopters, or people who are open-minded enough to try the things you're advocating. They may already be half convinced but are looking for an impetus to start the change. They may have even tried it and failed on their own.

Approach them before anyone else and ask for their opinions on what you're about to do. They may tell you some things that you hadn't considered: teams that might be good candidates to start with or places where people are more accepting of such changes. They may even tell you what to watch out for from their own personal experience.

By approaching them, you're helping to ensure that they're part of the process. People who feel part of the process usually try to help make it work. Make them your champions: ask them if they can help you and be the ones people can come to with questions. Prepare them for such events.

### **BLOCKERS**

Next, identify the *blockers*. These are the people in the organization who are most likely to resist the changes you're making. For example, a manager might object to adding unit tests, claiming that they'll add too much time to the development effort and increase the amount of code that needs to be maintained. Make them part of the process instead of resisters of it by giving them (at least those who are willing and able) an active role in the process.

The reasons why people might resist particular changes vary, and answers to some of the possible objections are covered in section 9.4. Some will be worried about job security, and some will feel comfortable with the way things are and object to any changes.

Going to these people and detailing all the things they could have done better is often nonconstructive, as I've found out the hard way. People don't like to be told what they don't do well. Instead, ask those people to help you in the process by being in charge of defining coding standards for unit tests, for example, or by doing code and test reviews with peers every other day. Or make them part of the team that chooses the course materials or outside consultants. You'll give them a new responsibility that will help them feel relied on and relevant in the organization. They need to be part of the change or they'll almost certainly undermine it.

### **9.1.3 Identify possible entry points**

Identify where in the organization you can start implementing changes. Most successful implementations take a steady route. Start with a pilot project in a small team, and see what happens. If all goes well, move on to other teams and other projects.

Here are some tips that will help you along the way:

- Choose smaller teams.
- Create subteams.
- Consider project feasibility.
- Use code and test reviews as a teaching tool.

These tips can take you a long way in a mostly hostile environment.

### **CHOOSE SMALLER TEAMS**

Identifying possible teams to start with is usually easy. You'll generally want a small team working on a low-profile project with low risks. If the risk is minimal, it's easier to convince people to try your proposed changes.

One caveat is that the team needs to have members who are open to changing the way they work and to learning new skills. Ironically, the people with less experience on a team are usually most likely to be open to change, and people with more experience

tend to be more entrenched in their way of doing things. If you can find a team with an experienced leader who's open to change, but that also includes less-experienced developers, it's likely that team will offer little resistance. Go to the team and ask their opinion on holding a pilot project. They'll tell you if this is (or is not) the right place to start.

#### **CREATE SUBTEAMS**

Another possible candidate for a pilot test is to form a subteam within an existing team. Almost every team will have a “black hole” component that needs to be maintained, and while it does many things right, it also has many bugs. Adding features for such a component is a tough task, and this kind of pain can drive people to experiment with a pilot project.

#### **CONSIDER PROJECT FEASIBILITY**

For a pilot project, make sure you're not biting off more than you can chew. It takes more experience to run more difficult projects, so you might want to have at least two options—a complicated project and an easier project—so that you can choose between them.

Now that you're mentally prepared for the task at hand, it's time to look at things you can do to make sure it all goes smoothly (or goes at all).

#### **USE CODE AND TEST REVIEWS AS A TEACHING TOOL**

If you're the technical lead on a small team (up to eight people), one of the best ways of teaching is instituting code reviews that also include test reviews. The idea is that as you review other people's code and tests, you teach them what you look for in the tests and your way of thinking about writing tests or approaching TDD. Here are some tips:

- Do the reviews in person, not through remote software. The personal connection lets much more information pass between you in nonverbal ways, so learning happens better and faster.
- In the first couple of weeks, review every line of code that gets checked in. This will help you avoid the “we didn't think this code needs reviewing” problem. If there's no red line at all (all code needs review), there's no moving it upward either, so no code is moved along.
- Add a third person to your code reviews, one who will sit on the side and learn how you review the code. This will allow them to later do code reviews themselves and teach others, so that you won't become a bottleneck for the team, as the only person capable of doing reviews. The idea is to develop others' ability to do code reviews and accept more responsibility.

