Quality Quest

Software Quality Articles by

Linda G. Hayes

Originally published as a monthly column in

Datamation
Magazine

Brought to you by

WorkSoft ENTERPRISE PRODUCTIVITY SOLUTIONS

Table of Contents

About the Author

Linda G. Hayes

linda@worksoft.com

As one of the nation's premier computer software entrepreneurs, Linda G. Hayes of Dallas stands out as a leading authority on software testing tools.

Linda, founder and former chief executive of the pioneering testing company AutoTester Inc. of Dallas, is now helping leading firms deal with the rapid shift to an electronic marketplace through a new company she has co-founded, WorkSoft Inc.

Linda has degrees in accounting, tax and law, is a CPA and member of the Texas State Bar. In 1982 she founded Petroware, an oil and gas accounting software vendor, and in 1986 co-founded Software Recording Corporation, the parent company of AutoTester.

She writes a monthly column, "Quality Quest," for Datamation magazine, speaks frequently at industry conferences, and is an often published author. She published the "Automated Testing Handbook", a guide to evaluating, implementing, and managing test automation tools. She also served as co-editor with Alka Jarvis for "Dare to be Excellent", which details best practices in actual case studies.

Linda also offers public courses on Automated Test Suites for Software Quality Engineering.

To read more of Linda Hayes' published work visit
www.worksoft.com

About WorkSoft, Inc.

WorkSoft, founded in 1998 by the industry's leading experts in software test automation, provides an automated approach to assuring that complex business systems are certified for production readiness.

The combination of the Certify™ repository, Business Process Certification™ methodology and Solution Set of services is the ultimate answer to enterprise-level automation across applications, platforms and test tools.

Certify™ transcends existing scripting technologies that are hard to learn and almost impossible to maintain. It requires no programming expertise; business users can become productive in a matter of hours, not days or weeks. When changes are made to applications, Certify™ automates maintenance through a unique facility that detects application changes, then provides impact analysis to identify and correct affected processes.

Certify™ supports WorkSoft's revolutionary approach to operations assurance, Business Process Certification™. This new philosophy and approach challenges traditional thinking about software testing and presents a model that makes sense and works in the real world of shrinking schedules and increasing risk.

WorkSoft's customers include some of the largest financial services and electronic commerce firms in the world. If your business operations rely on quality systems to survive, we can help.

To learn more about Certify visit www.worksoft.com

To Win at Software Development, Change the Game

by Linda G. Hayes

Originally published in Datamation magazine, July 1999

In order to succeed in developing software, you've got to discard old-fashioned methods and mindsets and come up with new ways to create good software.

Without exception, software developers and testers at every company I've ever worked in or with have harbored the belief that they--and they alone--are victims of uncaring management, undefined requirements, unreasonable customers, and absurd schedules, while other companies are developing and testing their software the "right" way.

Well I have news for you. Everyone, everywhere is in the same situation. The next company I see that has an orderly, timely, formal, and effective approach for developing software will be… the first. Don't get me wrong, lots of companies are making valiant attempts to define their development processes and stick to them, but somehow, somewhere, it all goes by the wayside.

Instead of clinging to the old systems-development life cycle because it makes sense--or used to--reorient your thinking to today's conditions.

The fundamental question is, why?

The good old days

Although we all laugh about how the mainframes of yore can fit on a chip today, the fact is, software development in the past was a far more disciplined process. Flowcharts, diagrams, and coding pads existed for a reason: At one point in time, people actually used them. Application development was approached as a scientific discipline and was commenced only after extensive analysis, planning, and design. Testing was equally systematic, with methodical techniques for deriving test conditions and cases. Computer programmers were revered as geniuses.

So what was different then?

It all comes down to time. In the early days, computers were new, and they were replacing tasks already being done manually. Accounting, for example, was commonly automated first. Because it was all so new and because there were manual alternatives in place, software could be developed over a two-to-three-year period and no one thought anything of it. Development took as long as was required to get it right.

The bad new days

Today, of course, 90 days is the norm for development cycles. Computers have infiltrated every aspect of our lives and company operations, forever changing the way we conduct communications and commerce. Software is a competitive weapon, used both offensively and defensively: If your competitor announces a new billing plan or fare-pricing program, you've got to respond quickly or risk losing business.

Software is a competitive weapon: If your competitor announces a new billing plan or fare-pricing program, you've got to respond quickly or risk losing business.

Ironically, this collapse in the timeline for development is exacerbated by an expansion of complexity in technology, a byproduct of advancement. The number of variables involved in hardware, software, and the applications they support now, compared to then, is orders of magnitude greater. So we've got more to do and less time to do it in.

And there you have it. It's not that everyone forgot how to do it right, or developed bad attitudes or sloppy practices, it's just that economic realities have forever changed. You can't win this game playing by the old rules.

What can you do about it?

Start by changing your beliefs. Quit castigating yourself, your company, your coworkers, and your customers for failing to adhere to principles that have all the immediacy and relevance of dinosaur mating rituals. Accept the fact that it's a new world and software is organic instead of static; that it's part of a continuous flow of economies and realities reflecting the state of commerce and not the state of the art.

Next, change your approach. Instead of clinging to the old systems-development life cycle because it makes sense--or used to--reorient your thinking to today's conditions. It's like changing from a marathon runner to a sprinter--it takes a completely different strategy and approach to succeed. Here are a few tips:

Become a team player. There is no substitute for a cross-functional team that includes marketing, development, testing, training, documentation, and support personnel. Act and think like a sports team: Everyone plays their position but covers for everyone else when necessary, and everyone wins or everyone loses.

Ask your customers to join the team. Instead of being treated as outsiders and critics of the product, customers should be insiders, participants to the whole process. If you sell your product to other organizations, recruit "alpha" sites to join in. This sense of ownership will help you--and your customers--make critical decisions about priorities, risks, and schedules. You'd be surprised how practical customers' demands become when they see what is actually involved.

Release early and often. Traditional wisdom says it's better to wait until all of the functionality is ready before releasing the product. These days, it's better to add functionality incrementally, in the smallest possible pieces that will still work and make sense. This gives you faster feedback and allows you to incorporate design changes before the product is too far along.

Pace yourself. Once you realize that every schedule will be a crunch, you'll stop believing you can kill yourself today and recover later. Later will never come. Instead, work hard and be just as aggressive about maintaining balance in your life. It will greatly improve your productivity and your attitude.

Measure what you do, not what you don't do. It's a crime to measure a product by its defects--it completely ignores what it does well. Software is never perfect, and it never will be. Instead, focus on what you achieved in terms of new functionality, problems corrected, and performance enhanced. This doesn't mean you should turn a blind eye to defects, just realize that they are only one small measure.

Simply put, the rules have changed. So, change your game and the way you play it.

Maximizing Customer Coverage

by Linda G. Hayes

Originally published in Datamation magazine, June 1999

Customer coverage testing means discovering how your software is actually being used and testing it that way.

In my April 1999 column, "The pain of platform possibilities," I discussed the impossibility of testing all the conceivable combinations of supported platforms. I also introduced the concept of certified vs. supported platforms as a strategy for identifying what can reasonably be tested internally and what should be supplemented with beta programs or risk mitigation strategies.

The next issue you are confronted with is what you can really test within the subset of certified platforms. Or, even if you support only one platform, what is realistic to expect in terms of test coverage? The word realistic is the operative word, by the way, because objectives like 100% test coverage are laughable in most cases.

Don't believe me? Let's do the math again.

Believe it

Let's say you are testing a file transfer system that supports six platforms, either sending or receiving, and on each platform you support three operating systems and two versions of each. Within each platform you have four protocols, five file formats, minimum and maximum file sizes, encrypted or nonencrypted, plus two different flavors of receipt confirmation.

Customer coverage testing means discovering how your software is actually being used and testing it that way.

Furthermore, let's assume you have reduced the scope of the test effort by agreeing to certify only four platforms and one version of each operating system per platform, and it takes you 45 minutes to configure a single source-target platform combination and 15 minutes to actually set-up, send, and receive a particular file, then check the results.

How many test cases and how much time do you need to test 100% of the possible combinations and only the valid boundaries?

I get 23,040 individual tests, which would require about 168 person-weeks if you get seven productive hours per day--and that's a stretch. You could probably automate a lot of this, but even that takes time and effort to develop and maintain. If you start adding in equivalence classes, including positive and negative cases, the number goes off the charts.

This is a fairly straightforward application. I don't know of that many companies--in fact, let's say none--that can afford this level of effort on a routine basis, although I sincerely hope those who are testing the software controlling weapons of mass destruction can and do.

Face it

My point is not to spread defeatism, but to simply expose the truth. Testing every possible combination of every factor is just not realistic. If you set this as your goal, or allow others to, you are doomed to fail and suffer.

Understand, I'm not against full coverage or high quality, but I am against setting unrealistic expectations that lead to disappointment, demoralization, and--ultimately--turmoil and turnover in the test department.

So the sane approach to test coverage is to devise a means of getting the most return on the time you are able to invest. This translates into reducing the most risk as opposed to achieving the most coverage.

The next question is, how?

Analyze it

Start with the most basic of all questions: Who are your customers? That is, who will be using this system, and what will they be doing with it? Most likely your customer support area can tell you this or, if not, perhaps the sales and marketing department. If you have to, review the sales contracts to see what has been sold.

