LESSONS IN TEST a-u-t-o-m-a-t-i-o-n

A manager's guide to avoiding pitfalls when automating testing

by Elfriede Dustin

I have worked on many projects at various companies where automated testing tools were introduced to a test program lifecycle for the first time. In reviewing these projects, I have accumulated a list of "Automated Testing Lessons Learned," taken from actual experiences and test engineer feedback. In this article I will share examples of this feedback compiled from real proj-ects, hoping that this information will help you avoid some typical false starts and roadblocks.

QUICK LOOK

- Problems encountered in real-world projects
- Suggested corrective actions to take

LESSONS LEARNED

The various tools used throughout the development lifecycle did not easily integrate.

A different tool was used for each phase of the lifecycle: a business modeling tool during the business analysis phase, a requirements management tool during the requirements phase, a design tool during the design phase, a test management tool during the testing phase, and so on. For metrics purposes—and to enforce consistency among the elements and traceability between phases—the goal was to have the output of one tool feed into the next tool used in the lifecycle. But because each of the tools used for the various phases was from a different vendor, they didn't easily integrate. Trying to overcome those challenges and integrate the tools on this project was a complex effort. Much time was spent trying to move information from one tool set to another, using elaborate programming techniques, and resulting in extra work. The code generated to make those tools integrate was not reusable later, because of new upgrades to the various tools.

One possible corrective action would be for each project team to conduct a feasibility study to measure the need to purchase tools that are already integrated into a unified suite. A cost/benefit analysis should be conducted to determine whether the potential benefits of buying an integrated suite of tools would outweigh the costs.

Duplicate information was kept in multiple repositories.

One of our project teams purchased a test management tool in addition to the already existing requirements management and automated testing tools. Duplicate information was kept in multiple repositories and was very difficult to maintain. In several instances, the implementation of more tools actually resulted in *less* productivity.

I have found that a requirements management tool

can be used as a test management tool. There is no need to maintain test information in both tool databases. Maintenance can be improved by simply keeping most of the test progress and test status information in one tool.

The automated testing tool drove the testing effort.

Often when a new tool is used for the first time on a testing program, more time is spent on automating test scripts than on actual *testing*. Test engineers may be eager to automate elaborate scripts, but may lose sight of the real goal, which is to test the application.

Keep in mind that automating test scripts is *part* of the testing effort, but does not replace the testing effort. Not everything can or should be automated. It is important to evaluate which tests lend themselves to automation. For example, only automate tests that are run many times, such as "smoke" (build verification) tests, regression tests, and mundane tests (tests that include many simple and repetitive steps). Also, automate tests that would be impossible (or prohibitively expensive) to perform manually, e.g., simulating 1,000 multi-user accesses.

Everyone on the testing staff was busy trying to automate scripts.

On some projects we found that the division of labor—

PERSPECTIVE

Making Your Tool Pay Off

Testing Manager Clive Bates had worked for the venerable London bank for eighteen years, and knew that within its hallowed vaults lay a wealth of assorted treasures. But if you made a left at International Acquisitions, walked straight through the IT offices, and into Storage Room B, you'd find an artifact of a different sort: in a dusty box on the third shelf lay the earthly remains of a management decision to automate testing.

It had seemed like a good idea at the time, recalls Bates. Buying the GUI test automation tool had been considered crucial in meeting one of the bank's priorities: getting a Windows-based Electronic Banking Application into the hands of its international corporate customers. If the project were successful, it would be the company's primary platform for moving into the lucrative world of dial-up banking.

But once the tool was purchased and staff were sent off to the suppliers' training courses, plans started to unravel. "It became very apparent that more preparation time was needed than Management had budgeted for," says Bates. The tool was pronounced more trouble than it was worth, relegated to Storage Room B, and the team went back to old-fashioned manual testing. "The testing team leader at the time was then very proud to have over 4000 manual scripts to test their application out," Bates remembers—but that would soon prove unwieldy.

