# Development-Driven Testing: Ensuring Testing Meets the Needs of Software Developers

**Tim Farley**
tim.farley@gylity.com

## Abstract

Testing to support the needs of software developers is fundamentally different from testing to support the needs of end users. While both end users (external customers) and software developers (internal customers) rely on testing to demonstrate the readiness of the product, what is meant by “product” and “ready” is completely different. For end users, the product is the final product and it is ready for use when it meets all requirements. For developers, the product is the software they are in the process of creating and it is ready when it supports further development.

Software Quality Assurance (SQA) and testing teams are often focused exclusively on ensuring final products are ready for external customer use. Such teams often lack either the capability or capacity to provide the testing services that internal software development customers require, beginning with a clear understanding of what testing benefits developers most. It is impossible to meet internal customer needs when those needs are unknown.

Development-Driven Testing (DDT) provides a framework for SQA and testing teams to understand and meet the needs of software developers. The approach assesses the interfaces between the SQA and development teams, the information exchanged between them and the timing of their interactions to determine whether they are adequate to support the needs of developers. Test development, documentation, execution and results reporting can then be optimized to improve that support. By ensuring that testing meets the unique needs of developers, SQA and testing teams can improve software development efficiency, shorten software development cycles and improve product quality.

This paper describes how to create a DDT strategy. It describes how to identify the unique needs of a development team and create a testing strategy to meet those needs by analyzing the interfaces, information and interactions between the teams. The paper concludes with a project walkthrough where adapting the testing strategy to meet developer needs led to improved product quality and development efficiency, and saved the company millions of dollars compared to legacy testing strategies.

## Biography

*Tim Farley has 25 years experience in software quality assurance. He has led teams working on products ranging from critical medical devices to iPhone apps. For the past 10 years, Tim has focused on test automation, tools and bringing consistent practices to diverse SQA and software development teams across large organizations. Tim last presented a paper at PNSQC in 1996 about creating “intranet” sites, which has since proven to be a pretty good idea.*

*Tim has degrees in Computer Science and Anthropology from Brown University.*

# 1 Testing that Meets the Needs of Developers

Software developers have needs. Unfortunately, software quality assurance (SQA) engineers don't always know what those needs are. SQA organizations that understand how software developers work, what is important to them and how testing can help can have a significant positive impact on developer productivity, project duration and product quality. An SQA organization that understands the needs of developers can ensure that the right testing is done at the right time and in a way that makes further development efficient.

It's not enough, though, to understand this in general. It is critical for SQA teams to understand the exact needs of each development team for each specific project. What improved efficiency for one team may be an irritating distraction for another.

Development-Driven Testing (DDT) provides a framework for understanding the specific needs of software developers and assessing whether those needs are being met. It examines the interfaces through which SQA and development organizations communicate, the information that gets communicated, and the timing and participants involved in these interactions. It optimizes testing processes, tools and practices to ensure that developer needs are met.

Note: In this paper, SQA refers to software quality assurance and testing organizations. Software development organizations will be referred to as Development.

# 2 Creating a Development-Driven Testing Strategy

Creating a Development-Driven Testing strategy involves identifying the needs of Development, assessing whether SQA is currently capable of meeting those needs, and finally optimizing the testing tools, processes and practices to meet those needs.

## 2.1 Identifying Developer Needs

The first step in creating a DDT strategy is to understand the needs of the development team. These needs will be discovered through team discussions, observations and participation. There are four topics that can be examined first to get the process started.

- Efficiency – How much time do developers spend doing what they intended to be doing?
- Feedback – How long does it take to determine whether something is correct?
- Unique Needs – What are the special things this team is trying to do?
- Measurements – How are we doing?

### 2.1.1 Efficiency

Efficiency is the extent to which developers spend their time doing what they intended to be doing. Under ideal circumstances, this means never being distracted from creating working software. Unfortunately, there are many distractions that can prevent developers from doing this.

- Misunderstood Requirements
- Broken Builds
- Defects

Obviously, SQA shouldn't be creating distractions that make development less efficient, but sometimes distractions happen.

- Defect reports may be incomplete and require developers to gather missing information on their own.

- Tests may be run at the wrong time or on the wrong build.
- Test results may be reported that are difficult for developers to interpret.
- Tests may be incomplete and lead to product failures that are only discovered post-launch.

All of these can unintentionally distract developers, slow software development and reduce product quality.