If you want to learn more about this technique, I wrote about it in my blog for technical leaders, at <http://5whys.com/blog/what-should-a-good-code-review-look-and-feel-like.html>.

## 9.2 Ways to succeed

There are two main ways an organization or team can start changing a process: bottom up or top down (and sometimes both). The two ways are very different, as you'll see, and either could be the right approach for your team or company. There's no one right way.

As you proceed, you'll need to learn how to convince management that *your* efforts should also be *their* efforts, or when it would be wise to bring in someone from outside to help. Making progress visible is important, as is setting clear goals that can be measured. Identifying and avoiding obstacles should also be high on your list. There are many battles that can be fought, and you need to choose the right ones.

### 9.2.1 Guerrilla implementation (bottom up)

Guerrilla-style implementation is all about starting out with a team, getting results, and only then convincing other people that the practices are worthwhile. Usually the driver for guerrilla implementation is the team that's tired of doing things the prescribed way. They set out to do things differently; they study on their own and make changes happen. When the team shows results, other people in the organization may decide to start implementing similar changes in their own teams.

In some cases, guerrilla-style implementation is a process *adopted* first by developers and then by management. At other times, it's a process *advocated* first by developers and then by management. The difference is that you can accomplish the first covertly, without the higher powers knowing about it. The latter is done in conjunction with management.

It's up to you to figure out which approach will work better. Sometimes the only way to change things is by covert operations. Avoid this if you can, but if there's no other way and you're sure the change is needed, you can just do it.

Don't take this as a recommendation to make a career-limiting move. Developers do things they didn't ask permission for all the time: debugging code, reading email, writing code comments, creating flow diagrams, and so on. These are all tasks developers do as a regular part of the job. The same goes for unit testing. Most developers already write tests of some sort (automated or not). The idea is to redirect that time spent on tests into something that will provide benefits in the long term.

### 9.2.2 Convincing management (top down)

The top-down move usually starts in one of two ways. A manager or a developer will initiate the process and start the rest of the organization moving in that direction, piece by piece. Or a midlevel manager may see a presentation, read a book (such as this one), or talk to a colleague about the benefits of specific changes to the way they work. Such a manager will usually initiate the process by giving a presentation to people in other teams or even using their authority to make the change happen.

### 9.2.3 Getting an outside champion

I highly recommend getting an outside person to help with the change. An outside consultant coming in to help with unit testing and related matters has advantages over someone who works in the company:

- *Freedom to speak*—A consultant can say things that people inside the company may not be willing to hear from someone who works there (“The code integrity is bad,” “Your tests are unreadable,” and so on).
- *Experience*—A consultant will have more experience dealing with resistance from the inside, coming up with good answers to tough questions, and knowing which buttons to push to get things going.
- *Dedicated time*—For a consultant, this is their job. Unlike other employees in the company who have better things to do than push for change (like writing software), the consultant does this full time and is dedicated to this purpose.

#### Code integrity

*Code integrity* is a term I use to describe the purpose behind a team’s development activities, in terms of code stability, maintainability, and feedback. Mostly, it means that the code does what it’s meant to do, and the team knows when it doesn’t.

These practices are all part of code integrity:

- Automated builds
- Continuous integration
- Unit testing and test-driven development
- Code consistency and agreed standards for quality
- Achieving the shortest time possible to fix bugs (or make failing tests pass)