I'm not against full coverage or high quality, but I am against setting unrealistic expectations of test coverage that lead to turmoil and turnover in the test department.

For example, an electronic commerce company discovered that most of its customers were financial institutions and that the overwhelming majority of them (85%) operated mainframe platforms running OS390 and using the SNA protocol. And, when researched further, it found that a single file format and encryption option (an industry standard) accounted for 90% of all file transfers.

This is a different kind of coverage. Instead of code or feature coverage, we'll call it "customer coverage."

Customer coverage

Customer coverage means discovering how your software is actually used and testing it that way. The easiest way to define coverage is to build user or customer "profiles" that describe a particular customer configuration and their typical activities. These profiles might be centered around industries, geographies, or other identifiers that affect the type of customer and how they use the software.

This has some interesting benefits. The first and most obvious is that it forms a natural basis for prioritization: you know what to do first and how to allocate your time and resources. It prevents you from wasting resources by trying--and failing--to test everything in every possible way. Instead, you make absolutely sure that the activities you know for a fact are critical are, in fact, thoroughly tested.

The second benefit is a little more subtle. Let's say you are running out of time to complete the test effort and it's critical to make the release date. If you have prioritized your test effort around customer profiles, you could do a "rolling release"--ship only to those customers whose profiles have been tested. That way, it's not all or nothing. If most of your customers fit into a particular profile, you can ship to most of them on time and only delay shipping to the minority of customers who fall outside the tested profile.

The third benefit is longer term but potentially very valuable. If you adopt this practice, it should lead to better information collection about your customers and how they use the software. Once you understand your users better, you can prioritize enhancement requests, new features, even bug fixes the same way--identify where you should allocate your development and test resources by achieving the highest rate of return in terms of lower support costs and higher customer satisfaction.

This entire approach also gives you a framework for incorporating reported problems into your test plan. Instead of just a "bug" that ends up multiplying as a new test throughout all of your test plans, you determine which customer reported it, what profile they fit into--or if you need a new profile--and add it there.

The goal is to continue to define and refine your test process so you know what to test and why, as well as what not to test and why. This is the only way to establish a basis for measuring, managing, and improving your test effort in a realistic--and achievable--way.

Management-Friendly Test Data

by Linda G. Hayes

Originally published in Datamation magazine, May 1999

Give management test metrics that measure the two things closest to an executive's heart: Time and money.

Metrics are trendy, and rightfully so: You can't manage what you can't measure. The problem is, almost all of us have slaved over collecting, formatting, and presenting vast storehouses of information that no one ever reads or comprehends, let alone acts upon. In our zeal to educate management about what we are doing, we end up inundating them with reams of reports that often simply confuse them.

Correlate all of your metrics into time and money.

The problem is not providing a quantity of information; it's providing the right information. The solution is to know what managers want to know and why they want to know it.

What's important?

A 1997 industry survey of software test organizations revealed an interesting paradox. When respondents--the majority of whom were in management--were asked to rank their most important objectives, first place went to meeting schedule deadlines and second place to producing a quality product. In a later question, when asked what their highest risk and cost were, based on past experience, first place went to high maintenance and support costs and second place to cost and schedule overruns.

Let's analyze this. Management's first priority is making the schedule, and their second is delivering a good product. Sometimes, this translates into sacrificing quality to make the schedule. On the other hand, experience shows (but apparently doesn't teach) that they encounter higher risks and costs from maintenance and support of low-quality products than they do from exceeding the schedule or budget.

Get it? Neither do I.

Why is it important?

The one clear thing is that management cares about time and money, and that makes sense. Most managers are measured on how well they meet their delivery schedules and their budgets. What seems to escape us, consistently, is the interplay between these two: If you meet the schedule but deliver a defective product, you spend more money trying to support and maintain that product.

Why is this painfully evident relationship so obviously ignored? Because of the imagined differentiation between development and maintenance. Few, if any, companies actually associate their maintenance and support costs with the sacrifices made to meet the production schedule. If you ship the product as scheduled, you "made it," regardless of whether that product boomerangs into a maintenance nightmare.

The most eloquent--and dramatic--example of this was when the new CEO of a software company asked me to review its operations to help him discover the reason why costs were increasing and revenues decreasing. After a couple of days of interviews, the mystery was solved. The company's budget for customer support was more than the budget for development, testing, training, and documentation combined.

Why was support so expensive? As the harried customer support manager explained it, the company had a huge backlog of bugs, some of which were actually years old, and these generated thousands of phone calls which her team was obliged to field.

Time and money wasted in support and maintenance on poor quality software could be reinvested in more resources to deliver new, high-quality products on time.

Why so many bugs? Because there weren't enough developers to maintain the products. Why not enough developers? Because there wasn't enough money to hire more. Why not enough money? Because support was so expensive. Why not increase revenues? Because they didn't have enough developers to create new products.

Get it? So did the CEO.

How do you say it?

It all comes down to this: How do you measure this phenomenon and communicate it in such a way as to have management understand and--most importantly--care?

Unfortunately, I cannot point to a magic answer. If you have one, send me an e-mail and I'll tell the world.

Until then, do the only thing you can do: Correlate all of your metrics into time and money. At a minimum, track the issues that arise after shipment and correlate them to the ones you either knew you shipped or could have predicted because you didn't finish testing. Remember, the number of reported problems is the number of defects multiplied by the number of customers who find them. So, shipping a single known defect can cause hundreds of problem reports.

The next trick is to convert this information into time and money. Money can be determined if you know the following:

- What the budget is for customer support and maintenance;
- How many problem reports are fielded.
- And how many fixes were made.

Time is harder to convert, of course, because you have to know how long it takes to field a call, make a fix, test it, and ship it. If you have a robust problem-tracking system you may have this information. If you don't, add up the manpower spent in maintenance and support and convert that into time.

Be sure to make the point that the time and money wasted in support and maintenance on poor quality software could be reinvested in more resources to deliver new, high-quality products on time.

The point

The real point, of course, is to understand your audience. Management cares about time and money, in that order. Present your metrics in such a way that you can correlate what you are measuring to what it costs--or saves--in time and money. All of the graphs, charts, and tables in the world won't matter without metrics that make sense.

The Pain of Platform Possibilities

by Linda G. Hayes

Originally published in Datamation magazine, April 1999

Just because the software your company develops runs on many platforms doesn't mean that every possible configuration can or should be tested.

While component-based architectures allow software developers to create applications that support many different databases, servers, and operating environments, they create a quality quagmire of nightmarish proportions for software testers.

The reason? It may take the same effort to develop an application for any ODBC-compliant database as it does for just one, but it takes a geometric multiple of that effort to test it because each and every database--in each and every potential platform configuration--must be tested. Different databases may have different reserved keywords, different sub- or supersets of ODBC support, or different constraints in different environments. Thus, each and every combination of all the elements must be tested together in order to truly assure quality.

"You mean to tell me you aren't even going to test the server platform that is used by the customer who signed our largest deal last quarter?!" he bellowed.

Do the math. If your application supports four different databases on six different hardware platforms under three different operating systems, you are looking at testing the same application 72 times! Throw in other variations, like middleware or network protocols, and you are in the stratosphere for test time.

Under such circumstances, any competent, thorough software tester who takes pride in shipping a quality product is doomed to be frustrated no matter how hard he tries. Not only is it impossible to test every single configuration possibility with the time and resources available, I have worked with several companies where the test group doesn't even have access to all of the supposedly supported platforms. As a result, customers uncover critical issues in the field, which is the most expensive place to fix them.

The odds are against reining in marketing or sales by limiting the platforms, since that's where the money is. What to do?

Define your terms

To defend its borders, the test group must define them. This means it must be clearly stated, accepted, and communicated to all concerned, both internally and externally to customers, which configurations are, in fact, tested and which are not. This frees the test group from spending all of its time explaining why--of the dozens of ones it did test--it did not test the exact one the customer is screaming about.

I recommend organizing around the concept of "certified" versus "supported" configurations. A "certified" configuration is one that is actually tested, while a "supported" configuration is one the company agrees to accept responsibility for resolving if it fails. This distinction is important for three key reasons: It defines the platforms within the scope of the test effort; it identifies the potential risk of those out of the scope; and it enables a mitigation strategy for those risks.

Certified configurations

The beauty of precisely defining which configurations the test group will actually test, or certify, is it reveals to the rest of the organization the cold realities of what testing is up against. For example, I was reviewing the certified configuration list with the sales VP for a financial services software company when he was shocked to discover that the test group was not going to test the server platform of a customer. "You mean to tell me you aren't even going to test the server platform that is used by the customer who signed our largest deal last quarter?" he bellowed. "Why the ---- not?"

I smiled. "Because our purchase request for a test server was denied." He looked astounded, then promised to get us access to one, somehow.

"Great," I said. I was now on a roll: "But there's one more thing. I either need two more people, or two more weeks in every test cycle to cover this additional platform."

Having just been the victor in a bloody battle to get development to agree to support this very server so he could book the sale, he was furious. "Why didn't someone tell me this before?" Again, I smiled. "No one asked the test group."

I am confident this scene plays out every day in many corporate enterprises. Adding support for a new platform is not just a development issue, it's a testing and support issue. In fact, you can compare the development effort to having a child: it may take nine months to develop it, but testing and support have to raise it for the next 18 or more years as it goes through every patch, release, and version of its life.