There will also likely be services that SQA provides for developers that neither help nor hinder efficiency. For example, Development may be perfectly satisfied with manual testing. SQA tools, processes and practices that are merely sufficient for developers will likely become candidates for change when SQA begins optimizing. SQA should also understand where it is absent from the software development process and why. There may be opportunities for SQA to increase developer efficiency by performing completely new services. For example, SQA may only test integrated systems and not the individual components that make up that system. Testing components may provide new and unexpected efficiency improvements.

### 2.1.2 Feedback

Feedback is when we learn whether something we've created is actually correct. For example, when Development builds a new release, SQA tests that release to determine whether the software functions as expected. The feedback comes when test results are returned to Development. Likewise, when SQA creates a set of tests, the feedback comes when Development reviews the tests. Feedback always requires the creation of something new and the confirmation that it is correct.

Feedback ensures that projects are on track to deliver what they intended. The timeliness of the feedback and the response when it is received have an enormous impact on software development efficiency. For example, testing features as soon as the code is available provides timely feedback to developers, and immediately fixing any failures found maintains developer efficiency. Feedback from external customers post-release is always untimely feedback and fixing post-release problems distracts developers from current work. It can also distract SQA when post-release fixes require extensive regression testing to prove that the new code is better.

SQA needs to understand which Development deliverables require confirmation, what it takes to confirm the deliverable, and when that confirmation needs to be delivered. This requires dedicated cooperation and collaboration between the teams to ensure that the right feedback is delivered at the right time. For example, developers might want their code changes tested immediately after check-in. To provide timely feedback, SQA would need to know what changed, what tests are appropriate to qualify the changes, who needs to know the test results and when those results are needed.

Feedback content and format should always be appropriate for the intended audience and use. Feedback that is delivered at the right time but is unusable can reduce efficiency just as much as not returning any feedback at all. For example, developers may decide to ignore test results if they are delivered in a format that requires too much of their own investigation and filtering to find what they need to know. This increases the risk that they will be distracted by more defects later. It is also a waste of time and resources for SQA to run tests that benefit no one.

SQA also needs to understand how feedback cycles are related. For example, a software release delivery cycle may consist of the following separate cycles:

- Create a new release and confirm that it has been built successfully
- Test the new release and review the results
- Investigate reported failures and identify their causes
- Fix the failures and run regression tests

Each part of this cycle provides input to the next part. A build is delivered to test. Test results are delivered to developers. Developers act upon the results. A new build is retested. Teams need to understand not only how to complete their parts of the cycle, but also what they need from other teams and what other teams need from them to complete their parts.

In general, SQA organizations need to understand what they do right and wrong from the Development perspective, and Development needs to understand what they do right and wrong for SQA. For example, SQA could be delivering test results to Development at the right time, in the right format and with the right content. If Development is not able to review the results and fix failures as expected, SQA can't regression test in a timely manner and the cycle becomes inefficient.

#### 2.1.3 Unique Needs

Developers have needs beyond efficiency and feedback. The needs could be related to the way the development team works by itself or the way it works with other development teams. The needs could be unique to the project or the current state of the project. The needs could also just be things that make the development team feel confident about their work and optimistic about the future success of their project. It might be difficult for developers to articulate these needs. It may be easier for SQA to suggest what they suspect might be a unique need to encourage developers to confirm or deny it. SQA should attempt to meet any unique developer needs.

#### 2.1.4 Measurements

SQA needs to understand the measurements that are valuable to Development. These will likely include measurements related to efficiency and feedback, such as "On Time Delivery" and "First Time Right" measurements. Examples of these include tests passing, defect reports that contain all required information, and tests that are available to run when code is available to be tested. SQA should confirm whether there are specific targets for these measurements, such as verifying defect fixes within a certain amount of time of the fix being available. SQA should also confirm whether developers are interested in tracking trends, such as daily time spent waiting for test results throughout the project.

SQA and Development should also discuss whether legacy[1] measurements are still important and accurate. Some legacy measurements might be misleading with a new Development-Driven Testing strategy. For example, on legacy projects, test execution time may have been used to measure SQA efficiency. By this measure, SQA becomes less efficient if more time is spent during test execution collecting better failure information that then allows defects to be corrected quicker. SQA and Development may decide that completely new measurements are needed. Development and SQA should also keep in mind whether there are measurements that can prove that the new testing strategy is achieving better results than the legacy testing strategies.