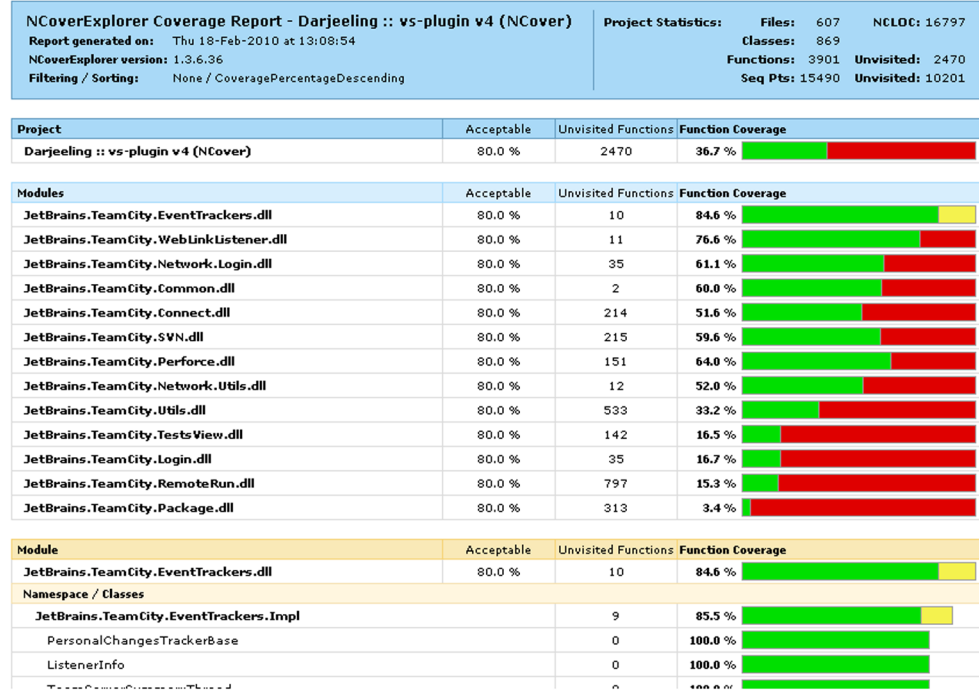
Some consider these to be values of development, and you can find them in methodologies such as Extreme Programming, but I like to say, “We have good code integrity,” instead of saying that I think we’re doing all these things well.

I’ve often seen a change break down because an overworked champion doesn’t have the time to dedicate to the process.

### 9.2.4 Making progress visible

It’s important to keep the progress and status of the change visible. Hang whiteboards or posters on walls in corridors or in the food-related areas where people congregate. The data displayed should be related to the goals you’re trying to achieve.

For example, show the number of passing or failing tests in the last nightly build. Keep a chart showing which teams are already running an automated build process. Put up a Scrum Burndown Chart of iteration progress or a test-code-coverage report (as shown in figure 9.1) if that’s what you have your goals set to. (You can learn more about Scrum at [www.controlchaos.com](http://www.controlchaos.com).) Put up contact details for yourself

Darjeeling > vs-plugin v4 (NCover) >  #5.0.119.0 (18 Feb 10 12:27)
[Overview](#)
[Changes \(1\)](#)
[Tests](#)
[Build Log](#)
[Build Parameters](#)
[Dependencies](#)
[Artifacts](#)
[Code Coverage](#)


**Figure 9.1** An example of a test-code-coverage report in TeamCity with NCover

and all the champions so someone can answer any questions that arise. Set up a big-screen LCD that's always showing, in big bold graphics, the status of the builds, what's currently running, and what's failing. Put that in a visible place where all developers can see, in a well-trafficked corridor, for example, or at the top of the team room main wall.

Your aim in using these charts is to connect with two groups:

- *The group undergoing the change*—People in this group will gain a greater feeling of accomplishment and pride as the charts (which are open to everyone) are updated, and they'll feel more compelled to complete the process because it's visible to others. They'll also be able to keep track of how they're doing compared to other groups. They may push harder knowing that another group implemented specific practices more quickly.
- *Those in the organization who aren't part of the process*—You're raising interest and curiosity among these people, triggering conversations and buzz, and creating a current that they can join if they choose.

### 9.2.5 Aiming for specific goals

Without goals, the change will be hard to measure and to communicate to others. It will be a vague “something” that can easily be shut down at the first sign of trouble.

Here are some goals you might want to consider:

- *Increase the amount of test-code coverage, parallel to code and test reviews.*

A study by Boris Beizer showed that developers who write tests and don't use code-coverage tools or other techniques to test code coverage will be naïvely optimistic about the coverage they gained from the tests. Another study, from *Peer Reviews in Software: A Practical Guide* (Addison-Wesley Professional, 2001) by Karl Wiegers, suggests that testing without code-coverage tools may result in coverage of only about 50% to 60% of the code. (There's much anecdotal evidence that by using TDD, you can get up to 95% or even 100% code coverage for logical code.)

