# Developing a Project Test Strategy

By Michael Kelly (www.MichaelDKelly.com)

**First published at www.informit.com**

## Why You Need a Test Strategy

I recently worked on a project in which I needed to develop a test strategy from the ground up. When I arrived on the project, the developers were attempting to follow a broken waterfall lifecycle. What that really meant was that development took place on a mostly ad hoc basis. The team had just doubled its number of developers (to about a dozen), and was in the process of exploring a more iterative approach with parallel development efforts. One novice tester, struggling to keep his head above water, provided the only testing for the project. At the same time that I arrived to help with testing, the team was hiring a new project manager and an architect, and was taking on very aggressive release commitments for the remainder of the year. Does this dismal situation sound familiar? I'm guessing it does, because this is not the first time I've encountered it.

The first order of business (besides learning where the nearest Coke machine was in relation to my cube) was to develop a test strategy. What is a test strategy? It depends on whom you ask. For the purposes of this article, we'll look at the test strategy as the objectives of all the test stages, test techniques, and test tools that apply to the project. Most importantly, the test strategy should help in facilitating the communication of the test process and its effects on the entire project team.

To guide our efforts in developing our test strategy, management was experiencing specific problems and was looking for some ways to solve them:

- Lack of test repeatability—the project was doing little to no regression testing
- Lack of test visibility—no metrics were gathered, and the only criterion for releasing code was the deadline
- Reactive build process—they did builds in response to project emergencies, without anticipating the needs of other build contributors
- No control over test environments or test data
- No unit or integration testing or code reviews after code moved to production
- No automation of simple processes or tests

This is the story of how we defined and implemented a test strategy for this project.

## Getting Started

When developing a test strategy, you need to get the key people in the project together, focus on the problems you're experiencing, and develop a long-term solution that you can implement over time. In addition to the bulleted list of problems above, our project's solution had to fulfill the basic requirement of a test strategy: *helping the project team to find the most important bugs as early as possible in the development lifecycle*. To find the most important defects early, the testing part of the project needs to align with the development part of the project, including the different testing phases, test types, project environments, how you promote code between environments, roles and responsibilities, and the common tools used.

Doesn't sound very easy, does it? But it's simpler than you might think!

## Keeping It Simple: Whiteboard Planning

A test strategy should be simple enough to fit on a whiteboard, and you should be able to explain what it means to anyone on the project team within a matter of minutes. This simplicity ensures that concepts are defined clearly and are simplified before you take the time to document them in more detail. The process of

putting things on a whiteboard is often participatory, and I find it's usually the best medium in helping people to share ideas. When people work on whiteboards, they draw nifty diagrams and flowcharts that everyone can understand.

When you develop the test strategy, you need to involve other people on the project. Often, project managers, development leads, architects, DBAs, and other key people on the project will have a better idea than you do of some of the technical resources available to them (tools or environments). In addition, your test strategy should cover the entire project lifecycle and everyone working on it. This means that you need the input of those technical people in order to succeed. At the very least, they can give you a more realistic idea of the types of testing that are taking place outside of the test team (unit testing, code reviews, runtime analysis, and so on). I typically try to find those people I perceive to be most involved on the project, and usher them into the room with me. Their insights and suggestions are often invaluable.

### Step 1: Outline the Basic Strategy

Once you get everyone together, start with a clean whiteboard. Put a column on the whiteboard for each aspect of testing that you want to capture or develop. For our project, I used a column for phases of testing (including types of testing performed on the project), the different code environments, and the measurements we would use to determine when to move the code between environments. It looked similar to the whiteboard in Figure 1.


**Figure 1 Whiteboard—strategy outline.**

Using a red marker, define what you currently do. List each phase of testing that the team currently executes consistently; under each phase, list the types of testing performed. You may find it helpful to clearly define for the group what each phase of testing represents and what happens there. There is no "right" definition; it's only important that everyone agrees with the definition you use. You may need to define the types of testing as well, but more importantly make sure that everyone understands how the types differ from each

other. Remember that you're facilitating the brainstorming for developing a test strategy; a clear framework requires clear definition.