## 2.2 Assessing SQA Capabilities

In addition to knowing the needs of Development, the SQA organization needs to know whether it has the capabilities to meet those needs. One way to assess this is to examine how the SQA and Development teams currently work together. The components of their working relationship can be thought of as:

- Interfaces
- Information
- Interactions

Once these are understood, it will be clear whether SQA can meet Development needs without changes.

[1] In this paper, the term legacy simply refers to anything that has been done before and is not a change for Development-Driven Testing.

### 2.2.1 Interfaces

Interfaces are the channels through which information is exchanged and the rules that govern their use. All communications require an interface. Interfaces include face-to-face discussions, e-mails, instant messages and phone calls. They also include tools and processes where information is produced, processed, stored and exchanged. Anything that requires the participation of both SQA and Development requires an interface.

Interfaces have to be used correctly for teams to work effectively. For example, the defect tracking system is an interface between SQA and Development. Incomplete defect reports can force developers to gather missing information on their own, making them less efficient. Using this interface correctly would help Developers maintain efficiency.

It should be clear to SQA which interfaces are needed to support Development requirements. SQA should have Development explicitly confirm them. SQA should not assume that interfaces that exist with other teams exist for the current team. Assumed interfaces that don't in fact exist will reduce efficiency and affect the timeliness and correctness of feedback.

### 2.2.2 Information

Information is the content exchanged through an interface and the format that content takes. Information can include code, tests, specifications, test results, builds, status, metrics, defect reports and schedules. Both the content and format are defined by the interface. Interfaces won't work effectively if the information does not conform to the interface requirements.

For example, tests are created using an interface. Requirements to be tested are identified, tests against those requirements are created, and those tests are reviewed to ensure that they are appropriate. A test development interface may define a process where requirements and tests are developed collaboratively between Development and SQA, and where test reviews are concurrent with test development. The interface may also define how those tests are documented so that they are efficient to write and review.

Information content and format should always be appropriate for the intended user and use. This will ensure that the information can be used efficiently. Deficiencies in testing information format and content will often be indicated by frequently repeated questions from developers. For example, developers may always ask whether a test failing on the current build passed the last time the test was run. Delays while waiting for answers can reduce developer efficiency.

### 2.2.3 Interactions

Interactions define when, how often and who is exchanging information using an interface. Interactions can include developers delivering a daily build to SQA, SQA running tests on a daily build, and SQA delivering daily build test results to developers. Feedback cycles always involve interactions.

Interactions are scheduled to occur when the information involved is most valuable. For example, developers benefit most when they receive test results immediately after they have made code changes. Waiting to find out if code changes worked is not efficient and delays development progress. Teams should also consider what could go wrong during an interaction so that there is adequate time to solve any problems that arise. For example, SQA could complete test execution as scheduled but leave no time for developers to fix any failures before the software release is due.

Teams should also ensure that any resources required to successfully complete an interaction will always be available, or agree on what would be an acceptable alternative. For example, if a specific hardware test fixture is not available for SQA to use, the teams could agree on an alternative fixture. The teams could also agree that the test should be delayed until the correct hardware is available. It is important for interactions to produce something of value, like credible test results, not simply take place.

Interactions can also affect and be affected by other interactions. SQA needs to understand how interactions are related. For example, SQA should schedule system testing to begin only after component testing has completed. This should be apparent to SQA since the system testing interaction can't being if the interface requires results information from the component tests, which isn't available until the component tests complete.

## 2.3 Optimizing the Testing Strategy

SQA and Development need to work well together to be efficient. Even teams that are working well together may need to become more efficient based on project constraints. SQA teams can optimize their interfaces, information and interactions with development teams to improve efficiency. Optimizations can include:

- Creating new interfaces or interactions
- Improving the information provided to interfaces
- Training staff to correctly use interfaces between Development and SQA
- Replacing ineffective interfaces, information and interactions

For any optimizations, SQA should confirm that the changes are having the intended effect. For example, SQA may find that developers are not processing test results quickly enough. To improve responsiveness, Development and SQA could decide to have SQA run tests less often, report results to developers less often or provide additional filtering of results so that only the most critical failures are reported to developers. (Development could also decide to increase their capacity to respond, but that is outside the scope of testing.) The teams should measure whether developer efficiency is improving.