A simple goal to measure is the percentage of the code covered by the tests. The more coverage, the better chance of finding bugs. It's not a silver bullet, though. You could easily have close to 100% code coverage with bad tests that don't mean anything. Low coverage is a bad sign; high coverage is a possible sign that things are better.

What you really want is high coverage along with continuous code and test reviews (as I explained earlier in this chapter). This way, you make sure tests aren't just written to satisfy coverage requirements (no asserts, for example) but are actually meaningful.

**NOTE** The study by Boris Beizer is discussed in Mark Johnson's article, “Dr. Boris Beizer on Software Testing: An Interview. Part I,” in *The Software QA Quarterly* (summer 1994). The other study is discussed in *Peer Reviews in Software: A Practical Guide*.

- *Increase the amount of test-code coverage relative to the amount of code churn.*

Some production systems will allow you to measure the amount of *code churn*—how many lines of code were changed between builds. The fewer lines of code changed, the fewer bugs you're likely to have introduced into a system. Calculating this isn't always practical, particularly in systems where you do a lot of code generation as part of the build process, but you can solve this by ignoring generated code. One system that allows you to measure code churn is Microsoft's Team System. (See Microsoft's “Analyze and Report on Code Churn and Code Coverage Using the Code Churn and Run Coverage Perspectives” at <http://msdn.microsoft.com/en-us/library/vstudio/ms244661.aspx>.)

- *Reduce the amount of bug reopening.*

It's easy to fix one thing and mistakenly break something else. If this doesn't happen often, it's a sign that you're able to fix things and maintain the system without breaking previous assumptions.



- *Reduce the average bug-fixing time (the time from bug opened to bug closed).*  
A system with good tests and coverage will usually allow you to fix things more quickly (assuming the tests are written in a maintainable manner). That, in turn, means better turnaround times and release cycles that are less stressful.

In *Code Complete* (Microsoft Press, 2nd edition, 2004), Steve McConnell outlines several metrics you can use to test progress. They include the following:

- The number of defects found per class by priority
- The number of defects per routine number of testing hours per bug found
- The average number of defects per test case

I highly recommend reading chapter 22 of McConnell's book, which deals with developer testing.

### 9.2.6 **Realizing that there will be hurdles**

There are always hurdles. Most will come from within the organizational structure, and some will be technical. The technical ones are easier to fix, because it's a matter of finding the right solution. The organizational ones need care and attention and a psychological approach.

It's important not to surrender to a feeling of temporary failure when an iteration goes bad, tests go slower than expected, and so on. It's sometimes hard to get going, and you'll need to persist for at least a couple of months to start feeling comfortable with the new process and to iron out all the kinks. Have management commit to continuing for at least three months even if things don't go as planned. It's important to get their agreement up front. You don't want to be running around trying to convince people in the middle of a stressful first month.

Also, absorb this short realization, shared by Tim Ottinger on Twitter (@Tottinge): "If your tests don't catch all defects, they still make it easier to fix the defects they didn't catch. It is a profound truth."

Now that we've looked at ways of ensuring things go right, let's look at some things that can lead to failure.

## 9.3 **Ways to fail**

In the preface to this book, I talked about one project I was involved with that failed, partly because unit testing wasn't implemented correctly. That's one way you can fail a project. I've listed several others here, along with one that cost me that project, and some things that can be done about them.

### 9.3.1 **Lack of a driving force**

In all the places where I've seen change fail, the lack of a driving force was the most powerful factor in play. Being a consistent driving force of change has its price. It will take time away from your normal job to teach others, help them, and wage internal political wars for change. You need to be willing to surrender the time you have for

these tasks, or the change won't happen. Bringing in an outside person, as mentioned in section 9.2.3, will help you in your quest for a consistent driving force.

### **9.3.2** *Lack of political support*

If your boss explicitly tells you not to make the change, there isn't a whole lot you can do, besides trying to convince management to see what you see. But sometimes the lack of support is much more subtle than that, and the trick is to realize that you're facing opposition.

For example, you may be told, "Sure, go ahead and implement those tests. We're adding 10% to your time to do this." Anything below 30% isn't realistic for beginning a unit testing effort. This is one way a manager may try to stop a trend—by choking it out of existence.