### Step 2: Document the Current Setup

Next, list the different project environments and all the measurements currently used to move builds between those environments. For our project, I queried each person present and found that the team had been performing system testing and some regression testing, and we had a handful of users doing ad hoc acceptance testing. There was no consistency to unit testing and integration testing, and any code reviews typically happened after the code was delivered to the customer. In our system testing, we did mostly requirements verification, with some functional and lifecycle testing after that. On the previous iteration, the team performed one or two improvised latency tests, so we included those. All regression testing, when time allowed, was manual and based on test cases from previous releases.

We had five project environments. Each developer had his or her local environment, which they all then integrated into a common development environment. Once integrated, the project was built to a test environment for system testing. Then, based on requirements verification and the delivery date (key measurements), the code moved to the quality assurance (QA) environment. Once the users reviewed the most desired functionality for the release, there was a series of signoffs (another key measurement) and the code moved to production. All said and done, we had the test strategy shown in Figure 2. In the Measurements column, we discussed the level of requirements verification that took place in each environment and agreed that those numbers accurately reflected the current process.

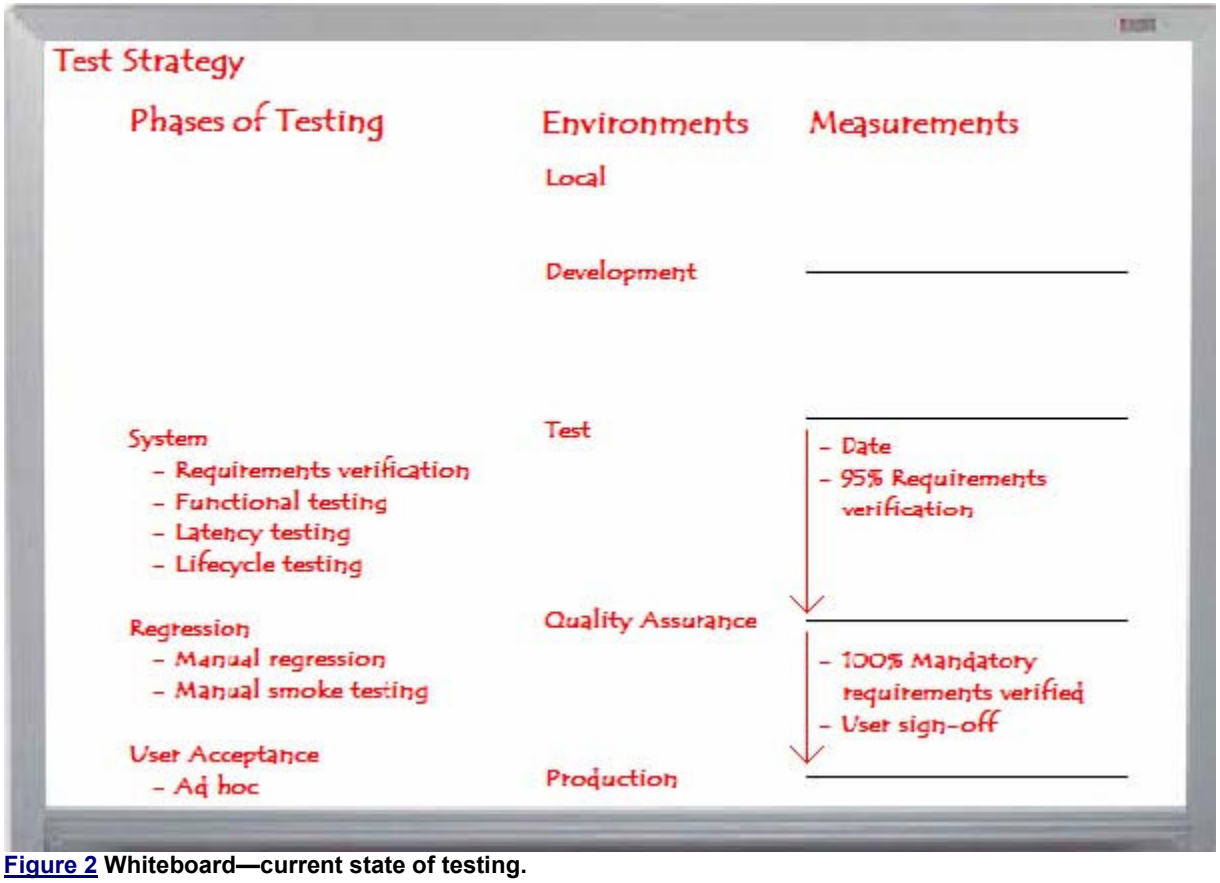


**Figure 2 Whiteboard—current state of testing.**

### Step 3: Brainstorm Improvements

Once everyone agreed that what we had on the whiteboard accurately reflected where we were, we started talking about what we wanted testing to achieve for the project. We talked about practices and tools that we thought would make us more effective and would match our current level of resources (both human and monetary). We knew that we probably couldn't expand the size of the testing team, for example, so we tried to focus our conversation around what the developers could do to help assist in testing. We also tried to focus on the key problems we were experiencing. (Remember that scary bulleted list I showed earlier?)

**TIP**

In a keynote at the Sixth IEEE International Workshop on Web Site Evolution, Hung Nguyen described his technique of taking a "bug centric" approach to developing a test strategy. His method looks at the problems found in production and works backward to create a strategy that will target these specific problems. He focuses on trying to add visibility to determine the cost and speed to production for each release. Whatever your context is, make sure that the group doing the brainstorming knows what problems they're working to solve. If you don't have a clear vision of what the test strategy needs to accomplish, step back and establish that vision first. The worst thing you can do is to put a strategy in place with the wrong objectives—that strategy is doomed to fail. It will create new problems or worsen existing problems. At best, it will end up solving some of the current problems accidentally, making the remaining problems appear more difficult to solve than they really are.