Sometimes optimizations can have the opposite effect. For example, SQA may begin reporting test results to developers more frequently because SQA automated their manual tests. SQA may have expected this to improve Development efficiency but instead it made developers less efficient because of the additional time needed to process the more frequent rest results. Changing the frequency of test execution or results reporting could recover development efficiency.

Changing tools and processes can improve interfaces too, though careful thought has to be given to how those tools and processes will be deployed to the teams and how the teams will be trained to use them. Inadequate planning will likely reduce developer efficiency. For example, Development and SQA may determine that a new defect tracking system will improve developer efficiency. Putting the new tool in place without any training will likely reduce efficiency. Deploying the new tool in the middle of a critical period in the project will also likely reduce efficiency, even if the team has been properly trained. SQA needs to know not just what optimizations to make, but when to make them.

Tool changes can also optimize information. For example, if critical information is often missing from defect reports, it may be more effective to change the defect tracking system to require that information than it is to train users to always include it voluntarily. Tool changes can also optimize interactions, such as using an automated test system to execute tests whenever there are check-ins rather than relying on manually initiated test execution.

Interface optimizations can include improving SQA judgment about which interface to use when, such as when to make a phone call instead of submit a defect report to notify a developer about a new failure. This can have a surprisingly significant impact on how well the SQA and Development teams work together.

SQA and Development need to agree on the optimization strategy. While the SQA team may be able to implement some optimizations entirely within the SQA team, other changes may require the cooperation of the developers and program management. SQA needs to ensure that the project team understands what is involved in implementing the new testing strategy, how that strategy will be deployed and what benefits the team can expect.

# 3 Applying Development-Driven Testing

The following describes the software testing strategy for the development of a new print engine[2] component for a large print/copy/scan/fax system. This was a “clean sheet” design delivered over 5 years involving 20 software engineers, 200 mechanical engineers and 5 SQA Engineers collocated at one site. The print engine was developed iteratively, meaning:

- Enough code was written to bring up a new prototype revision
- Development learned from that prototype how to create the next revision
- Old code was discarded and new code was written to run the new prototype revision

SQA was engaged in the project following the initial iteration. Over six months, SQA and Development collaborated to produce the list of Development needs, the assessment of SQA capabilities and the new testing strategy optimized to meet Development needs.

## 3.1 Identifying Development Needs

There were several program constraints that influenced the needs of the development organization:

- Over 200 engineers were involved in the iterative creation of the hardware and software. The daily cost to the program for this many engineers was incredibly high. Even a single day of lost productivity would be an unacceptable expense and anything that significantly reduced the long-term efficiency of the team would put the entire program at risk.
- Iterative hardware development meant that prototypes would be hand-built and extremely expensive. Software developers and SQA would have few, if any, prototypes for software development and testing. Mechanical engineers would need to be able to use whatever prototypes they have for as long as possible.
- In addition to the print engine, the rest of the components making up the complete system would be new too. Integration of these components into a system would not be at the same site as print engine development. There would be at least an 8-hour delay between finding a problem at the integration site and engaging a print engine developer to investigate.

Based on these constraints, the following critical Development needs were identified:

- Immediate Feedback. Failures needed to be found immediately and fixes needed to be provided during the same day so that a stable baseline always existed for further development.
- Support for Multiple Simultaneous Builds. The program would need to extend the lives of prototypes as much as possible. It would be too costly to replace all prototypes whenever a new revision was released. Software would have to be written, maintained and tested for all prototype revisions in use. Development and SQA would have to create and ensure working software for multiple prototype revisions without regular access to hardware.
- Component Qualification. The print engine should never be the source of system integration problems and always needed to perform as expected. Any print engine problems found during system integration could never be addressed in a timely manner and would always create unacceptable delays and distractions for the print engine team.

[2] A print engine is the part of a printer that actually puts the image on the page.

## 3.2 Assessing SQA Capabilities

For each critical need, SQA Lead assessed the existing interfaces, information and interactions in use during legacy projects and determined whether they would be capable of meeting the needs of the print engine software developers.

### 3.2.1 Immediate Feedback

SQA investigated what generated feedback to developers, how often it occurred and whether it was sufficient to support the critical needs of Development.