You need to recognize that you're facing opposition, but once you do, it's easy to identify. When you tell them that their limitations aren't realistic, you'll be told, "So don't do it."

### **9.3.3** *Bad implementations and first impressions*

If you're planning to implement unit testing without prior knowledge of how to write good unit tests, do yourself one big favor: involve someone who has experience, and follow best practices (such as those outlined in this book).

I've seen developers jump into the deep water without a proper understanding of what to do or where to start, and it's not a good place to be. Not only will it take a huge amount of time to learn how to make changes that are acceptable for your situation, but you'll also lose a lot of credibility along the way for starting out with a bad implementation. This can lead to the pilot project being shut down.

If you read this book's preface, you'll know that this happened to me. You have only a couple of months to get things up to speed and convince the higher-ups that you're achieving results. Make that time count, and remove any risks that you can. If you don't know how to write good tests, read a book or get a consultant. If you don't know how to make your code testable, do the same. Don't waste time reinventing testing methods.

### **9.3.4** *Lack of team support*

If your team doesn't support your efforts, it will be nearly impossible to succeed, because you'll have a hard time consolidating your extra work on the new process with your regular work. You should strive to have your team be part of the new process or at least not interfere with it.

Talk to your team members about the changes. Getting their support one by one is sometimes a good way to start, but talking to them as a group about your efforts—and answering their hard questions—can also prove valuable. Whatever you do, don't take the team's support for granted. Make sure you know what you're getting into; these are the people you have to work with on a daily basis.

Regardless of how you proceed, you're going to be asked tough questions about unit testing. The following questions and answers will help prepare you for your discussions with people who can make or break your agenda for change.

## 9.4 Influence factors

One of the things I find fascinating even more than unit tests is people and why they behave the way they do. It can be very frustrating to try to get someone to start doing something (like TDD, for example), and regardless of your best efforts, they just don't do it. You may have already tried reasoning with them, but you see they just don't do anything about your little talk.

A great book about the subject of influence is *Influencer: The Power to Change Anything* (McGraw-Hill, 2007) by Kerry Patterson, Joseph Grenny, David Maxfield, Ron McMillan, and Al Switzler. You can find a link to it at <http://5whys.com/recommended-books/>. The mantra of that book is a profound one: For every behavior that you see—the world is perfectly designed for that behavior to happen. That means that there are other factors except the person wanting to do something or being able to do it that influence their behavior. Yet, we rarely look beyond those two factors.

The book exposes us to six influence factors:

Personal ability	Does the person have all the skills or knowledge to perform what is required?
Personal motivation	Does the person take satisfaction from the right behavior or dislike the wrong behavior? Do they have the self-control to engage in the behavior when it's hardest to do so?
Social ability	Do you or others provide the help, information, and resources required by that person, particularly at critical times?
Social motivation	Are the people around them actively encouraging the right behavior and discouraging the wrong behavior? Are you or others modeling the right behavior in an effective way?
Structural (environmental) ability	Are there aspects in the environment (building, budget, and so on) that make the behavior convenient, easy, and safe? Are there enough cues and reminders to stay on course?
Structural motivation	Are there clear and meaningful rewards (such as pay, bonuses, or incentives) when you or others behave the right or wrong way? Do short-term rewards match the desired long-term results and behaviors you want to reinforce or want to avoid?

Consider this a short checklist to start understanding why things aren't going your way. Then consider another important fact: there might be more than one factor in play. For the behavior to change, you should change all the factors in play. If you change just one, the behavior won't change.

Here's an example of an imaginary checklist I've made about someone not performing TDD. (Keep in mind that this differs for each person in each organization.)