"I think that initially everyone had great expectations after the vendors' hype," Bates says, "but had underestimated the amount of preparation and planning and training." It was a double whammy. Not only had the tool been brought in too late, but after the vendor's enthusiastic presentations, Management had underestimated the amount of time needed to train non-programming staff to use the tool.

Months into the manual testing, Management realized the next round of regression testing would be beyond the scope of mere humans, and called upon Bates to set up a new team.

The tool had been on the shelf so long that it was now out-of-date and needed upgrading. "But the real first step was to sit down and plan what we wanted to use the tool *for*," Bates says. He had some manual scripts for the application's first release, and used these as a guide to set down the structure of the scripting his team was going to do. "So instead of just picking the scripts up and recording," says Bates, "we broke them down into a hierarchy so we could use them as a template."

Training was the next step; by now, the employees who had sat through hours of tool training had moved on to other projects, so an entirely new investment in staff education had to be budgeted. To supplement in-house knowledge, Bates brought in vendor consultants in four-hour increments throughout the project, just to help the team check on its status and keep on the straight and narrow.

This time around, Management kept a stiff upper lip and agreed to devote the necessary resources to the project. To maintain the original release schedules, Bates was allowed to build a large team. "With 4000 scripts to convert, and lots of new modules coming in," he says, "they knew that we'd be up to our necks in regression testing…and they were very aware that this was a key product to our organization."

To keep on top of the schedule, a parallel effort (manual and automated) was maintained until the tool testing could take over. With a central repository, a dedicated test bed, and a designated in-house tool expert on-call around the clock, Bates' teams were able to release the software on time. "That was my first experience with testing tools," says Bates, and it taught him a few things. First, that it's vital to invest some of your best resources in the setup and planning stages. Be realistic about your automation goals and write your scripts with the tool in mind so you can record them easily and avoid duplication. Calculate when payback (ROI) is expected, and be realistic; this is essential for justification in a business plan.

Bates also recommends budgeting ahead for ongoing maintenance. "An automated test tool can't be put on the shelf with no use and no updates, and then be expected to work months later," he says. Part of that budgeting process is convincing Management to allot the time and resources to do it right. "And just as importantly," stresses Bates, "you have to realize that even the best tools aren't magic."

—A.W.

breaking up responsibilities so that all required testing activities are accomplished—had not been adequately defined. As a result, the entire team focused on the development of automated testing scripts.

It's important to clearly define this division of duties. It is not necessary for the entire testing team to spend its time automating scripts; only a portion of the test engineers who have a development background should spend their time automating scripts. Manual test engineer expertise is still necessary to focus on all other test aspects. Again, let me stress that it is not feasible to automate *everything*.

Elaborate test scripts were developed, duplicating the development effort.

I have witnessed test script development that resulted in an almost complete duplication of the development effort, through overuse of the testing tool's programming language. In one of our projects, the application itself used a complex algorithm to calculate various interest rates and mortgage rates. The tester recreated these algorithms using the testing tool. Too much time was spent on automating scripts, without much additional value gained. One cumbersome script was developed using the tool's programming language—but the same script could have been developed using the capture/playback feature of the tool and simply modifying the generated script in a fraction of time. The test team must be careful not to duplicate the development effort; this is a risk when developing elaborate test scripts. For each automated testing program it is important to conduct an automation analysis, and to determine the best approach to automation by estimating the highest return.

Automated test script creation was cumbersome.

It's important that all teams involved understand that test script automation doesn't happen automatically, no matter what the vendor claims. On one project, test engineers with manual test backgrounds were involved in creating the automated scripts. Basing their assumptions on the vendor claims of the tool's ease of use, the test engineers complained that the creation of automated scripts took longer than expected, and that too many workaround solutions had to be found.

It's important to understand that the tools are never as easy to use as the tool vendor claims. It is also beneficial to include one person on the testing staff who has programming knowledge and appropriate tool training, so that she can mentor the rest of the testing staff responsible for automation.