As we brainstormed, we agreed on a number of points and reached some conclusions:

- More structure and rigor in unit and integration testing would allow us to find more problems sooner, provide an initial level of automation, and give us some metrics with which we could better track the progress of development so we could make more informed decisions about when to move code. (Our application was built using mostly J2EE and Oracle, with some other technologies thrown in here and there. Both J2EE and Oracle have many robust and free tools that aid in unit and integration testing.)
- In systems testing, we concluded that with each release our user base would grow and that their demands would require increasingly more rigorous performance testing—something for which we had barely scratched the surface.
- We needed to move away from our dependence on requirements verification as our primary type of testing. While that was important, we were neglecting security testing, usability testing, configuration testing, data integrity testing, and hundreds of other types of tests.
- We decided to include some session-based exploratory testing that we would initially execute in pairs, until we felt more comfortable with the process and developed our rapid-learning and problem-solving skills. Once we felt more comfortable performing exploratory testing, we could begin executing more sessions on our own.
- We felt that we needed to establish a formal automated smoke test that we could use in all environments, along with a set of automated regression scripts to test high-risk functionality and high-volume transactions.
- We knew that our user acceptance testing (UAT) was not nearly as effective as it could be. So we committed to taking the time to develop a more detailed UAT test plan, along with some detailed test scripts for users to follow and more detailed training material to help bring them up to speed quickly. This is not to say that we wanted to take full responsibility for UAT, just that we wanted UAT to run more smoothly, which we could help by providing more guidelines, resources, and training.
- We agreed on metrics for when we should move code between environments. For both unit testing and integration testing, a 90% test pass rate would offer us enough confidence in the code to promote it, even knowing there were still some bugs to be worked out—as long as none of the bugs were showstoppers.
- We determined that we would be stricter in code reviews, ensuring that they would happen early in the process (preferably while the code was being written or very closely following) rather than after the release. Once we created our smoke tests, the code would have to pass those tests at 100% before progressing to the next level.
- In system testing, we agreed that we would not promote any critical or high-level defects, regardless of date, but that we would allow all other defects to be promoted and to let the user community decide whether they wanted the problems fixed now or later.