Personal ability	Does the person have all the skills or knowledge to perform what is required?	Yes. They went through a three-day TDD course with Roy Osherove.
Personal motivation	Does the person take satisfaction from the right behavior or dislike the wrong behavior? Do they have the self-control to engage in the behavior when it's hardest to do so?	I spoke with them and they like doing TDD.
Social ability	Do you or others provide the help, information, and resources required by that person, particularly at critical times?	Yes.
Social motivation	Are the people around them actively encouraging the right behavior and discouraging the wrong behavior? Are you or others modeling the right behavior in an effective way?	As much as possible.
Structural (environmental) ability	Are there aspects in the environment (building, budget, and so on) that make the behavior convenient, easy, and safe? Are there enough cues and reminders to stay on course?	* They don't have a budget for a build machine.
Structural motivation	Are there clear and meaningful rewards (such as pay, bonuses, or incentives) when you or others behave the right or wrong way? Do short-term rewards match the desired long-term results and behaviors you want to reinforce or want to avoid?	* When they try to spend time unit testing, their managers tell them they're wasting time. If they ship early and crappy, they get a bonus.

I put asterisks next to the items in the right column that require work. Here I've identified two issues that need to be resolved. Solving only the build machine budget problem won't change the behavior. They have to get a build machine *and* deter their managers from giving a bonus on shipping crappy stuff quickly.

I write much more on this stuff in *Notes to a Software Team Leader*, a book about running a technical team. You can find it at [5whys.com](http://5whys.com).

## 9.5 ***Tough questions and answers***

This section covers some questions I've come across in various places. They usually arise from the premise that implementing unit testing can hurt someone personally—a manager concerned about their deadlines or a QA employee concerned about their relevancy. Once you understand where a question is coming from, it's important to address the issue, directly or indirectly. Otherwise, there will always be subtle resistance.

### 9.5.1 ***How much time will unit testing add to the current process?***

Team leaders, project managers, and clients are the ones who usually ask how much time unit testing will add to the process. They're the people at the front lines in terms of timing.

Let's begin with some facts. Studies have shown that raising the overall code quality in a project can increase productivity and shorten schedules. How does this match up with the fact that writing tests makes coding slower? Through maintainability and the ease of fixing bugs, mostly.

**NOTE** For studies on code quality and productivity, see *Programming Productivity* (McGraw-Hill College, 1986) and *Software Assessments, Benchmarks, and Best Practices* (Addison-Wesley Professional, 2000). Both are by Capers Jones.

When asking about time, team leaders may really be asking, “What should I tell my project manager when we go way past our due date?” They may actually think the process is useful but are looking for ammunition for the upcoming battle. They may also be asking the question not in terms of the whole product but in terms of specific feature sets or functionality.

A project manager or customer who asks about timing, on the other hand, will usually be talking in terms of full product releases.

Because different people care about different scopes, your answers may vary. For example, unit testing can double the time it takes to implement a specific feature, but the overall release date for the product may actually be reduced. To understand this, let's look at a real example I was involved with.

#### A TALE OF TWO FEATURES

A large company I consulted with wanted to implement unit testing in their process, beginning with a pilot project. The pilot consisted of a group of developers adding a new feature to a large existing application. The company's main livelihood was in creating this large billing application and customizing parts of it for various clients. The company had thousands of developers around the world.

The following measures were taken to test the pilot's success:

- The time the team spent on each of the development stages
- The overall time for the project to be released to the client
- The number of bugs found by the client after the release

The same statistics were collected for a similar feature created by a different team for a different client. The two features were nearly the same size, and the teams were roughly at the same skill and experience level. Both tasks were customization efforts—one with unit tests, the other without. Table 9.1 shows the differences in time.

**Table 9.1 Team progress and output measured with and without tests**

Stage	Team without tests	Team with tests
Implementation (coding)	7 days	14 days
Integration	7 days	2 days

**Table 9.1 Team progress and output measured with and without tests (continued)**

Stage	Team without tests	Team with tests
Testing and bug fixing	Testing, 3 days Fixing, 3 days Testing, 3 days Fixing, 2 days Testing, 1 day Total: 12 days	Testing, 3 days Fixing, 1 day Testing, 1 day Fixing, 1 day Testing, 1 day Total: 9 days
Overall release time	26 days	24 days
Bugs found in production	71	11

Overall, the time to release with tests was less than without tests. Still, the managers on the team with unit tests didn't initially believe the pilot would be a success because they only looked at the implementation (coding) statistic (the first row in table 9.1) as the criteria for success, instead of the bottom line. It took twice the amount of time to code the feature (because unit tests require you to write more code). Despite this, the extra time was more than compensated for when the QA team found fewer bugs to deal with.