- Test Development – Can tests be developed and maintained quickly enough to provide immediate feedback? When are the tests needed? Who is involved in creating the tests?
- Test Execution – Can tests be run frequently enough to provide immediate feedback? When do the tests need to be run? How often? Who needs to run the tests?
- Results Reporting – Can results be reported clearly enough so that they can be acted upon immediately? Who needs the results? When are the results needed?

#### 3.2.1.1 Test Development

**Current Status:** SQA developed tests against traditional specifications. The specifications were written by developers and identified testable requirements. SQA developed test cases against the testable requirements and developers reviewed the tests to ensure that they were appropriate. Tests were documented to show traceability to requirements and written in a clear, concise manner to allow a wide range of reviewers to provide useful feedback. Specification changes were communicated to SQA through the defect tracking system.

**New Requirements:** Developers would not be writing any specifications. Software development would be driven by the results of iterative experiments rather than static requirements. Any specifications that were written would not affect print engine software development, would only be used by SQA to develop tests against, and would require constant maintenance. SQA would still need to demonstrate test coverage against requirements in a reviewable format.

**Assessment:** SQA would need to create a new way of developing tests without specifications written by developers and anew way of keeping those tests up to date despite constant code changes.

#### 3.2.1.2 Test Execution

**Current Status:** Test execution cycles typically lasted from one week to two months depending on the scope. Testing cycles contained both fully automated (no human interaction required) and manual (human interaction required) testing, with tests run on prototype revisions most closely resembling the production units external customers would receive. The release tested during planned testing cycles could be a daily build, a weekly release or a release corresponding to a program milestone.

**New Requirements:** Testing the print engine would require more frequent testing cycles. Rather than one week to two months, testing cycles would need to complete in one day. Each testing cycle would need to be a complete regression test of all print engine functionality. SQA budget constraints would prevent hiring enough manual testers to execute a complete regression test every day. Budget constraints would also prevent purchasing and maintaining a sufficient number of prototypes to execute a complete daily regression test.

**Assessment:** SQA would need to create a new way of executing tests that did not require a large number of manual testers and prototypes and that could complete a full regression test daily in time for developers to make use of the results.

#### 3.2.1.3 Results Reporting

**Current Status:** SQA reported results weekly to a cross-functional team at the program status meeting. For weekly testing cycles, the report would summarize the entire testing cycle. For longer testing cycles, the weekly report would show the progress being made against the test cycle plan. The report was compiled by hand and included data from the test management database as well as test results communicated through e-mail and spreadsheets. The finished report was e-mailed to the project team members.

**New Requirements:** Testing the print engine would require more timely reporting of results. Results would have to be delivered daily and at a time when developers could act upon them effectively. The results would need to contain information targeted to developers rather than a cross-functional program team and would need to be in a format that assisted developers in identifying and isolating regression failures. All the information needed to investigate a failure would need to be included with the results.

**Assessment:** SQA would need to create a new way of reporting results that supported daily regression testing and defect correction.

### 3.2.2 Support for Multiple Simultaneous Builds

**Current Status:** SQA always tested one software build at a time. The software build was for the most recent, customer-intent prototype. SQA would frequently have to upgrade prototypes during the course of a project so that tests were always run on customer-intent hardware. Testing qualified the hardware and software for external customer use.

**New Requirements:** Print engine testing would need to focus on enabling hardware and software developers to sustain their development efforts. Multiple, simultaneous builds would have to be tested to allow development to continue on any available prototypes. All requirements for test execution, including running a complete regression test and reporting results daily, would have to be met for each software build for each prototype revision in use.

**Assessment:** SQA would need to change the way it executed tests and reported results so that multiple, simultaneous builds could be tested concurrently.

### 3.2.3 Component Qualification

**Current Status:** SQA only tested integrated systems. No components were tested separately. Software releases contained all components integrated into a system and system tests were run on customer-intent hardware prototypes. Human testers interacted with the devices in the same way that end users would.

**New Requirements:** Testing the print engine would require testing the component without integrating it with the rest of the system. Tests would have to use some other mechanism of interacting with the print engine software than with the physical hardware interfaces (front panel buttons, doors, trays, etc.…). Sufficient hardware would not be available to SQA to run the tests on prototypes.

**Assessment:** SQA would need to create tests to qualify the print engine component in isolation of the rest of the system and the print engine tests would need to run without prototype hardware.