Training was too late in the process, so test engineers lacked tool knowledge.

Sometimes tool training is initiated too late in the project for it to be useful for the test engineers using the tool. On one of our projects this resulted in tools not being used correctly. Often, for example, only the capture/playback portion of the testing tool was used, and scripts had to be repeatedly recreated, causing much frustration.

When introducing an automated tool to a new project, it's important that tool training be incorporated early in the schedule as one of the important milestones. Since testing needs to be involved throughout the system development lifecycle, tool training should happen early in the cycle for it to be useful—and to ensure that tool issues can be brought up and resolved early. This involvement allows for testability and automation capabilities to be built into the system-under-test.

Mentors are also very important when first introducing tools to the testing program. Mentors must be very knowledgeable and should advise, but shouldn't do the actual work.

The test tool was introduced to the testing program with two weeks left for system testing.

I can recall one project in which system testing was behind schedule, and Management introduced a new testing tool in the hopes of speeding up the testing effort. Since we had a test automation expert on the team, we were able to leverage the use of the testing tool for such efforts as creating a smoke test script. The smoke test script automated the major functionality of the system; and before a new system test build was accepted, the smoke test script was played back to verify that previously working functionality had not been affected by new fixes.

We had taken a risk by introducing the tool so late in the process, but in this case we came out ahead: the script allowed for some timesaving. If no test automation expert had been on the test team, I would have suggested that the test team not accept the use of an automated testing tool this late in the lifecycle. The tool's learning curve would not have allowed us to gain any benefits from incorporating it this late in the testing lifecycle.

Testers resisted the tool.

The best automation tool in the world won't help your test efforts if your team resists using it. In one case, the tool remained in the box—hardly any effort was invested in incorporating it into the process. The test engineers felt that their manual process worked fine, and they didn't want to bother with the additional setup work involved in introducing this tool.

When first introducing a new tool to the testing program, mentors are very important, but you also need tool champions—advocates of the tool. These are team members who have experience with the tool, and who have first-hand experience in its successful implementation.

There were expectations of early payback.

Often when a new tool is introduced to a project, the expectations for the return on investment are very high. Project members anticipate that the tool will immediately narrow down the testing scope, meaning reducing cost and schedule. In reality, chances are that initially the tool will actually *increase* the testing scope.

It is therefore very important to manage expectations. An automated testing tool does not replace manual testing, nor does it replace the test engineer. Initially, the test effort will increase, but when automation is done correctly it will decrease on subsequent releases.

The tool had problems recognizing third-party controls (widgets).

Another aspect of managing expectations is understanding the tool's capabilities. Is it compatible with the system-under-test? On some projects the test engineers were surprised to find out that a specific tool could not be used for some parts of the application. During the tool evaluation period it is important to verify that third-party controls (widgets) used in the system-under-test are compatible with the automated testing tool's capabilities.

If a testing tool is already in-house, but the system architecture has not been developed yet, the test engineer can give the developers a list of compatible third-party controls that are supported by the test tool vendor. If an incompatible third-party control is proposed, the test engineer should require a justification and explain the consequences.

A lack of test development guidelines was noted.

One program had several test engineers, each using a different style for creating test scripts. Maintaining the scripts was a nightmare. Script readability and maintainability is greatly increased when test engineers can rely on development guidelines.

The tool was intrusive, but the development staff wasn't informed of this problem until late in the testing lifecycle.

Some testing tools are intrusive—actual code has to be inserted into the code developed for the system-under-test in order for the automated tool to work correctly. In this case, the development staff wasn't informed that the automated tool was intrusive; when they finally found out, they were very reluctant to incorporate the necessary changes into their code. Because of uncertainty about the tool's intrusiveness, the first time that the system-under-test didn't function as expected the intrusive tool was immediately blamed (even though there was no evidence that it had been the culprit).