That's why it's important to emphasize that although unit testing can increase the amount of time it takes to implement a feature, the overall time requirements balance out over the product's release cycle because of increased quality and maintainability.

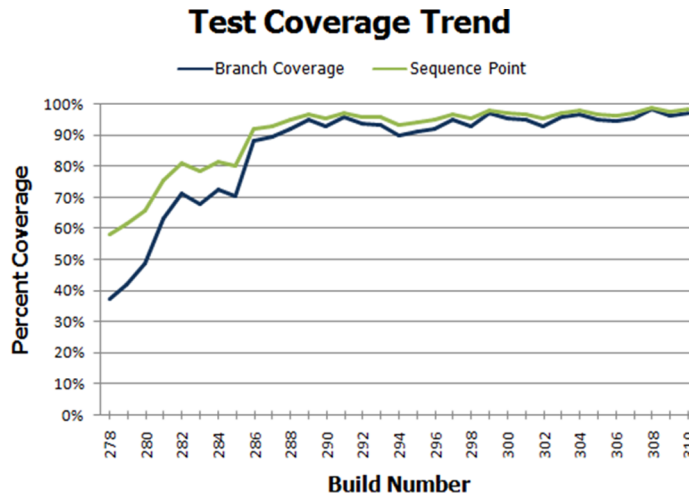
### 9.5.2 *Will my QA job be at risk because of unit testing?*

Unit testing doesn't eliminate QA-related jobs. QA engineers will receive the application with full unit test suites, which means they can make sure all the unit tests pass before they start their own testing process. Having unit tests in place will actually make their job more interesting. Instead of doing UI debugging (where every second button click results in an exception of some sort), they'll be able to focus on finding more logical (applicative) bugs in real-world scenarios. Unit tests provide the first layer of defense against bugs, and QA work provides the second layer—the user's acceptance layer. As with security, the application always needs to have more than one layer of protection. Allowing the QA process to focus on the larger issues can produce better applications.

In some places, QA engineers write code, and they can help write unit tests for the application. That happens in conjunction with the work of the application developers and not instead of it. Both developers and QA engineers can write unit tests.

### 9.5.3 *How do we know unit tests are actually working?*

To determine whether your unit testing is working, create a metric of some sort, as discussed in section 9.2.5. If you can measure it, you'll have a way to know; plus, you'll feel it.



**Figure 9.2** An example test-code-coverage trend report

Figure 9.2 shows a sample test-code-coverage report (coverage per build). Creating a report like this, by running a tool like NCover for .NET automatically during the build process, can demonstrate progress in one aspect of development.

Code coverage is a good starting point if you're wondering whether you're missing unit tests.

### 9.5.4 Is there proof that unit testing helps?

There aren't any specific studies unit tests on whether unit testing helps achieve better code quality that I can point to. Most related studies talk about adopting specific agile methods, with unit testing being just one of them. Some empirical evidence can be gleaned from the web, of companies and colleagues having great results and never wanting to go back to a code base without tests.

A few studies on TDD can be found at <http://biblio.gdinwiddie.com/biblio/Studies-OfTestDrivenDevelopment>.

### 9.5.5 Why is the QA department still finding bugs?

The job of a QA engineer is to find bugs at many different levels, attacking the application from many different approaches. Usually a QA engineer will perform integration-style testing, which can find problems that unit tests can't. For example, the way different components work together in production may point out bugs even though the individual components pass unit tests (which work well in isolation). In addition, a QA engineer may test things in terms of use cases or full scenarios that unit tests usually won't cover. That approach can discover logical bugs or acceptance-related bugs and is a great help to ensuring better project quality.

A study by Glenford Myre showed that developers writing tests were not really looking for bugs, and so they found only half to two-thirds of the bugs in an application. Broadly, that means there will always be jobs for QA engineers, no matter what. Although

that study is over 34 years old, I think the same mentality holds today, which makes the results still relevant, at least for me.

**NOTE** Glenford Myre's study is discussed in "A controlled experiment in program testing and code walkthroughs/inspections," in *Communications of the ACM* 21, no. 9 (September 1979), 760–69.