Supported configurations

Once you have identified the certified configurations, anything else marketing wants to sell becomes "supported." This means that although the company doesn't represent that it has, in fact, tested a precise configuration, it agrees to accept responsibility for resolving any issues that arise.

The ultimate benefit of clarifying what the company will test and what it won't is that it gives everyone a chance to mitigate risk.

At first this may sound like a PR problem, but in reality it's a plus. If the customer has a problem with a supported environment, it doesn't automatically raise the question of whether the company tests anything at all. Without this distinction--and we've all heard it before--when a customer uncovers a major problem with an obscure combination, he immediately freaks out and questions the competency of the test organization altogether. For most battle-weary testers, this can be a morale killer.

With supported environments, at least there's notice up front about what the company is committing to test versus what it's willing to take responsibility for. As a result, the customer realizes the risk it's assuming in adopting this configuration.

Mitigating risk

The ultimate benefit of clarifying what the company will test and what it won't is that it gives everyone a chance to mitigate risk. If company officials are concerned a key account is at risk because its configuration is not certified, they can mitigate that risk in one of two ways: One, they can invest in additional time and resources to cover that configuration; or, two, they can include the account in the beta program.

In fact, customers with supported configurations would be ideal candidates for early release. This would allow the test organization to extend its coverage without bloating staff or costs, and it would allow customers whose configurations are at risk to participate in the test process.

The point is, it's wrong to simply assume that test organizations can fully test every platform and every possible configuration the software can potentially run on. Both the company and its customers should be fully informed--and the test group fully equipped--to deal with the risks posed by the many potential platform possibilities. That way, testers have a hope of meeting quality expectations, because they will be both clearly defined and, more importantly, actually achievable in the real world.

The Problem with Problem Tracking

by Linda G. Hayes

Originally published in Datamation magazine, March 1999

Putting a system in place to follow up on software management issues is harder than you think.

Just when you think you have a handle on your development and testing processes, you realize you really don't. Now, you find out you need a robust means of managing the issues that arise from building and supporting your own software. Sometimes these issues are outright bugs, sometimes design flaws, and sometimes enhancement requests. But whatever their nature, you need an orderly way of tracking their existence, status, and disposition.

Putting a system in place to follow up on software management issues is harder than you think.

Just when you think you have a handle on your development and testing processes, you realize you really don't. Now, you find out you need a robust means of managing the issues that arise from building and supporting your own software. Sometimes these issues are outright bugs, sometimes design flaws, and sometimes enhancement requests. But whatever their nature, you need an orderly way of tracking their existence, status, and disposition.

What starts out as an innocent problem-tracking product acquisition often turns into yet another development and administration commitment.

You probably already have a tracking system of some kind in place: usually a homegrown database or customized groupware. And you may even have several of these systems sprinkled throughout different departments. The odds of keeping these homegrown systems maintained and supported--not to mention enhanced--is a full-time job with only part-time resources. You discover that it's not as simple as maintaining a database of reported or requested items. In addition, you must incorporate company policy about who is notified and when, what the workflow and the service levels are at each phase, who participates in prioritization, and who decides when an item can be closed. Whether this is done centrally through IT or at the individual project level, it requires ongoing support for new platforms, technologies, and features.

To tackle these and other issues, you decide to acquire one of the growing number of commercially available problem-tracking solutions. But what starts out as an innocent product acquisition often turns into a politically charged process, and yet another development and administration commitment. How does this happen, and what can you do about it? At least one experienced survivor has some observations and advice.

The search

Take the case of Tyler Barnett, the staff engineer at Lexmark International Inc., who's been responsible for the company's IT problem-tracking system over the past 10 years. Lexmark, based in Lexington, Ky., started out as IBM's printer division and now develops and manufactures printers, supplies, and supporting software. "Our system originally ran on an IBM VM mainframe, then it was ported to an X-Windows client/server application in the early '90s," he says. "The needs of the business have changed, and workarounds we used to keep the old application going couldn't be extended any longer, especially since [the app] was not Y2K compliant."

Barnett originally planned to develop a custom, Web-deployed system, but he realized that the requirements would keep growing and, with the Year 2000 looming as a nonnegotiable deadline, it was unlikely he would be able to complete the development and conversion to a customized system in time. Instead, Barnett assembled a shopping list of requirements and went out to buy a packaged, but flexible, solution.

His list included security, accountability, file attachments, and improved performance, as well as additional fields and more flexible workflow management. External vendor access to problem tracking was important to Lexmark. At the same time, the system had to be able to restrict access within certain groups such as some vendors or developers. Web access would be a major advantage, with the ability to broadcast information among the responsible parties instead of requiring them to drill down in search of it.

According to Barnett, "Whatever I did, I wanted to deploy a single system that could be used by everyone. I didn't want to glue disparate tracking systems together or maintain them. Yet, management also wanted to see everything from one viewpoint." Obviously, this was quite a challenge.

The solution

Eventually Barnett chose TeamTrack by TeamShare Inc. of Colorado Springs, Colo. But while making the product selection might seem like the conclusion of the process, it was actually only the beginning.

"We went into production after only two months," Barnett says, "but that [timeframe] was driven by business requirements. I'd have stretched it a few months longer and put more research and thought into the rollout." What he discovered was his company's processes didn't necessarily map to the ones contemplated by the product. The "one-issue, one-person" paradigm simply didn't fit the Lexmark business model, and neither did the Web-page notification scheme. Barnett redesigned the e-mail interface to run as an outside process, and he's still working on a group-based workflow approach.

Some degree of customization is inevitable, so plan for it. If you expect to "install and go" out of the box, you will be disappointed.

Barnett's experience is more the rule than the exception. A packaged product cannot, as a practical matter, accommodate the wide variety of ways and means in which companies manage their development processes, not to mention the technical landscape in which it's installed. Some degree of customization is inevitable, so plan for it. If you expect to "install and go" out of the box, you will be disappointed.

The results

For Barnett, the good news about buying TeamTrack and investing in the customization is that it was worth the effort. "We have enjoyed significant positive impacts from the new system," he notes. "The file attachments capability fixed a time-lag problem by getting failing test cases into the hands of the developers. Previously, we had to describe where the test case lived and how to get to it, and we lost a lot of time because of this."

Reporting and analysis have also been enhanced. Instead of exporting the data into spreadsheets and creating their own charts and graphs, the new system provides users with built-in reports as well as enhanced spreadsheet interfaces. Search times are dramatically reduced with a relational database, and Barnett expects to see a reduced client-support workload, since the new system uses HTML/Javascript instead of X-Windows, which is harder to support.

The light at the end of Barnett's tunnel is when he completes the training for about 30 administrators in the weeks ahead so they can begin to determine their own area's destiny. At that point, he can transfer daily administration of the system to administrators and concentrate on server support, maintenance, and policy.

The lessons

As Barnett's experience proves, acquiring a packaged product for issue and problem management is a smart move, but not a free ride. You'll probably save time by keeping the old system patched and glued together while you spend time gathering requirements and finding a product that meets your company's needs. Then you will still have to incorporate the nuances that make your company's internal processes unique.

So while the IT department may never be free of maintaining and supporting an issue- and problem-tracking system, daily administration can be given to the project groups, which accomplishes the ultimate goal: to put project groups in control of their own destiny. And, of course, you will also reap the benefits of improving your processes.

How to Achieve Effective Test Automation

by Linda G. Hayes

Originally published in Datamation magazine, February 1999

For all the cautionary horror stories about shelfware in test automation, there are success stories that remind us why we keep trying. This is one of them.

Harbinger Corp. is a worldwide provider of electronic commerce software, services, and solutions, specializing in the traditional electronic data interchange (EDI) field and trying to establish itself as a provider of secure Internet commerce tools and services.

Headquartered in Atlanta, Harbinger boasts more than $120 million in revenues, more than 1,000 employees, and almost 40,000 customers worldwide. It needs to support customer whose machines range from desktops to mainframes, which poses a testing challenge for covering not only functionality but also interplatform compatibility.

Only two years ago, testing was performed part-time by an employee who was also responsible for software distribution. However, growth quickly rendered this impractical, and part of the funds expended for acquiring two new companies was earmarked to invest in quality. Karl Lewis, manager of software process and methodology, was put in charge of leveraging this investment to increase the coverage and consistency of the test process.

The plan

Lewis quickly identified test automation as a primary objective. With five different server platforms to be tested and over 3,000 individual tests to perform, automation was the only practical means of getting the job done. Aware that he didn't have the in-house expertise to automate and wary of the potential learning curve, Lewis selected a local consulting firm with expertise in automation to assist him with tool selection and implementation.

No matter how elegantly designed or developed, a test script built without application understanding is worthless.

One of the factors in his selection of a consulting firm was the proposed implementation approach. Instead of writing test cases as "scripts," or programs, Lewis preferred using a table-based technique where the test cases are stored as data parameters that are then executed by a script framework. This shields those writing test cases from the scripting language, thus reducing the learning curve and enabling ease of test-case maintenance.

Further, unlike most companies that are testing the newest graphical environments, Harbinger relies primarily on a text-based, command-line interface for invoking its functions. This characteristic constrains the tool selection process, but it simplifies the implementation process, as text interfaces are inherently less complex than graphical ones.

With the implementation approach and test tool in place, the project was launched.