- We added code-coverage metrics as a tool that, along with defect trend analysis, could help us measure the effectiveness of our system test efforts.

Grabbing a different color of marker, I recorded our brainstorming session on the whiteboard (see Figure 3).

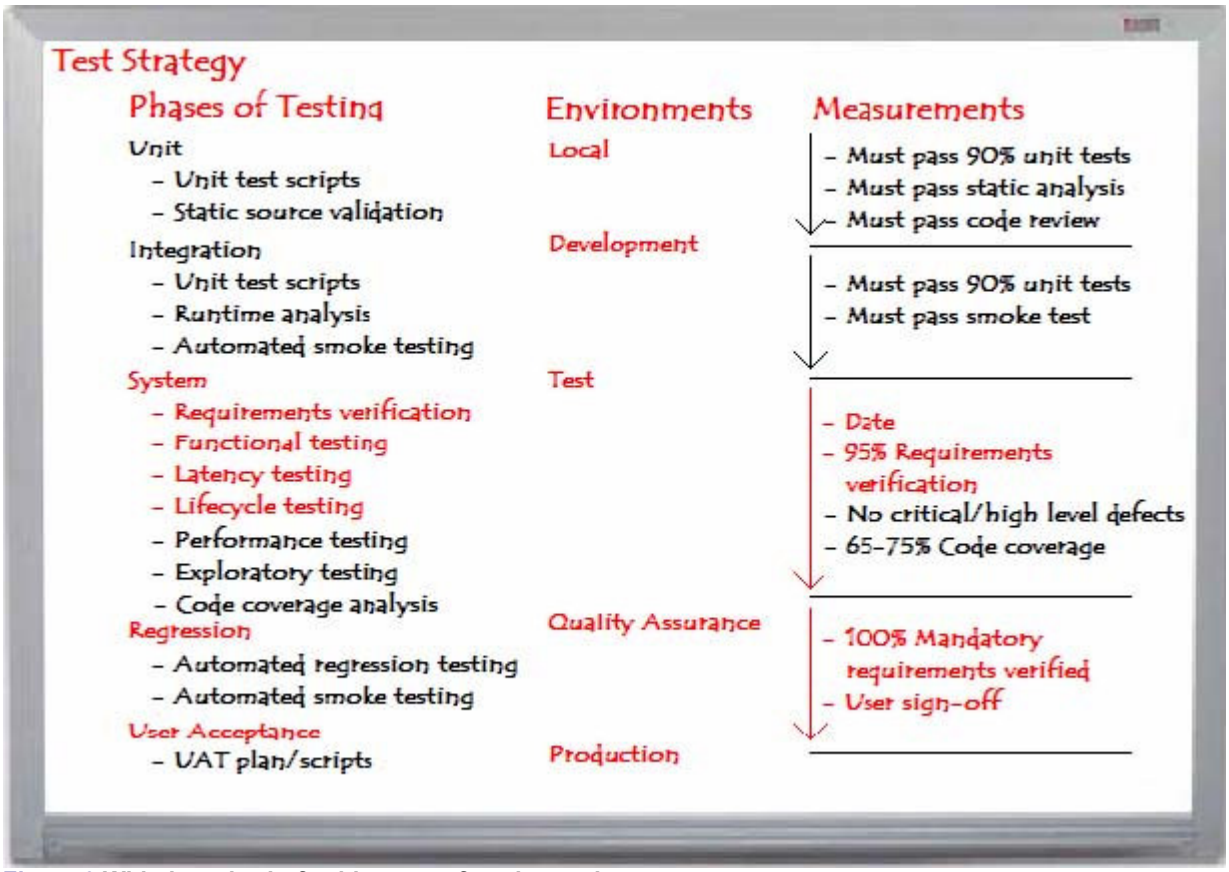


**Figure 3 Whiteboard—draft with types of testing and measurements.**

### Step 4: Organize the Plan

At this point, I asked everyone in the room to check that we all had agreement and felt as if we could succeed with this plan. Our next step was to establish who was responsible for what and where each of these activities would actually take place. Using yet another marker, we spent two or three minutes drawing brackets and arrows (see Figure 4).