### 9.5.6 We have lots of code without tests: where do we start?

Studies conducted in the 1970s and 1990s showed that, typically, 90% of the bugs are found in 20% of the code. The trick is to find the code that has the most problems. More often than not, any team can tell you which components are the most problematic. Start there. You can always add some metrics, as discussed in section 9.2.5, relating to the number of bugs per class.

**NOTE** Studies that show 90% of the bugs being in 20% of the code include the following: Albert Endres, "An analysis of errors and their causes in system programs," *IEEE Transactions on Software Engineering* 2 (June 1975), 140–49; Lee L. Gremillion, "Determinants of program repair maintenance requirements," *Communications of the ACM* 27, no. 9 (August 1994), 926–32; Barry W. Boehm, "Industrial software metrics top 10 list," *IEEE Software* 4, no. 9 (September 1997), 94–95; and Shull and others, "What we have learned about fighting defects," *Proceedings of the 9th International Symposium on Software Metrics* (2002), 249–59.

Testing legacy code requires a different approach than when writing new code with tests. See chapter 10 for more details.

### 9.5.7 We work in several languages: is unit testing feasible?

Sometimes tests written in one language can test code written in other languages, especially if it's a .NET mix of languages. You can write tests in C# to test code written in VB.NET, for example. Sometimes each team writes tests in the language they develop in: C# developers can write tests in C# using one of the many frameworks available (MSTest, NUnit as first examples), and C++ developers can write tests using one of the C++-oriented frameworks, such as CppUnit. I've also seen solutions where people who write C++ code would write managed C++ wrappers around it and write tests in C# against those managed C++ wrappers, which made things easier to write and maintain.

### 9.5.8 What if we develop a combination of software and hardware?

If your application is made of a combination of software and hardware, you need to write tests for the software. Chances are you already have some sort of hardware simulator, and the tests you write can take advantage of this. It may take a little more work, but it's definitely possible, and companies do this all the time.



### 9.5.9 How can we know we don't have bugs in our tests?

You need to make sure your tests fail when they should and pass when they should. TDD is a great way to make sure you don't forget to check those things. See chapter 1 for a short walk-through of TDD.

### 9.5.10 My debugger shows that my code works; why do I need tests?

Debuggers don't help with multithreaded code much. Also, you may be sure your code works fine, but what about other people's code? How do you know it works? How do they know your code works and that they haven't broken anything when they make changes? Remember that coding is the first step in the life of the code. Most of its life, the code will be in maintenance mode. You need to make sure it will tell people when it breaks, using unit tests.

A study held by Curtis, Krasner, and Iscoe showed that most defects don't come from the code itself but result from miscommunication between people, requirements that keep changing, and a lack of application domain knowledge. Even if you're the world's greatest coder, chances are that if someone tells you to code the wrong thing, you'll do it. And when you need to change it, you'll be glad you have tests for everything else to make sure you don't break it.

**NOTE** The study by Bill Curtis, H. Krasner, and N. Iscoe is "A field study of the software design process for large systems," *Communications of the ACM* 31, no. 11 (November 1999), 1269–97.

### 9.5.11 Must we do TDD-style coding?

TDD is a style choice. I personally see a lot of value in TDD, and many people find it productive and beneficial, but others find that writing the tests after the code is good enough for them. You can make your own choice.

If this question arises from a fear of too much change happening at once, the learning process can be broken up into several intermediate steps:

- Learn unit testing from books such as this, and use tools such as Typemock Isolator or JMockIt so that you don't have to worry about design aspects while testing.
- Learn good design techniques, such as SOLID (which is discussed in chapter 11.).
- Learn to do test-driven development. (A good book is *Test-Driven Development: By Example*, by Kent Beck.)

This approach makes learning easier, and you can get started more quickly with less loss of time to the project.

## 9.6 Summary

Implementing unit testing in the organization is something that many readers of this book will have to face at one time or another. Be prepared. Make sure you have good

answers to the questions you're likely to be asked. Make sure that you don't alienate the people who can help you. Make sure you're ready for what could be an uphill battle. Understand the forces of influence.

In the next chapter, we'll take a look at legacy code and examine tools and techniques for working with it.