In order to prevent this from happening, the test engineers need to involve the development staff when selecting an automated tool. Developers need to know well in advance that the tool requires code additions (if applicable—not all tools are intrusive). Developers can be assured that the tool will not cause any problems by offering them feedback from other companies who have experience using the tool, and by showing documented vendor claims to that effect.

Reports produced by the tool were useless.

The test engineering staff on one project spent much time setting up elaborate customized reports using Crystal Report Writer, which was part of the automated testing tool. The reports were never used, since the data required for the report was never accumulated in the tool.

Before creating any elaborate reports, verify that the specific type of data is actually collected. Set up only reports specific to the data that will be generated. Produce reports as requested by Management or customers, and those reports required internally by the test program for measuring test progress and test status.

Tools were selected and purchased before a system engineering environment was defined.

Some teams are eager to bring in automated tools. But there's such a thing as too much eagerness: tools that are evaluated and purchased without having a system architecture in place can cause problems. When the decision for the architecture is being made, many compatibility issues can surface between the tools already purchased and the suggested architecture. I remember projects in which workaround solutions had to be found while trying to match the system-engineering environment to the requirements of the tools already purchased. A lot of vendor inquiries had to be made to determine whether the next release of the tools might be compatible with the system middle-layer that the project team wanted to choose.

To avoid these problems, it's important that a system architecture be defined—and test tools selected—with all requirements (tool and architecture) in mind.

Various tool versions were in use.

It's possible for everyone to be using the same tool, and yet still not be able to talk to each other. On one project that had numerous tool licenses, we had various tool versions in use. That meant that scripts created in one version of the tool were not compatible in another version, causing significant compatibility problems and requiring many workaround solutions.

One way to prevent this from happening is to ensure that tool upgrades are centralized and managed by a Configuration Management department.

The new tool upgrade wasn't compatible with the existing system engineering environment.

Keep on top of product "improvements." On one project, a new tool version involved an upgrade that used the MS-Mail system for automatic defect notification to the MSExchange system. The vendor had omitted mentioning this upgrade detail. The project had invested a lot of money in a variety of test tool licenses and was caught by surprise by this new tool upgrade. The project's company actually had plans to move to the Lotus Notes mailing system, but was still using MSMail. And using the Microsoft Exchange system, unfortunately, was not part of their plans at all. An elaborate workaround solution had to be found to allow for the tool's compatibility with Lotus Notes.

Whenever a new tool upgrade is received, verify any major changes with the vendor. It's often to your benefit

PERSPECTIVE

Recognizing Your Options

Testing tools sometimes *don't* recognize third-party controls. What do you do? Pronounce it an impasse, and discard the tool? Insist the developers discard the widget in your system that's creating problems? Or do you find a workaround? Paul Herzlich is a strong believer in using a diplomatic approach—that it's often better to find a way for the test tool to adapt to the difficult situation than to call off negotiations entirely.

The London-based test manager has seen a myriad of variations on the basic theme of incompatibility; sometimes it's the tool not being able to recognize the third-party control, and sometimes it's simply unusual implementations that trip up the test tool. If the test tool isn't directly compatible with every aspect of the system you're developing, does that reduce the overall testability? "It probably does," Herzlich concedes. "But that doesn't make the software untestable—or at least it shouldn't."

Be careful, he stresses, in limiting your developers' use of certain third-party elements based on tool capabilities—while tools are constantly being updated, they're never going to specifically support every new application configuration. "If Development is writing a new system, they're probably using features the vendors haven't even thought of yet," says Herzlich. You're not going to be helping Development find solutions if your only advice to them is "Don't use that, don't do that, the tool hasn't caught up yet."

Herzlich recalls one project that included an embedded spreadsheet product that wasn't supported by the team's test tool. "But with our schedule," he says, "there was no question of us going back to the developers and saying 'don't use that feature,' because it was a key part of the product." And switching tools wasn't an answer either; none of the testing tools at the time accurately or completely supported it. "We could have approached it by writing our own test routines in *C*," says Herzlich, "but we decided to use the features that WERE there in the tool's scripting language to program our own support for the third-party component ."