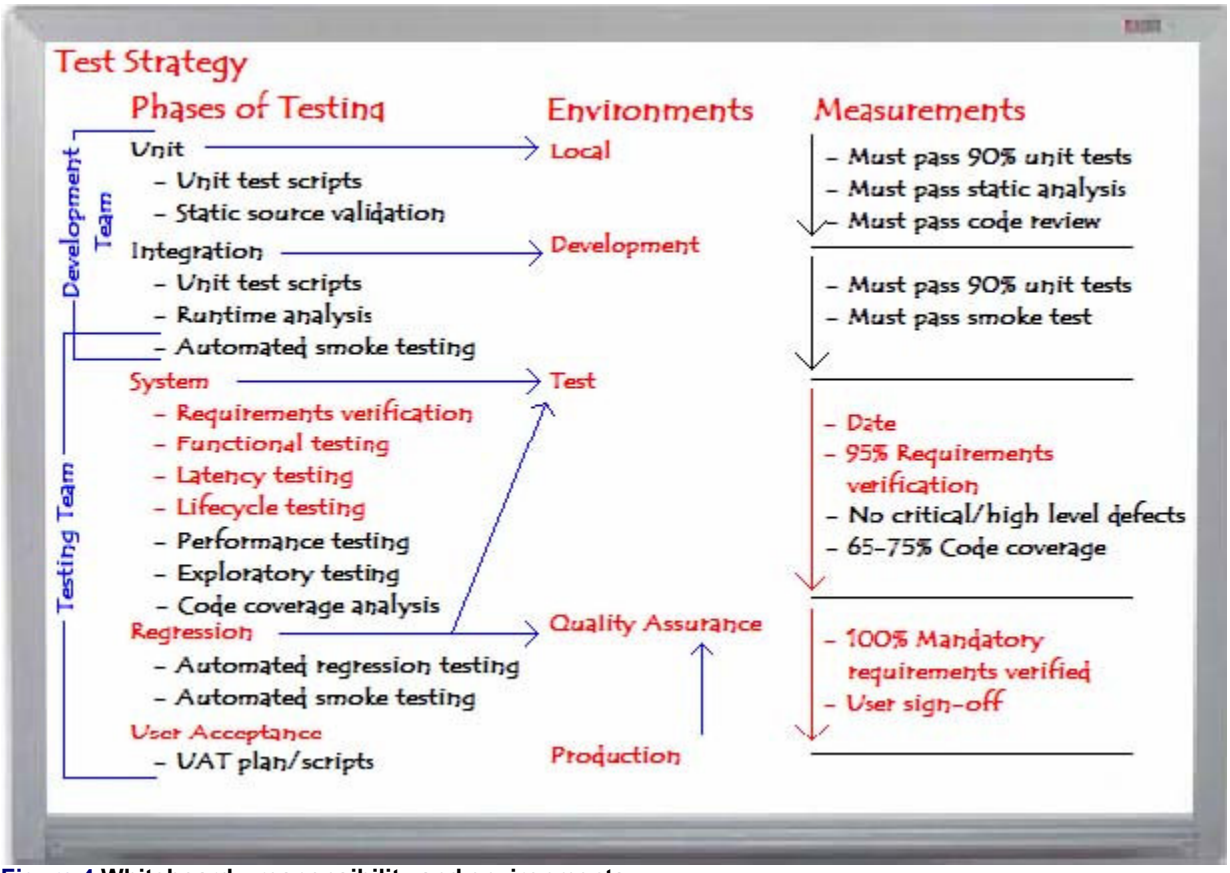


**Figure 4 Whiteboard—responsibility and environments.**

These groupings reflected the teams involved on our project. It's quite possible that you will have more teams involved on your project—multiple development or test teams, or even separate infrastructure teams or QA. The arrows we drew on our whiteboard represented the environments in which we would execute the types of testing identified. While not perfect, they gave us an outline for where we wanted to run most of the tests.