The project

One of the first--and most time-consuming--steps was to teach the consultants the Harbinger EDI product, which is quite powerful and complex. Although the plan was to acquire automation expertise from the consultants, it was actually more difficult to transfer the Harbinger product expertise to them. Harbinger's Karl Lewis managed the project, and although he eventually hired a new full-time tester to take over the automation effort going forward, he learned the hard way that he did it out of sequence.

"If I had it to do over again," says Lewis, "I would have hired and trained my new tester on our products first and then let that person learn automation from the consultants. After all, the consultants were temporary, so the expertise they gained on our products was lost."

This lesson is learned too often by too many. Automation tends to focus attention on the details of the test tool and the accompanying skills, which may eclipse the importance of the critical skill that makes a good tester: domain expertise. No matter how elegantly designed or developed, a test script built without application understanding is worthless.

The results

The automation project at Harbinger was a success. The table-based approach proved to be flexible enough to allow a single set of transactions to be executed against multiple servers by simply changing switch settings. Test coverage improved five-fold. When Harbinger acquired and integrated yet another product line, all of the test development tools were in place.

Another important advantage is that other areas of the organization are able to contribute to the test library. Customer-support personnel created test cases to verify program temporary fixes (PTFs), and the developers are able to automate their nightly build process. Harbinger's Lewis attributes this to the table-based approach as well as to a "make-test" utility that automates much of the creation of the test cases by populating a template with the variable parameters.

The future

Succeeding once is rarely the real problem with automation: one-project wonders abound. The real challenge arises when there are changes either in personnel or the application. "Turnover is tough," says Lewis. When you only have one dedicated tester, automation continuity is difficult to maintain. Application changes can be functional or platform-related, each with its own ripple effect.

Karl Lewis' experiences at Harbinger demonstrate a valuable lesson: Successful automation of testing requires more than elegant test scripts; it requires application understanding, too. It only makes sense that you must start with effective tests before you can make them efficient through automation.

Coder's Conundrum

by Linda G. Hayes

Originally published in Datamation magazine, January 1999

If problems are found early in the software development cycle, they're easier to correct. So reward your developers for writing lines of error-free code.

Like any QA bigot, I can be critical of developers who seem to favor speed over quality when it comes to tracking bugs and testing during development. But, as in most cases, there are two sides to this story, and once you understand each perspective it makes the coder's conundrum clear.

In a nutshell, programmers are not trained, scheduled, or rewarded for testing. Testing is not taught as part of the computer science curriculum--it's a distinct discipline from both design and development. Schedules rarely allocate time for unit or integration test, and since problems are seldom tracked by developers, there is no reward for delivering well-tested code.

Coder's conundrum unraveled

Create a culture that values quality by implementing a few simple changes.

- Teach testing: Educate your coders about unit and integration testing.
- Allocate testing time: Allow more time for developer testing--it's worth it in the long run.
- Track errors: Save time and rework by tracking who creates the bugs.
- Monitor productivity: Don't just measure developer productivity by lines of code, measure it by lines of error-free code.

- Reward: Recognize and reward those who produce the cleanest code. Add quality objectives to their annual reviews.

"I'm measured by what I produce" in terms of lines of code, says a seasoned developer in a mutual funds company, "not by the number of bugs" in the code. "My schedules are always aggressive, so if I take extra time to test more thoroughly, I will be penalized."

How does this happen? And, more importantly, what can you do about it?

Say what

Before we can expect developers to unit or integration test, we have to define what they are and explain it to them. Take the time to answer questions such as:

- What constitutes a unit test? Where do integration tests begin and end?
- Must each developer verify all classes of input and output, valid and invalid, as well as all error handling?
- Is 100% coverage required? Is it reasonable?
- Is it measured by lines of code, or must it include all possible pathways and states?

While it may not be practical to launch a full-scale developer's education and training program, small steps can make a big difference. For example, one financial services company developed a simple unit test template that called for a list of each of the categories of input and output, the rule(s) that affected them, the selected test values, and the expected results. The developer completed the template for each unit, then--and this is key--it was reviewed by a cross-functional team of user representatives, QA staff, and development managers.

This basic process yielded many benefits. Incomplete or missing requirements were uncovered, thereby improving the programmer's understanding. The test templates brought a consistency to the process and provided continuity to new developers whose task is to maintain existing code. Perhaps most importantly, quality increased and the cycle time to complete projects declined...but it wasn't free.

Say when

The cost of better unit and integration test is, of course, time. Unless time is allocated and dedicated, testing won't get done.

In the case above, the formalization of unit testing added about 12% to the development schedule but decreased overall cycle time by 30%. Why? Because the problems were found early, so they were easy to correct.

So if you expect your developers to do a credible testing job, allocate the time for them to do it.

Say thanks

I have always been bewildered by managers who refuse to track errors back to the individual developers. I know all the reasons--we don't want to play the blame game, it will discourage developers from taking on complex tasks, it won't reflect the variances in difficulty among modules, and so forth--but I don't buy them. Ironically, testers often know who the culprits are, but their own managers usually don't.

If you really believe that it's cheaper and faster for developers to find problems than it is for testers or customers, then you should encourage them to do it. Measurement is a great motivator. Recognize and reward those who produce the cleanest code. Add quality objectives to their annual reviews. Create a culture that values quality.

While I don't advocate the use of error counts to hold anyone up to scorn and ridicule, I do think they are excellent indicators of the need for training or discipline. At one global consulting company, for example, the test group for an outsourced application development project determined that almost 90% of the problems found during testing originated with a single programmer. Upon investigation it became clear that the individual had misrepresented his skills in order to receive, in effect, on-the-job training.

Whether the right decision is to remove the person or train him is not what's important: what matters is that it's the wrong decision to simply ignore the numbers and thus pay the penalty down the line in support and rework costs.

Say so

So instead of complaining about coders, come to their rescue. Help them help you. Teach them about testing, give them the time to do it, and reward them when they do. You'll probably be surprised to discover that they'll participate enthusiastically, since I've never met coders who didn't take pride in their work...once they understood what it was.

The Data Dilemma: Test, Don't Experiment

by Linda G. Hayes

Originally published in Datamation magazine, December 1998

Make your test-data strategy an investment instead of an expense.

Unless you're ready to wear a white coat, carry a clipboard, and run a laboratory, you have to get control of your test environment. After the software itself, the data is the most critical element in gaining control. A basic tenet of software testing is that you must know both the input conditions of the data and the expected output results to perform a valid test.

If you don't know either of these, it's not a test; it's an experiment, because you don't know what will happen. This predictability is important even for manual testing, but for automated testing it's an absolute necessity.

Extract test data from production, seed it with known data then archive it for reuse.

Gathering test data, in fact, is usually more than half the effort in creating a stable test bed and the most problematic by far. For example, in an extreme case, testing an airline fare pricing application required tens of thousands of setup transactions to create the cities, flights, passengers, and fares needed to exercise all of the requirements. The actual test itself took less time than the data setup.

Let's look at two opposing strategies for acquiring and managing test data and consider the possibility of using the best of both.

Pulling from production

The most common test-data acquisition technique is to pull data from production. For an existing application, this approach seems both logical and practical: Production represents reality, in that it contains the actual situations the software must deal with, and it offers both depth and breadth while ostensibly saving the time required to create new data.

There are at least two major drawbacks, however. First, the test platform seldom replicates production capacity, and so a subset must be extracted. Acquiring this subset is not as easy as taking every nth record or some flat percentage of the data: The complex interrelationships between files means that the subset must be internally cohesive. For example, the selected transactions must reference valid selected master accounts, and the totals must coincide with balances and histories. Identifying these relationships and tracing through all of the files to assure that the subset makes sense can be a major undertaking in and of itself.

The second major drawback of this approach is that the tests themselves and the extracted data must be constantly modified to work together. The most basic tenet of testing says we must know the input conditions for a valid test--in this case, the data contents. Each fresh extraction starts everything over. If, for example, a payroll tax test requires an employee whose year-to-date earnings will cross over the FICA limit on the next paycheck, the person performing the test must either find such an employee in the subset, modify one, or add one. If the test is automated, it too must be modified for the new employee number and related information.

Thus, the time savings become illusory because there is no preservation of prior work. All effort to establish the proper test conditions is lost every time the extracted data is refreshed.

Starting from scratch

The other data acquisition technique is to start from scratch, in effect reconstructing the test data each time. This approach has the benefit of complete control--the content is always known and can be enhanced or extended over time, preserving prior efforts. Internal cohesion is assured because the software itself creates and maintains the interrelationships, and changes to file structures or record layouts are automatically incorporated.

One technique for acquiring data allows for steady, constant expansion of test data.

But reconstructing test data is not free from hazards. The most obvious is that, without automation, it's highly impractical for large-scale applications. But less obvious is the fact that some files cannot be created through online interaction: they are system-generated only through interfaces or processing cycles. Thus, it may not be possible to start from a truly clean slate.

Another compelling argument against starting from scratch is that data created in a vacuum, so to speak, lacks the expanse of production. Unique or unusual situations that often arise in the real world may not be contemplated by test designers. Granted, this technique allows for steady and constant expansion of the test data as necessary circumstances are discovered, but it lacks the randomness that makes acquiring data from production so appealing.