## 3.3 Optimizing the Testing Strategy

The SQA team needed to create a new strategy for testing print engines. It needed to adapt test development, execution and results reporting practices to provide software developers with immediate feedback. SQA needed to create new methods to test multiple, simultaneous builds and test software components in isolation of the rest of the system. And it needed to be able to execute this strategy while meeting budget constraints for the program.

The new strategy involved the creation of:

- A simulation environment for testing print engine components without hardware.
- Fully automated tests requiring no human interaction for execution or results reporting.
- A scalable test infrastructure that supported testing multiple, simultaneous builds.

SQA also changed the interfaces, information and interactions involved with developing and executing tests, reporting results and investigating failures.

### 3.3.1 Test Development

To address test development, SQA Engineers collaborated with developers to create test documentation that could act as specifications by example. This eliminated the need for separate specifications and improved developer efficiency. Understanding the requirements and tests before coding also prevented design and implementation problems that would have reduced developer efficiency. Constant face-to-face conversations between developers and SQA Engineers ensured that the tests were always up to date and correct. All tests were developed in parallel to the print engine code so that tests were available to run as soon as the code was ready to be tested.

### 3.3.2 Test Execution

Several changes were needed to enable SQA to run a complete regression test of the print engine component without people or prototypes.

- SQA collaborated with developers to create a simulation environment using only a print engine controller board and a power supply. All interfaces to external hardware were simulated in the print engine code. With the code running on the actual controller, testing would ensure all real time requirements were met. The cost of the print engine simulator was 1/100th that of a full prototype, allowing SQA to have an adequate supply for parallel test execution.
- SQA and developers created a new programming language to simulate communications with other system components. This allowed the print engine component to be tested in isolation. The programming language implemented the same communications protocol used by all the components in the system.
- SQA created automated tests. No manual tests were written. The automated tests could be run without modification on a simulator or a hardware prototype. While tests would run without any human interaction on simulators, they would prompt the test operator to interact with the device when run on a hardware prototype. Developers ensured that the simulator supported all actions needed to execute tests and verify results.
- SQA created a scalable test environment that allowed test execution to be distributed across multiple simulators. Automated tests were structured to support parallel test execution, with no single test suite taking more than 1 hour to complete. This allowed the number of tests to grow, yet still have test execution complete in the same amount of time, by simply running the tests on more simulators. Testing multiple builds only required adding more simulators.
- The SQA automated test environment allowed developers to select and run automated tests. Automated tests initiated by developers would run exactly as they ran during planned SQA testing and would contain all the same results, links and logs.

### 3.3.3 Results Reporting

Many changes were required to ensure that results were reported at the right time, that the results were actionable and that developers acted upon them.

- SQA designed the automated test environment to monitor the daily build status, upgrade simulators with the latest build, and execute tests. The test environment captured and stored all logs generated by the simulators, test environment and automated tests, and associate them with the individual test results.
- The test environment automatically packaged up all results, links to tests and log files, and built an e-mail message. The results e-mail was sent to all developers. Because no human interaction was required, the end-to-end process could run as soon as the daily build was available, typically at 1:00am, and results would be waiting in the inboxes of developers when they started their day.
- The format of the test results e-mail allowed developers to see the most important information without scrolling. Developers could quickly tell whether any further reading was required for that daily build.
- Test results were organized by software subsystem and named using the same terms used by developers. This made it intuitive to find test results of interest.
- Test results were listed with simple tests first. Test results for complex tests were built upon the known results of the simple tests. This made defect isolation more efficient.
- The report highlighted differences from the previous testing cycle (yesterday's build) so the impact of changes would be clear immediately. Differences included tests that were now failing as well as now passing.
- On the occasion when test failures were not immediately addressed, the failures were annotated in the report with references to defect reports. Annotations also included the number of days the failure had persisted in the code. This allowed developers to quickly identify new failures.
- SQA created a series of automated tests to determine quickly which features were supported in each new build. The test environment only executed the tests for which all the required features were supported. This eliminated obvious failures from the report that would otherwise be a distraction to developers.
- Software development managers ensured that failures were addressed as quickly as they were reported. Managers reviewed daily test results and contacted their developers directly to ensure that failures would be fixed and retested that morning. Failures where the fixes that could not be delivered immediately were addressed by backing out the changes and rebuilding the code. The SQA team provided the same responsiveness to test and test environment failures.
- SQA investigated, isolated and recommended fixes for print engine failures. SQA Engineers reviewed the daily test results against the check-ins from the previous day and contacted the developers to discuss the possible causes of failures. This helped to pinpoint defects more quickly and became a critical service for the development team.