### Step 5: Organize the Tools

The final step in developing our test strategy was to document which tools we would be using to make all of this planning actually happen. The company already had a large investment in the IBM Rational products, so that factor influenced many of our decisions. We supplemented those tools with other tools where it made sense. For unit testing, for example, we chose JUnit because some of our developers already knew how to use it—and it's free and easy to learn. For static analysis, we chose Jlint. Rational filled in all of the other tool choices: ClearCase for source and test asset control, ClearQuest for issue tracking; Purify, Quantify, and PureCoverage for runtime analysis; Requisite Pro for requirements management; and Robot and TestManager for test automation. We talked about using some other tools for tasks such as runtime analysis and source control, but it just made sense for us to keep everything on a common platform for which we already had support. Our final whiteboard contained the information shown in Figure 5.

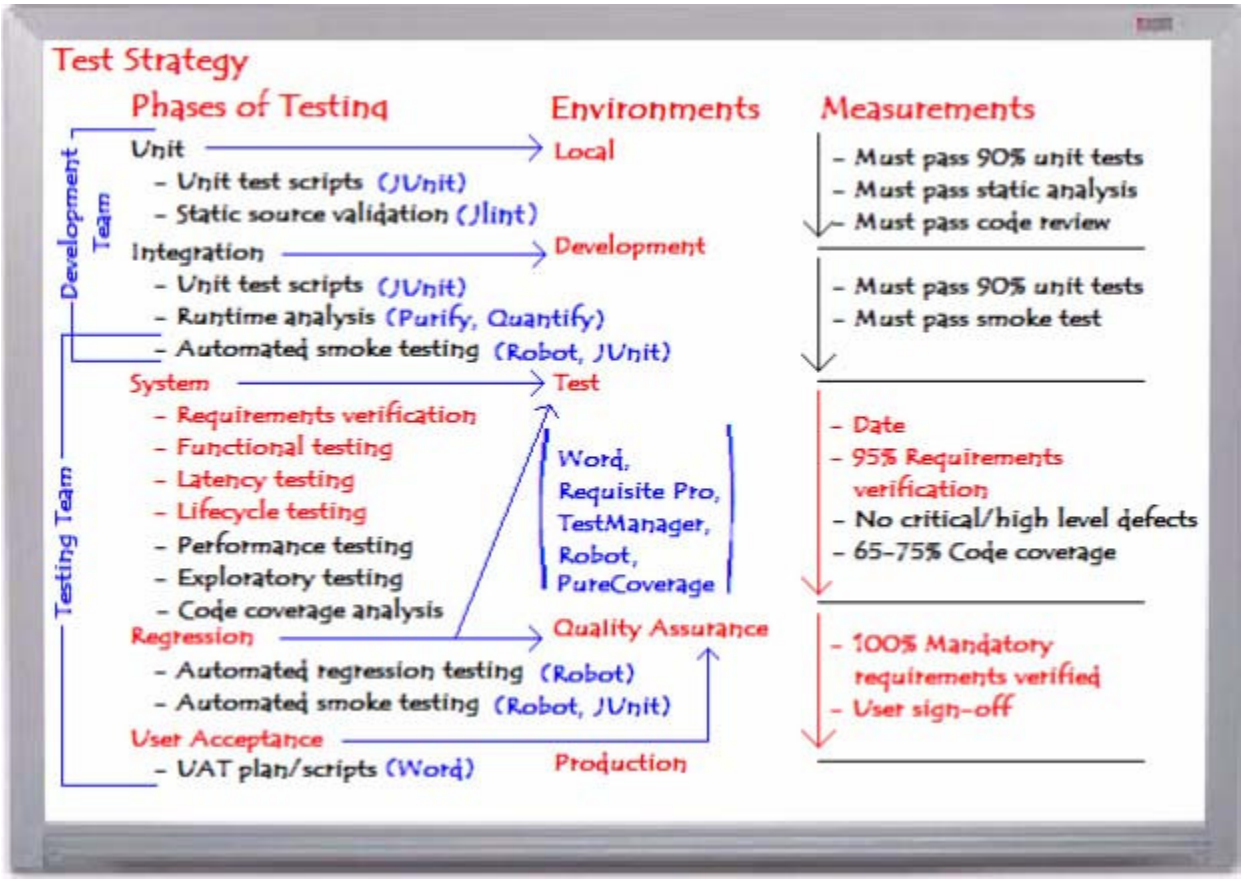


**Figure 5 Whiteboard—final strategy.**

That's it—hard part over. The next step is to implement it. Okay, maybe the hard part isn't actually over....

## Implementation

Now that you have a strategy, you need to share it with everyone on the project. Gather everyone around the whiteboard—or, better yet, use Visio to convert the whiteboard into a slide or series of slides. Ask everyone who developed the strategy to help explain the strategy, the reasoning behind it, and your plans for implementation. Having everyone participate helps give the feeling that this wasn't one person's pipedream, and will assist in getting buy-in from the whole team. Answer questions, and be open to changing the strategy based on feedback. Someone may know of a better tool, a more appropriate technique, or a more meaningful measurement than the ones you selected in your brainstorming session.

Once everyone (or almost everyone) agrees that this is an acceptable solution to the problems, develop an implementation plan. In the plan, answer questions like these:

- In what iteration will we include each new type of testing?
- How will we train the team to perform testing they have never done before?
- When will we install, configure, and train for each of the new tools?