The best of both worlds

As in most cases, the best solution to the problem of gathering test data is neither extreme. The ideal scenario may be to begin with an extract from production, seed it with known data, then archive it for reuse. This solution provides a dose of reality tempered by a measure of control.

This is the strategy adopted by a major mutual fund company to enable test automation. Without predictable, repeatable data there was no practical means of reusing automated tests across releases. Although much of the data could be created through the online interface, such as funds, customers, and accounts, other data had to be extracted from production. Testing statements and tax reports, for example, required historical transactions that could not be generated except by multiple execution cycles. So, the alternative of acquiring the data from production and performing the necessary maintenance on the tests proved to be less time-consuming. Once the data was assembled, it was archived for reuse.

It's still not easy. You must surmount the cohesion challenge, assuring that the subset you acquire makes sense, and you must have an efficient means of creating the additional data needed for test conditions. Furthermore, you must treat the resulting data as the valuable asset that it is, instituting procedures for archiving it safely so that it can be restored and reused.

Although a popular and sensible concept, reuse brings its own issues. For time-sensitive applications, which many if not most are, reusing the data over and over is not viable unless you can roll the data dates forward or the system dates back. For example, an employee who is 64 one month may turn 65 the next, resulting in different tax consequences for pension payouts. Luckily, in many cases exactly such capabilities were left behind by Y2K projects.

Furthermore, modifications to file structures and record layouts demand data conversions, but this may be seen as an advantage since--hopefully--the conversions are tested against the test bed before they are performed against production.

As you can see, the data dilemma is not easily solved no matter which route you choose, and the amount of time and effort will be significant in any event. What's important to realize is that your test-data strategy is integral to the integrity of your overall test process, and without a thoughtful and consistent approach it will be an expense instead of an investment.

Adopt a Winning Strategy

by Linda G. Hayes

Originally published in Datamation magazine, November 1998

If you present risk assessment in terms that management can relate to, like loss of business, full-coverage software testing doesn't have to be just wishful thinking. Full-coverage software testing is possible; you just have to know which moves to make.

Planning for full software test coverage usually sounds like preparing only one strategy for a championship chess match.

The assumption that you will have the opportunity to completely cover all of the features and functions--both new and old--of the next software release sounds like planning to win a game without considering your opponents moves. It's wishful, if not downright naïve, thinking.

So why plan full-coverage software testing? For the simple, but compelling reason that you can't assess a risk you haven't identified.

Once you've identified the risks, however, you can properly deal with the contingencies and emergencies that routinely disrupt the best-laid plans. Only when you've mastered all of the standard chess openings, can you adopt a successful strategy.

Setting up the board

A large West Coast bank was developing a new version of the system that supports its loan officers in more than 600 branch banks. This system calculates effective interest rates, interest payments, and maturity dates, as well as produces the host of documents and disclosures required to initiate loans and collateralize them.

Because loans are the lifeblood of banking, this system is crucial, and a senior manager was appointed to oversee the test effort. With extensive lending experience as well as development savvy, she was in a position to understand the inherent risks in both activities.

What she wasn't familiar with, however, was software testing, and so she proceeded in blissful ignorance to plan for 100% test coverage of all types of loans, collateral, interest levels, and payment types. She even brought her technical background to the fore by acquiring a test automation tool and developing a complete repository of test cases that exercised the entire system, as she understood it. When the software arrived, she was ready.

Opening moves

The first build had all of the warts to be expected: the installation process left out some critical files, the configuration was incompatible with the platform, and there were protection faults from time to time. But subsequent builds made steady progress and eventually the software was stable enough to withstand the execution of the complete test suite, revealing more subtle problems.

Because the bulk of the tests were automated, it was possible to establish a very crisp schedule: the complete test process required approximately 76 hours to run if the software was properly installed and configured, and if there were no hard crashes. Reviewing the printed output--the actual loan documents--took another two days of manual effort.

Attack

Build and test iterations continued, until the time finally arrived for the field test, which was to be conducted at a subset of 40 branches that maintained a parallel process with a combination of the old system and some manual procedures. After a 30-day burn-in period the system was supposed to go live across the coast. Due to the high number of branches, this was to be accomplished through a marathon upload process whereby the local branch servers would receive the software over the weekend.

The field test uncovered yet more issues, as was to be expected, and these were corrected as found. Weekly builds and test iterations kept the 40 branches busy, but their backup processes protected them from any undue consequences. That is, until the final week before the scheduled go-live date.

During this week the system was to produce calculations and reports for the month-end posting of accrued interest from the new loan system. The reports and their calculations were erroneous, but the cause was not immediately clear. As the deadline approached, the development team became more and more frantic to uncover the problem and get it fixed. Finally, late that week, the culprit was found and fixed. It was time to test and then to ship.

Check

But an interesting thing happened when the software arrived for the final test. The test manager looked at her watch, looked at the calendar, and concluded they would not make the Sunday upload window. She said so, provoking a storm of protest.

All of the usual suggestions were trotted out. Add more people! No need, she said, it's mostly automated. Work overtime! The 76 hours are around the clock already, she pointed out. Automate more! The documents must be personally verified because the scanned images were too sensitive to slight variations, she explained.

And so it went, until the final, inevitable option made its appearance: test less.

Check mate in two moves

Ah, she said, no problem at all. If making the date is more important than making sure the system is ready, we can certainly do that. Here's how: we will leave off testing all car loans, since they are the most complex and require the most time. That will shave off the extra time and we can make the date, just barely.

The reaction was swift: What? You can't skip car loans--they are a huge part of our loan volume, they need special documentation, the collateral has to be properly secured, the title restricted, and so on and so on. It's just too risky!

Hmm, she said. Then we'll have to forget about second mortgages. That's the only other class of loan that has enough time in the test schedule to make the difference we need.

The test manager got same reaction: Are you crazy? Mortgages represent huge amounts of money, even more documents, and besides, we're marketing them like mad right now. You can't possibly gamble with those!

And on it went. By the time the dust had settled, guess what? The go-live date was delayed.

Game over

So what made the difference? Simple, it was an assessment of risk, and the lesson is deceptively straightforward. If you don't plan to test the entire system, you don't know what you are leaving out; if you don't know what you are leaving out, you don't know what's at risk. And, if you don't know what's at risk, you can't weigh it against the perennial risk of missing the schedule.

For the bank, the risk of delaying the roll-out of its new loan processing system was weighed against the risk of failed system functionality. The result was that a fully tested--and consequently a fully operational--system was more important than a brief delay in deployment.

If you don't plan to test the entire system, you don't know what you are leaving out; if you don't know what you are leaving out, you don't know what's at risk.

In most cases, the test manager would have insisted that they could not test the entire system, and management would have tut-tutted but proceeded to cut testing in order to meet the deadline. So when critical functionality went awry, the test manager would have been denounced for allowing something so important to go by the wayside.

The key in the bank's case was that the test manager knew enough about the business to put the risk assessment in terms that management would relate to: money. If loans are processed incorrectly, business is lost.

The bottom line? Full-coverage software testing is possible; you just have to know which moves to make. Test cases may have no meaning to a bank, but loans do.

Fractional People

by Linda G. Hayes

Originally published in Datamation magazine, October 1998

Software releases are cyclic by nature. So where do you find qualified testers when they're needed, and where do they go when testing is done?

Understanding the dynamics behind the ratio of developers to testers is only half of the problem involved in developing, maintaining, and enhancing applications (see last month's column, "The irrational ratio"). The other half is making the ratio work organizationally. To do this, you need to answer two key questions: Where do the testers come from when you need them? And where do they go when you don't need them?

The three most likely strategies for shoring up test resources are automation, consultants, or borrowed resources.

The cyclical nature of software releases means that the demand for testing waxes and wanes, depending on the phase and type of development going on. This fluctuating need is the crux of the quality problem in many companies. Since testing resources can't afford to be idle, testers often are given other responsibilities to fill in the downtime. As a result, it's not uncommon for companies to have fractional headcounts in testing. But this strategy breaks down during periods of high testing demand--large or even multiple releases may demand more than the other half of someone's time.

If your company is large enough, it may be that testers can be a shared resource across multiple applications, migrating like fruit pickers (no offense intended) from one area to the next. This approach only works well, however, when there is some semblance of centralized control over testing that is able to apportion and schedule resources across development projects.

Companies that are too small to have multiple projects or where there is no central mechanism to allocate test resources are in a bind. These firms can't justify enough full-time resources to meet peak demand, but they also can't get the job done with part-timers. So what do you do?

Software, services, or scavenging

The three most likely strategies for shoring up test resources are automation, consultants, or borrowed resources. Although the initial investment in automation is substantial, it can take up the slack by executing regression tests automatically. So modification and enhancement releases can be tamed somewhat by reusing the same test cases over and over. The downside to automation is that it's an advanced skill that is hard to keep current on a part-time basis.

Consultants may be called in, either to provide people or automation expertise. The benefit of using consulting services is that it's a variable cost, used to backfill the peak of the cycle or to shore up the expertise of internal personnel. The drawback, besides cost, is that temporary resources have a learning curve on the application under test and there is little or no guarantee that you can acquire the same consultants from one release to the next.

Automation also creates a continuity problem. Consultants may be able to jump start an automation project, but if there is no internal resource trained to take ownership when the consulting engagement is complete, the scripts will fall into disuse as application changes require maintenance.