If all else fails, says Herzlich, any good testware should give you two basic tools that allow you test almost anything: bitmap recognition and text recognition. "You need a tool that allows you to program, in script language, around the incompatibility—and you need the programming skills to do that." While it's great to find that perfect match between test tool and application, the measure of a test tool, Herzlich says, is how many features it gives you to allow you to work around situations that are LESS than ideal. *—A.W.*

to become a beta testing site for any new tool upgrades; this is a good way to incorporate your project's issues or needs regarding any new upgrade.

It is also advisable to test a new tool upgrade in an isolated environment to verify its compatibility with the currently existing system engineering environment, before rolling out any new tool upgrade on a project.

The tool's database didn't allow for scalability.

One project implemented a tool that used Access as its database. Once the test bed grew (test scripts, test requirements, etc.) and multiple test engineers tried to access the database, multi-user access problems surfaced. The database became corrupted several times and backups had to be restored.

To avoid this problem, pick a tool that allows for scalability using a robust database. Additionally, make sure to back up your testing tool database daily, if not twice a day.

Incorrect use of a test tool's management functionality results in wasted time.

In one specific test tool that allows for test management, test requirements can be entered into the tool in a hierarchical fashion. On one project, a person spent two weeks to enter test requirements in this hierarchy—ending up with 1,600 test requirements. When the test team was then ready to automate these test requirements, they realized that there was not enough time or resources to automate each of the 1,600 requirements. Much of the time spent entering these requirements was wasted, since only a fraction of the test requirements would eventually be automated. Planning and laying out milestones (what, when, and how something will be accomplished) still applies to automated testing.

Solving Problems

These are just some of the various lessons I've learned, gathered from years working on various automated testing programs. But it's not an exhaustive list. How should you deal with the problems you'll encounter? In my book, *Automated Software Testing*, my co-authors and I discuss the following test improvement process:

The test team needs to adopt, as part of its culture, an ongoing iterative process focusing on lessons learned. Such a program encourages test engineers to raise corrective action proposals *immediately*, when such actions could potentially have a significant effect on test program performance. Test managers, meanwhile, need to promote such leadership behavior from each test engineer.

The metrics that are collected throughout the test lifecycle—and especially during the test execution phase—will help pinpoint problems that need to be addressed. Consequently, the test team should periodically

"take a pulse" on the quality of the test effort. This will help improve test results, whenever necessary, by altering the specific ways that the test team performs its business. Not only should the test team concentrate on lessons learned for the test lifecycle, but it should also point out issues pertaining to the system development lifecycle.

Lessons you have learned, metrics evaluations, and any corresponding improvement activity or corrective actions need to be documented throughout the entire test process in an easily accessible central repository. A test team intranet site can prove very effective in maintaining such documentation. For example, lessons learned could be documented using a requirements management tool database, such as DOORS. Keeping an up-to-date database of important lessons learned provides all individuals on the project team with the ability to monitor the progress and status of issues throughout the project. A test team intranet can also make lessons learned available to all software professionals within the organization.

The test manager needs to ensure that records of lessons learned are viewed as improvement opportunities. The records, for example, should not list the names of the individuals involved in the particular activity. Each record should include at least one corrective action that suggests how the process or process output can be improved.

When lessons learned and metrics evaluation sessions take place at the end of a system development lifecycle, it is too late to take any corrective action for the current project. Nevertheless, lessons learned that are recorded at this stage can help to improve future test efforts; so it's still worthwhile for the test team to document lessons learned, even at this late stage. The time and energy you invest in recording what you've learned will pay off on your next project. STQE

Elfriede Dustin is a test manager at Computer Sciences Corporation and has recently published a book titled Automated Software Testing. *She is also president of the Washington, DC area Rational Testing User Group. To contact her or to find out more about the book or the Rational Testing User Group, see* http://www.autotestco.com.