- Who will be responsible for managing each phase of testing and ensuring that the established measurements are used?
- How do we plan to revise and update this strategy going forward?
- How will we measure whether this strategy is effective?
- Who is responsible for maintaining the strategy?

As you do this further thinking, you'll encounter other questions about implementation, based on your project context. Just make sure that you have the resources you need (human, hardware, and software); that you have the time and ability to train people to be effective at things that may be new to them; and that you start slow and build momentum over time.

The project discussed in this article still hasn't implemented everything in the test strategy yet. We found some changes to be more effective than others. We've evolved the strategy over time, but we still focus on incorporating a new tool or technique each iteration, or we focus on training people to be more effective at the tasks we're already performing. Because our strategy is so simple and in a format that we can easily change and update, we find it both flexible and helpful as we try to write better software.

## References

For more information on developing your own test strategy, you might find the following resources helpful:


- Lessons Learned in Software Testing: A Context-Driven Approach (Wiley, 2001), by Cem Kaner et al., has an entire chapter dedicated to planning a test strategy.
- Software Testing: A Guide to the TMap Approach (Addison-Wesley, 2001, ISBN 0201745712), by Martin Pol et al., also has a chapter on the topic.
- "Test Strategy and Test Plan," by Jeff Nyman, is an excellent resource for more information on creating a test strategy.
- Try the Heuristic Test Strategy Model (PDF), by James Bach.


For more information on the tools we used, look here:

- Jlint
- JUnit
- IBM Rational

## Michael Kelly


Michael Kelly is a Senior SQA Specialist for CTI Group with experience in software development and testing. Mike has published numerous articles on topics software testing and has presented at several conferences on software testing. Mike is currently serving as the Program Director for the Indianapolis Quality Assurance Association and the Membership Chair for the Association for Software Testing. You can reach Mike by email at Mike@MichaelDKelly.com.