Borrowing resources from other areas in the company is the third option. End users, customer support personnel, developers, trainers, technical writers, even customers may be pressed into service. The plus side of this approach is that these sources probably bring application expertise that outsiders can't, and they have a vested interest in the quality of the product. The negative side is that these contributors may not have the testing skills that make their efforts effective or repeatable.

One common way of recruiting customers is through an "early release" or "beta site" program. In this scenario, customers take delivery of software that is admittedly not fully tested, with the understanding that they will report any problems they find. Microsoft has employed this technique liberally, yielding thousands if not millions of testers for their products. It's also a viable solution when the development company cannot possibly own, let alone test, all possible platforms and combinations.

The key to a successful early release program is to tightly manage the error reporting process and assure that the schedule permits enough time to address the issues raised before the final release is due. It's not a free ride, however, as even--and maybe especially--early release customers will require support. If the software is substantially unstable, this approach may backfire by generating an excessive level of support requirements.

Rationalizing the ratio

Unfortunately, there is no magic answer to resolving the resource ratio, but there are some tricks and techniques that can make a difference:

Plan your test: Put some thought into what you must cover in your test and what you can afford to leave out. Write it down. Setting the scope can guide future testers so that they don't wander into the weeds, spending time and wasting effort. Although the creative instinct is often credited with uncovering problems, unplanned tests may reveal obscure bugs while missing major ones.

Document your tests: The simple (but not easy) act of reducing your test cases into writing can work wonders by making them transferable and repeatable. Borrowed or temporary testers can be instantly productive if they have documented tests to follow, and you will have a much better grasp of what was tested and how.

Dumb-down your tests: Take the expertise out of the tester and add it to the test case. In other words, write test cases that are extremely detailed; assume nothing. Instead of saying, "Create an order and verify the pricing," say, "Choose new order from the order meu, then enter the Customer Number 12345, click OK...." You get the picture.

Standardize your tests: Don't allow test cases to be organized at the whim of the individual. It takes more time to create tests from scratch than to fill in a template, and trying to follow differing styles when executing tests is not efficient. Pick a format, build a template, and stick to it.

If you follow these basic rules, you will be better positioned to press temporary testing resources into effective service or to employ automation when it's appropriate. And, hopefully, arrive at a rational way of managing the developers-to-testers ratio in your company.

The Irrational Ratio

by Linda G. Hayes

Originally published in Datamation magazine, September 1998

While there's no hard and fast rule for determining the best mix of developers and testers, the secret is to know when and why to adjust the ratio based on the type of development and the phase of the project.

Do you remember learning about irrational numbers? I always pictured them as being somehow unreasonable, numbers you had to coax or cajole into behaving rationally.

Actually, they are numbers that never end--no matter how far you carry them out--like pi. In that sense, the term is particularly apropos for describing the proper ratio between developers and testers, because the answer keeps changing the longer you look at it.

We have all learned the hard way that adding more people does not always add more productivity.

Not a conference or class goes by that the question is not asked: What's the magic ratio? Some breathlessly report that they heard Microsoft has a one-to-one ratio, while others abjectly confess that their companies have 10 or more developers for every tester. So what's the best answer? That's easy. It depends.

The ideal ratio between developers and testers depends on the type of development and the phase of the project. In some cases, developers should outnumber testers, but in others--believe it or not--the testers should outnumber the developers. The secret is to know when and why to adjust the ratio.

Planning and design

In the earliest stages of developing a new application--that is, one being written from scratch--developers should outnumber testers. Although test planning and design can and should commence during the planning and design phases of the development project, the fact is that there is simply more work to be done by developers. Until the design stabilizes, which in most cases happens during coding, testers can at best define and prepare the test environment and process.

The ideal ratio at this stage is probably one tester per team, or about one for every four or five developers. Most applications are built with a team this size; especially large applications might have multiple teams, but each team typically stays small. We have all learned the hard way that adding more people does not always add more productivity.

There is an exception, however. In some cases, new applications are really just modifications of existing ones. For example, a retail data analysis firm developed six applications that were simply just special versions of the same system customized for different customers. The same application in other cases might be ported to execute on multiple platforms. The effort in these situations is more like maintenance and enhancement, so the rules are different.

Coding and testing

As a new application completes the coding phase and moves into test, the ratio should start to shift in favor of testers. In fact, a ratio of one or two developers per tester is probably realistic.

This happens because the testers now have enough information to design, develop, and execute test cases. It takes time to review the application's intricacies in order to design the proper tests, then more time to develop the tests, and then even more time to execute the tests and document the results. Then it takes yet more time to work closely with developers to resolve issues. The iterative retesting of corrections and changes adds substantially to the testing workload as the application nears release.

Developers, on the other hand, start to wind down in this phase, primarily addressing issues that arise, but--hopefully--not adding anything new. In fact, developers often start looking for a new challenge as the application moves inexorably toward that programmer purgatory known as maintenance.

Maintenance and enhancement

This is the phase where the ratio usually gets out of whack. If I tell you that you need more testers than developers, you'll think I'm crazy. But it's true.

Testing ratio tricks and techniques

If you follow these basic rules, you will be in a better position to effectively use temporary resources and/or employ automation in testing when it's appropriate. These tricks and techniques may also help you achieve a more rational method for managing the ratio of developers to testers in application development.

- Plan your test
- Document your tests
- Dumb-down your tests
- Standardize your tests

Look at it this way. Let's say you have a 10,000-line application to which you're adding some features as well as fixing some problems, and let's estimate that you will add or change 25% of the code. So, from a development point of view, you have a quarter of the development effort originally required to create the application.

But from a testing point of view, you still have 100% of the code to test, because of the well-known phenomenon of unexpected impact from changes. An apparently tiny, isolated code change can have far-reaching effects. Examples abound, but the most famous is the single-line (some say single-character) aberration that brought down AT&T's network in January 1990 and cost the company and its customers more than a billion dollars.

So the rule is that all features of an application, whether new or old, are tested every time there are any changes, no matter how theoretically minor. Testing old features is called "regression testing," which refers to the problem of a prior capability being negatively affected by new features or modifications.

A similar situation arises when a new application is created by modifying an existing one, or by porting one to a new platform. From a development point of view, the level of change may be only a fraction of the original code, but from a testing point of view, it's a completely new application.

What this all means is that it takes fewer developers to enhance or maintain an application, but the same number of testers as when it was brand new. In fact, it may take even more testers, since successive releases of applications almost always add new functionality without taking any away.

Testability Standards for Automation

by Linda G. Hayes

Originally published in Datamation magazine, August 1998

The insertion of a test tool between the tester and the application shifts the level of their interaction, making them integrated stages of a continuum.

Over the transom, applications development and testing should be history. Developers should be building applications with automated testing in mind from the beginning. By focusing on how testing tools work and how they interact with applications, developers can save themselves a lot of time and aggravation and save their employers a lot of money.

The most powerful way to integrate software development with automated testing is to exchange the application information that is needed by the tool. Virtually all test tools that support object-oriented testing require that the application map be defined or captured; this map details the windows and objects that make up the user interface, including their class. This map information is located within most graphical applications as the resource file; for mainframes, this is usually a screen map. Whatever the form, it is used by the application and can be shared with the test tool.

Rapid development with manual testing is a formula for failure--failure to hit the market window or to deliver a solid product in anything resembling a timely fashion.

What this means is that development provides not only the software and release notes to testing, but also the map file that will be used by the test tool. This allows the test organization to both accelerate the initial automation effort and to quickly spot changes, such as new windows or control classes, that require modifications to the automated test library. It creates a bridge between development and test.

How test tools work

Test tools can interact with an application in one of two ways, commonly known as analog or object-oriented mode. In analog mode, the tool generates events such as mouse clicks or keyboard entry without regard to their context. Clicks are performed at specified row and column pixel coordinates, and keyboard entry occurs wherever the cursor is located. For example, a recorded mouse click might generate a statement that says "Right Click at 653,122" --which is of course ambiguous as to what, exactly, this click accomplished in the application.

Although widely and easily available, this technique is of limited use because it has no context and thus is impossible to read and maintain. The slightest change, including different display monitors or video cards, will affect the script playback. The inability to detect context also cripples verification of results and recovery from errors.

Object-oriented mode, on the other hand, is context-aware. The test tool actually insinuates itself into the messaging layer between the window manager and the application, and can thus address individual objects or controls. Instead of a click at a location, the script can generate the selection of a particular item out of a list, for example, or from a menu. This technique is only available, however, in those cases where the tool can recognize the class of object(s) within the application.

With the proliferation of new class libraries, both purchased and user-defined, the odds are getting more and more remote each passing day that the test tool will readily recognize all of the application objects. So the question arises, what to do?

Testing for testability

The best line of defense is for the development organization to adopt testability as a design requirement. This means that, all other things being equal, standard object classes will be used instead of nonstandard ones. If nonstandard classes are needed, then every effort should be made to select a class that is supported by the test tool. Many tools offer extensions for special classes.

Class support is easy to determine. Simply place the tool in "capture" or record mode and interact with each type of object class, then review the resulting script. If you see coordinate clicks and keystrokes, you've got a problem. To solve this problem, the first step is to see whether the object class can be mapped to another, standard class: Called "aliasing" or "class-mapping" by most tools, this technique allows the tool to treat a nonstandard class as a standard one. If that doesn't work, you may have to either instrument the source code with a special hook for the test tool or write special functions yourself in another language, such as C. This is the bad news.

