Improving Software Quality: Nine Best-Practices for Test Automation

The double-edged sword of "go-to-market quickly" with "as few resources as possible" causes many software development teams to cut corners when it comes to automated testing. It's certainly not a case of failing to understand the value of automated testing; but in today's frenzied business climate—where upper management pushes to do more, faster at a lower cost—it's easy to lose sight of the basics.

But at the same time, ignoring the basics of software test automation is likely to create even bigger problems. Dissatisfied end users lead to lower revenues while bugs that go undiscovered until late in the coding process will cause development costs to soar.

The mission to improve software quality usually hinges on streamlining the development process, which depends heavily on testing efficiency. To create streamlined automate testing, it's key to design a plan prior developing the software code. Without careful test planning, more bugs occur during development because programmers can't think of everything to test for as they write the code—they're always going to miss something.

To assist with the test automation planning process, Intervise has developed the following nine best-practices for test automation that can be applied by teams utilizing both the Waterfall and Agile development methodologies. Intervise has refined and enhanced these best practices over the course of 20+ years of providing application testing services to federal and commercial clients.

Best Practice #1: Plan for Test Automation as Early as Possible

Successful test automation requires dedicated machine resources and special software that is not needed for manual test execution. If you don't plan for the expense and acquisition time early in the planning process, it could hinder your chances for success while making cost and schedule overruns more likely. This is especially true if you are considering performance, load and stress testing: the required hardware and software tools tend to be expensive.

During the planning phases of the project, take the time to develop and articulate the *test strategy*, also known as the test approach or high-level test plan. This is the means by which you factor the testing requirements into the planning process.

The test strategy should encompass all forms of manual and automated testing. From a test automation point-of-view, the strategy should include the following aspects:

- The types of tests to be automated
- Test equipment required
- Required software tools and the license costs
- Data management implications
- Estimates for sizing, cost and staffing

The test strategy provides a useful vehicle for the critical review of the test-effort scope:

- The important things to test (you can't test everything)
- The cost of testing (some tests are expensive relative to others)
- The tradeoffs and implications of testing decisions

Stakeholder review of the test strategy is an important part of the review process because it is critical to have everyone on the same page due to the cost and schedule implications. It is normal to have widely-diverging opinions that are strenuously argued, but the time spent is a worthwhile investment.

Best Practice #2: Plan to Automate Tests for Both Functional and Non-Functional Requirements

System requirements generally fall into one of two categories: *Functional requirements* describe the specific things the system will do, the results it will produce, or the functions/features that the system will provide. *Non-functional requirements* describe how, to what extent, or to what degree the system will fulfill its functional requirements.

Tests for non-functional requirements (sometimes called *characterization tests*) often must be automated because many runs may be required to determine which system configurations give optimum performance. These are often more expensive to create, execute and maintain than tests for functional requirements. The reason is rooted in the number of environmental factors you must control to generate meaningful results from the tests. The set of conditions (e.g. the load on the system, number of users, amount of free memory) can be difficult to control.

Tests for functional requirements, on the other hand, tend to be pass/fail in nature—the Application Under Test (AUT) either gives the expected answer or it doesn't.

If you plan to automate both functional and non-functional tests, you should think about the design of the tests at the same time in order to take advantage of the building-block approach inherent in architectures such as Keyword-Based Test Design. Considering both functional and non-functional requirements during the test-design process will often yield opportunities to share the test-automation code or design code that can be used in both kinds of test suites.

Best Practice #3: Establish a Test Automation Architecture

The software testing landscape is littered with the bones of automation projects that started with high hopes—only to see them dashed when the tests were shown to be too expensive to maintain over the long haul. One key to producing cost-effective test automation is to minimize the maintenance costs. The application under test *will* change, and the automation *will* break. Another key is to maximize the reuse of the automation assets by implementing modular 'building blocks' that can be used in many different testing scenarios.

Building automated software tests require the same skill set and level of discipline as any other software development effort. Selecting a good test automation architecture is thus foundational to building automated tests that are both robust (resistant to breakage) and maintainable (easy to fix when the inevitable happens).

Best Practice #4: Keep Functional Test Automation Out of the Critical Path

Automated tests take from three to ten times longer to develop and debug than the equivalent manual tests. Since the goal of any software project is to deliver high-quality software as quickly as possible, putting test automation in the project's critical path means that either testing is going to take three to ten times longer, or else you will complete three to ten times less testing in the same amount of time. This is usually not acceptable to most projects.

It is generally better to decouple the automated test effort from the release schedule and let the automation engineers develop tests to use as regression tests in the next releases of the product. At the same time, it also makes sense to rely on traditional manual testing (directed, exploratory or both) to achieve the maximum amount of testing within the current release schedule. The manual tests can then be put into the queue as candidates for automation.

Best Practice #5: Design Manual Tests in Collaboration with Test Automation Engineers

It is a mistake to assume that automated tests can ever replace manual testing of developed software. The ideal arrangement is to design tests with manual and automated test engineers collaborating on the design. For example, manual testers often have a better knowledge of the business domain than automation engineers, who in turn can often design data-driven tests that can test by using a large number of different inputs—far more than a manual tester could execute in a reasonable time.