### 3.3.4 Support for Multiple Simultaneous Builds

Development agreed to create and maintain builds for the current prototype and the previous two revisions. SQA tested all three builds daily. Automated tests and the test environment code branched along with the print engine code for each new prototype revision so that the right tests would be run in the right environment for each build. Separate results reports were e-mailed for each build tested.

### 3.3.5 Component Qualification

SQA verified all print engine component requirements by executing a complete automated regression test on print engine simulators. SQA supplemented the simulator tests to ensure that the print engine component would work correctly in a fully integrated hardware system. This included:

- Verifying the system communications protocol requirements. A complete automated test was run as part of the daily regression test to verify that the print engine component met all communications protocol requirements.
- Verifying that the print engine ran correctly when integrated into the complete system. Once a week, simulator testing was supplemented with testing on a real device. The 4-hour test covered the highest priority features and could be run by one person on one prototype.
- Running a complete regression test on a hardware prototype prior to every significant program milestone. The test, which took 80 hours to run on simulators, took one human tester 6 weeks to run on a prototype. SQA planned for these testing cycles to complete a month before each program milestone. This provided developers with enough time to fix any print engine problems before the component had to be released for system integration.

### 3.4 Impact

The print engine went through 8 prototype iterations. The final version of the software contained 450,000 lines of code. When the product was released, it was the most reliable print engine in the company's history. The testing strategy was credited with reducing the duration of prototype iterations and overall product development. The print engine was never the cause of integration failures during 5 years of SQA engagement and automated testing. Automated testing and hardware simulation saved the company tens of millions of dollars over manual testing on full prototypes. By eliminating most printing of physical pages, hardware simulation alone reduced paper and ink expenses enough to pay for the entire SQA effort.

Executive management enlisted other teams to investigate how the methods could be applied to their programs throughout the company. The automated test environment was replicated at other development sites and enhanced to test integrated systems as well as print engines. SQA also applied the Development-Driven Testing methods to other internal customers, including technical publications, manufacturing, customer support, field service and program management.

## 4 Conclusion

SQA organizations need to see software developers as customers for their testing services and ensure that the services provided absolutely meet their customers' needs. This requires understanding how the development team works, what is important to them and how testing fits into the process. SQA organizations need to reassess with each new project and each new development team whether their current testing practices are adequate. Crafting new testing strategies becomes a collaborative effort between SQA and Development.

Development-Driven Testing provides a framework for discovering the needs of developers, analyzing the interfaces, information and interactions between Development and SQA and assessing whether current SQA practices are capable of meeting Development needs. The DDT framework can be applied to other internal customers in addition to software developers.

SQA organizations can do more than validate products for end users. They can use testing to improve software development efficiency, reduce software development costs and improve product quality. SQA can have a significant impact on how products are developed and how successful they will be.

## Acknowledgements

I would like to thank Mike Herbert, Jeff Lowe, Lynd Wieman, and Jeremy Smith for their contributions to this process, and to Tektronix for teaching me about internal customers many years ago. Special thanks to my talented reviewers for helping me turn this into something that people will want to read. Extra special thanks to David Butt for suggesting the name Development-Driven Testing. I like it!

## Recommended Reading

While the name Development-Driven Testing may be new, the idea is not. It is focused on providing excellent SQA services to internal software development customers to improve software development efficiency. DDT is best understood as an outgrowth of many generally accepted practices from the past few decades.

Beck, Kent 2000. *Extreme Programming Explained*. Addison-Wesley.

Coplien, James and Harrison, Neil 2005. *Organizational Patterns of Agile Software Development*. Pearson Prentice Hall.

Gilb, Tom 1988. *Principles of Software Engineering Management*. Addison-Wesley.

Grady, Robert 1992. *Practical Software Metrics for Project Management and Process Improvement*. Prentice Hall.

Poppendieck, Mary and Poppendieck, Tom 2003. *Lean Software Development*. Addison-Wesley.

Spradley, James 1980. *Participant Observation*. Holt Rinehart Winston.

Weinberg, Gerald 1986. *Becoming a Technical Leader*. Dorset House.

Weinberg, Gerald 1989. *Exploring Requirements: Quality Before Design*. Dorset House.