The good news is that development practices are moving toward standards, such as Microsoft's ActiveX, that will help test tools deal with new object classes more consistently.

Testing for consistency

Once you get over the compatibility hurdle, if you do, then the next line of defense is consistency. This is a simple idea: The application should appear and behave consistently from one area to another. Consistent behavior means that the test tool can encapsulate activities into easily developed and shared functions, speeding development and easing maintenance.

In practice, however, it's a little more complicated. Multiple developers working on a single application invite the opportunity for each to do things his or her own way, and there is seldom time or attention paid to the code inspections and reviews that screen variations out. Standards usually exist, in some form or another, but enforcement is usually lax to nonexistent.

The most obvious layer of consistency is in terminology. The push button that accepts information on a window, for example, should be called the same thing from one to the next: Whether it is "OK" or "Enter" or "Accept" or "Yes" or a green check mark is not important, only that it is one or another but not several. Window titles should either display the open file name or not. Again, the key is not so much what you choose as the fact that you choose, and then follow the determination.

It gets a little more complex when you start dealing with keys. Accelerator and function keys should likewise behave the same across the application. For example, Tab should always move focus to the next object, the Escape key should always exit the active window or dialog, or F1 should always summon Help. Any company that ignores these conventions should be noted.

Aside from simplifying test automation, consistency also delivers usability. Inconsistent application behavior confuses end users and makes the application more difficult to learn and use. Consistency, on the other hand, makes it easier to document, train, and support applications.

Integrated stages of a continuum

Although aligning development practices with test tool requirements may sound like the tail wagging the dog, consider the fact that development tools and languages have greatly accelerated the rate at which functionality can be created. Components can be purchased and incorporated that instantly add complex capabilities, and code from one application can be tweaked and inserted into another.

Unfortunately, this speed only creates a bottleneck in testing unless technology can be brought to bear. Rapid development with manual testing is a formula for failure--failure to hit the market window or to deliver a solid product in anything resembling a timely fashion.

So instead of viewing testing as an activity separate and distinct from development, view them as integrated stages of a continuum, both of which must perform optimally for the project to succeed.

Checkpoint Charlie: The Hand-Off from Development to Test

by Linda G. Hayes

Originally published in Datamation magazine, July 1998

Speed doesn't always count, especially when it comes to allocating testing time in the development process.

One of the most confusing aspects of the software development lifecycle is that there are separate phases known as "development" and "test." Calling the two phases "development" and "test" implies that developers only develop and that testers only test.

Nothing could be further from the truth, however, and this misconception causes untold cost and confusion throughout a product's life.

The concept of build verification is simple: It's the hand-off from development to test. Any build--or release or update or whatever--that is delivered for testing must first pass this test.

In fact, some companies subscribe to the view that speed is all that counts in development, so thorough testing is not only unrewarded for developers, but perhaps even penalized. What this stance ignores, however, is that speed is measured by overall delivery: Time supposedly saved during development will be repaid--with interest and penalties--during test and production.

Emphasizing speed in the development process means there's little or no time allocated to testing during the development phase--and inadequate time after development. Test schedules end up being undefined or poorly defined. And those poorly defined schedules lead to vague development testing processes.

The more vague the development testing process, the more difficult it is for the testers to define and defend their boundaries. A test group that is constantly trying to atone for missing or inadequate development testing cannot meet its own testing responsibilities, which are already substantial if not overwhelming.

So what type of testing should developers do, and how do you know when they're through?

Unit test

There is a substantial--and critical--amount of testing that can only be done by developers, because only they have access to the information needed: the code. The most basic is called "unit" test, and it refers to the testing associated with a single module of code, such as a program. Unit test involves assuring that the code itself works as written-- that field edits accept and reject valid and invalid data, error messages are issued when appropriate, files are properly read and/or written, and so forth.

While most everyone agrees that unit test is necessary and belongs in development, the overwhelming majority of organizations allow this phase to be completely informal and subject to the whims of the developer. The criteria for when to submit a unit for integration can be anywhere from a gut feel to a successful compile to a stringently tested module, or anywhere in between. Unit test plans, let alone test cases, are rarely documented or even understood.

Yet this phase is literally the only point in the entire software lifecycle when the code itself is exposed. Only the developer knows the design and construction of the code itself: where branches occur, how errors are trapped, and what limits are imposed on arrays and other structures. In fact, for Year 2000 testing, the developers may be the only players in the entire project who know precisely where changes have been made as well as the only ones able to assure that all required date ranges are fully exercised for each modification.

The simple fact that testing is done by individual developers should not be a license to exempt it from any formal test process or review whatsoever. Only by clearly defining the testing roles of the development and test organizations, and specifying the hand-off between them, can quality be achieved. Thus, the hand-off of a unit of code into an integration or build phase should include, at an absolute minimum, a list of the conditions that were tested so that the test group need not repeat them and future developers can repeat and expand them.

Integration test

The next level of testing, integration, concerns itself with the merger of units into a functioning whole, whether it's a subsystem or complete application. Usually the integration process, also called the build process, produces an executable system. Where unit test relies on the construction of the code, integration test revolves around the architecture of the design: the boundaries between the units and how they interface.

Perhaps not surprising, but nevertheless disturbing, is the fact that this level of testing is also almost exclusively informal. Victory is often declared if the build compiles successfully. Yet the build process--especially in this new world of reusable objects--is fraught with opportunities for error. (See June 1998's Quality Quest, "Don't take anything for granted in component development.") Simple differences between the development and build environment can wreak major havoc with the final product: .dll files can be incorrect or missing altogether, platform configuration options can vary, data files can change contents, and a host of other variables can go awry. Tiny differences can cause a domino effect throughout the system.

Again, only the development organization is privileged to information about how units interact: which parameters, files, switches, and calls are passed back and forth, which objects are shared internally or externally, and what platforms are involved and where. Without an orderly integration test process, performed by knowledgeable system architects, further testing is continually at risk for functionality to be dropped or broken from one build to the next.

Thus, an acceptable integration test hand-off should include the list of configuration options and interfaces tested and the pertinent conditions for each.

System test

Although unit and integration testing are commonly understood to belong to development, the system test is where it begins to get hazy. Who does it belong to? What does it include?

In view of the perpetual fog around this test phase, I propose that it be redefined--and reorganized--as a build verification test.

The concept of build verification is simple: It's the hand-off from development to test. Any build--or release or update or whatever--that is delivered for testing must first pass this test. It serves as the ultimate integration test, assuring that all units and subsystems behave as a coherent whole when installed on the test platform as opposed to the development environment. Its purpose is to prove that the system has sufficient technical integrity to enable business functionality to be verified.

This type of test can be thought of as "inch deep, mile wide." It covers the broadest area of the system but only minimally, assuring that each screen or window is accessible and verifying that all files are present and available and that no modules or libraries or other components are missing, and so forth. Sometimes called a roll-call, walk-through, or smoke test, the system test phase seeks to establish that the software is internally sound and complete.

Although the system test phase should be defined by the test organization, and perhaps even executed by them because it requires the test environment, the formal test phase does not officially commence until the application has passed the developers' system test successfully. In other words, the developers' system test phase is the entry criterion into formal testing by testers.

The gateway

Without this gateway, or any objective measurement, there is no real means for determining when software is actually suitable for testing. The simple act of delivering software to testers doesn't mean that development is complete or that testing has begun. It's not uncommon--as many testers will testify--to get an early build or two that will not even install properly, let alone execute. The sticky part is that there is usually a limited amount of time allocated to test anyway, and when this is eaten away trying to stabilize the system, testers end up with twice the work, half the time, and all the blame.

This is not an indictment of development, but it is a condemnation of the absurdity that developers do not test. If a developer cannot be relied upon to test and deliver a solid, stable code component, the answer is not for testers to make up for it. The answer is to either retrain or replace the developer.

Don't Take Anything for Granted in Component Development

by Linda G. Hayes

Originally published in Datamation magazine, June 1998

Component-based applications development is an exciting idea with frightening consequences. On the exciting side, it helps developers create more powerful applications much more quickly through reuse. On the frightening side, it blurs the boundaries of applications, which may now span platforms and even companies, thereby creating boundless opportunities for problems.

Something that works today can fail tomorrow because of the installation of objects by someone else for something that had nothing to do with your application.

The potential for problems makes it important to understand this new development approach and how it affects the test process, because unless control is established at the beginning, it will probably never be regained. The bill comes due at the test phase, when the so-called benefits are eaten up dealing with competing components in the test environment, the tester's system, and even the end user's system.

What begins as a time and cost saver during development using reusable components can easily become a sinkhole of time and effort during the test phase and an endless expense throughout customer support. So pick your poison: reuse or redundancy?

Understanding objects

Components, also known as objects, are chunks of functionality that may be reused. Most personal computer users become aware of them as ".dll" files, or dynamically linked libraries, which are often reported as either "missing or incorrect" when a user tries to launch an application that depends on a .dll buried elsewhere on a hard drive. To get a feel for the number of these babies running around your system, do a search for the file type ".dll" and sit back. On my system I have more than 500 within the Windows subdirectory alone... and I'm not even a developer.