If the manual and automation teams adopt a methodology such as Keyword-Based Test Design, they can produce additional savings by making the documentation of the manual tests more concise and in a form that can be automated without having to perform a 'translation step' from manual documentation to automated test. The cost of the collaboration effort is more than recouped by the savings realized further on in the project.

Best Practice #6: Use Keyword-Based Test Design Techniques

Keyword-Based Test Design techniques have been accepted as a cost effective way to automate tests for over 10 years. Keywords are also known as Domain Specific Test Languages and are ideal for testing the AUT through a Graphical User Interface (GUI). From an automation point of view, Keyword-Based Test Design techniques provide the following advantages:

- A well-defined structure for producing modular tests with a high degree of reuse.
- Test scripts that are easy to maintain and enhance as the AUT or test equipment changes.
- Excellent return on investment, especially when used for regression tests over the life of the application.

Keyword-Based Test Design is also useful for manual test documentation:

- Provides a two-level documentation scheme that answers the question 'How much detail should I include in my test design?'
- Helps insulate the test design from the details of the implementation.
- Makes the transition to automated testing easier.

Best Practice #7: Provide for Fast Data-Reset Capabilities

If your application uses a database or other back-end data store, test automation will require increased discipline in the management of the data. This is because a key requirement for automated tests is to run them repeatedly. If your automated tests change the state of the database, you'll need to provide a way to reset the data to its previous state before you rerun the automated tests.

Plan to implement project-wide, test-data management policies and procedures. These should support the rapid reset of the database state. A good rule of thumb is to plan to reset the underlying data stores in one or two hours.

As delivery deadlines approach, your test team will be under increased pressure for a quick turnaround of test automation runs. Lack of a strategy for handling the test data reset could result in schedule delays. There are different strategies for managing the test data, and since the strategy, policies and procedures affect development as well as testing, be sure to secure buy-in from the groups that will be affected.

Remember to plan for enough duplication of data stores to handle the anticipated number of parallel test cycles. At the very least, each test and development environment should have a separate, independent set of data stores. In some cases, you may want more than one set per test environment.

Best Practice #8: Review Test Artifacts with Development

Formal, technical reviews of engineering documents provide the best return on investment of any testing activity. Schedule enough time to conduct a thorough review of the test strategy and (later) the test plan documentation. The goal is to ensure test planning aligns with the designed functionality.

The reviews should be formal in nature:

- Schedule well in advance with members of both development and test in attendance.
- Distribute review documents far enough in advance so that participants can prepare their comments.
- Cover material in a systematic manner during the review meeting.
- Minimize discussion of alternate or better ways of accomplishing the test: It is common for review sessions to become a forum where the design of the application and the design of the tests are examined, clarified, corrected and extended. This is acceptable as long as everyone agrees the discussion is warranted.

Best Practice #9: Use Agile Principles

Test automation efforts are affected by shifting schedules, changes to the AUT, requirement changes, and changes in project priorities. For this reason, it is very difficult to set a realistic test development schedule in advance and to stick to the schedule if you actually manage to develop one.

Adopting agile principles can help. Agile accepts and embraces the fact that change is continuous and inevitable. It helps you deal with the changes and uncertainties in a disciplined way. Note that we are not saying you must implement the whole agile process—only that you can use certain parts to your great advantage.

Develop your automated test in a series of two-week sprints:

- List all tasks requiring effort on the part of a team member in the sprint plan.
- Keep the amount of work scheduled in the sprint constant.
- Plan the content of the sprint at the very beginning of the sprint.
- Ensure the Test Lead maintains the sprint plan and tracks progress; progress should be reviewed at daily stand-up meetings.
- Develop burndown charts that are updated frequently (daily) to show progress against the plan; educate management in how to use the information in the burndown chart.
- Calculate and continuously update the team's *velocity*, a measure of how much work the team can accomplish during the sprint; this helps forecast when the project will be completed.

Conclusion: Avoid the Costs of Dissatisfied Customers and Bugs Discovered Late in the Process

In applying the nine best-practices for test automation, you'll be taking a high-level view of the entire project.

Start with the first best practice and then work your way down to ensure your team executes upon a sound strategy. As you consider the tactical best practices, which are somewhat independent, you can choose those that apply to each development project.

Success will largely depend upon your ability to persuade automated testers, manual testers and developers to all communicate effectively with each other. Cross-group communications is often the first function to be sacrificed when there's not enough time and corners have to be cut. But being disciplined in making sure sufficient communications occur makes the entire development process flow more smoothly.

Measuring ROI presents a difficult task because automated testing is just one component of the software development process. Essentially, you test to avoid costs—not to generate revenue or reduce development costs. But consider the significance of the costs you avoid—code changes discovered late in the process and customers that are not satisfied with the results of your software. By following these nine best practices, you can identify the holes and how to repair them in order to prevent your software from crashing and burning.

About the Author:

Kenneth "Chip" Groder, Intervise Testing Group Lead, has been involved with software quality and testing for over 30 years. He started his career in 1979 with Digital Equipment Corporation as part of the Software Quality Management group, and since then, he has held test engineering, management and architect positions with a variety of companies. Chip led the team that introduced GUI test automation to DEC in the early 1990s and has specialized in automation ever since. As a Test Automation Architect with Intervise Consultants, he has helped a number of Fortune 500 companies implement successful, company-wide test-automation programs.