Some objects are simply shared within an application or module, but others can be shared across applications. Objects that are shared across applications typically follow a binary-level standard that dictates the form in which they are called and how they respond. The two primary standards are Corba, or Common Object Request Broker Architecture, and COM/DCOM, or Component Object Model/Distributed Component Object Model. Both standards provide for reusability and have rules about forward and backward compatibility. Unfortunately the rules are not always followed, and when they're not, things gets weird.

A puzzle project

Odds are, a developer is part of a development team whose disparate efforts must coalesce to form a whole application that can be shipped to a customer and installed. This is commonly referred to as the "build process," and it's becoming a puzzle project of imposing proportions when components are involved.

During the build process, all of the various bits and pieces of code must be gathered and compiled into an executable module, usually an ".exe" file. The object files are referenced in the executable, but are not always contained within it, especially if they are reusable. If they're not internal, objects must accompany the application or be already available in a known location for that application to run.

Object lessons

So will you be able to convince your company that reusable objects are a passing fad to be avoided altogether? A better strategy is to negotiate a truce. Here are a few suggestions:

(1) Do use an object repository to track and inventory components in a system, and make it known when something changes.

(2) Don't share objects externally by permitting applications development to depend on universal objects that are expected to be on user desktops; deliver what you need with the product.

(3) Do employ reuse in development, but use redundancy in testing and delivery through copies of components stored in private directories.

Source: PlugIn Datamation

Because each developer typically has his or her own development system, with all of its attendant configuration options and related files, it's no trivial effort to gather interdependent code from independent sources and have it behave as a unit. Any variances between the contents of the build system and the developer's system can potentially create problems, which they often do. In a recent project at a Big Six accounting firm, for example, simply defining the build process and all of its components required three months of trial and error.

Suppose, for instance, that you are a developer who needs date functionality in a program. You decide that incorporating a proven, robust object is faster and more reliable than developing a new one. You therefore purchase a date-manipulation component that only needs to be customized for your company's holiday schedule. This bit of programming means that as a developer, you only have to maintain the new holiday rules. Thrifty and nifty, right?

Well, maybe not. Continuing this example, assume the build process is conquered and the software is installed. Now suppose that a user--whether a tester or a customer--installs an application from another vendor for a different purpose and that this application is also date-sensitive. The developer of this vendor's application likewise acquired the same handy date routine module as you did, only he incorporated a newer version. When the user installs this application, the date routine .dll file of the initial application is overwritten. A warning message may or may not have been displayed, but even if one was, the user probably didn't have a clue that this message concerned multiple date routine files, and so proceeded with the installation.

Now, if you, the developer of the first date routine in this example, and the vendor's application developer played by all the rules, each version of the object would support all prior versions and each calling application would behave accordingly. The theory is that new versions do not remove prior functions, they only add new ones in the form of new interfaces to the object. Thus, a developer calling an object first checks for the expected functions within the interface--in this case the company-specific holiday schedule. If they're not there, he checks for prior ones, and so forth, as he looks for the appropriate functionality.

What it all means

Unfortunately, the reality is that objects are as dynamic as the rest of technology, and developers have enough to do without plowing through geological strata of interface functions. The result is that an object that's intrinsic to your application can be replaced, modified, and otherwise compromised by other applications as well as other people who you have never met and don't even know exist.

The nightmare of this reality sets in during test and deployment. An application that worked fine in the development environment becomes intransigent during test, and one that was subdued during test runs amok on the end user's system. Just to make it all a little more problematic, something that works today can fail tomorrow because of the installation of objects by someone else for something that had nothing to do with your application... or so you thought.

To a tester, these issues manifest in subtle ways.

First, the tester's system must either already have the necessary objects or they must be installed.

Second, the tester runs the risk that there are competing object versions on the test system.

Third, the tester may dutifully configure and manage the test environment, only to find that the end user has competing components.

The biggest problem, of course, is knowing where to look. It's one thing to know that a conflict exists, it is quite another to trace it to its source. The hierarchical structure of an object search can mean components are potentially found in more than one place.

Solutions you can use

So what can you do? The first and best insurance policy is a fierce dedication to maintaining a system inventory of all components that comprise an application or are required by it. Don't just keep track of the executable files, but include all related object files as well, in addition to their precise sizes, dates, and target directories. These are called object repositories, and there are commercial products that can help track them.

Installing a new application should be approached with care and suspicion, and any changes introduced as a result should be carefully documented. Without this vigilance, it will not be possible to know when and where a change was introduced. Implementing or installing new object versions should be treated as any other code change--the change should be documented and the impact should be tested.

Chances are, your test tool will react violently to the intrusion of new objects in the user interface, especially if they are nonstandard. While many tools allow nonstandard objects to be "aliased" or "class-mapped" to a standard class, this not only takes extra time, but it may not always work. If your tool doesn't recognize the object class, it will treat it using "analog mode," which means using mouse clicks and pixel coordinates that are only slightly less painful than dentist drills and root canals.

Ironically, the best strategy is to not share objects that are part of your application. That is, install any necessary components in a private directory, and don't rely on the existence of system or shared objects. True, this defeats the reusability factor in theory. But, in fact, all it does is sacrifice some disk space to redundancy. And what's a little redundancy among friends?

Time-Boxing Your Way to a Better Product

by Linda G. Hayes

Originally published in Datamation magazine, May 1998

Regardless of the boundless optimism with which we all enter projects, experience teaches that we will exit them in a Death March state of defeated exhaustion.

To paraphrase a recent conference speech by Tom DeMarco, the insightful author of books and essays on software development, most projects are like Greek tragedies: During the first half, it appears that man is in control of events; the climax reveals that events are in fact in control of man.

So it is with software development. At first we believe that the project controls the schedule, then we realize that the schedule was in control all along or, more likely, out of control.

This is how testing is routinely sacrificed. Not out of a conscious decision to give quality short shrift, but out of a series of compromises that conspire to delay the project...but not the deadline. At the end, quality is the only price left to pay.

What would happen if we became realists and just admitted, up front, that the schedule was the primary driver?

This concept, also known as "time-boxing," is a schedule-driven approach that treats deadlines as sacred, while resources and requirements are variable. When combined with the idea of a "rolling release," which comprises continuous incremental deliveries that are scheduled early and often, this approach can work wonders or terrors, depending on how you go about it. Over the past three years I have seen an increasing number of companies apply this approach to projects as large as an SAP implementation and to those as small as a departmental client/server application, with impressive results.

Working in boxes

A time-boxed project works like this. The final delivery date is identified first, with frequent intermediate releases preceding it. These may be one, two, or three months apart depending on the size and length of the project, but usually no more than three. Each release should introduce sufficient functionality to permit user interaction. The goal is to get started as soon as possible, so that feedback has a chance to be incorporated before the project is over. So, instead of a one-year development project, you would have four quarterly releases, each building on the last.

This approach forces requirements to be ranked by priority and order of implementation. Although the content of each release is aggressively planned in advance, it's understood that the release will ship on the designated day with or without all requirements. The schedule rules--but we knew that, anyway.

In this scenario, testing continues constantly. Instead of an end-of-the-line, frenzied event, it's a constant companion to development, providing verification of each build and successive feature or function. In the later releases, corrections are being introduced and tested as well.

The advantage of this approach is that it can produce a better product faster. Ambiguities and inconsistencies in the requirements are quickly flushed out when the software meets the cold light of user interaction. Rolling releases encourage constant refinement of requirements and continuous testing; this helps prevent projects from going too far astray or too late to test.

And, if the truth be admitted, even "completed" software development projects are never finished. Any product that is being used is subject to a constant flow of issues and enhancements as the users and business it serves evolve and change. Viewed in this way, the project never really ends: It may simply metamorphose into monthly releases.

Know where you're going

The flip side of this rosy picture is that discipline becomes sacrificed to speed. Although on the surface it might seem that focusing on the schedule encourages minimal planning and design, the opposite is true. The system architecture must be thoughtfully structured to enable the delivery of usable increments of functionality, providing a solid foundation quickly that supports future expansion without limiting it. It means you have to know where you're going before you start. If you don't, you'll degrade into producing rewrites instead of adding requirements.

One of the means for keeping tighter control on the software while enabling continuous change is to produce a software build at least weekly, if not daily. This requires the software to be assembled, available, and accounted for constantly, so that rapid responses to issues are possible. The longer the time stretch between builds, the greater the opportunity for confusion and mayhem.

It should go without saying--but we will say it anyway--that relentless project management and good software hygiene is essential. Constant coordination is needed between developers, testers, and users. And source and version control with complete configuration management is crucial among developers. A rapid-fire schedule is unforgiving, and frequent releases are uncompromising. It's either happening or it's not.

So simply declaring a time-box schedule and rolling release strategy, without the concurrent commitment to the supporting disciplines, will merely result in an interminable series of Death Sprints--each worse than the last.

Accelerate or be left behind

Let's admit it. Market opportunities and product cycles are compressing; the days of multiyear development projects are gone, along with the untold millions wasted on runaways and endless delays. In this, the time of I/Net speeds, our approaches to software development and testing need to accelerate or be left behind. Time-boxing and rolling releases are just formalizations of what the market is forcing to happen. The key is to do it voluntarily and consciously, instead of unwillingly and unknowingly.

The Confidence Game

by Linda G. Hayes

Originally published in Datamation magazine, April 1998

When it comes to software development projects, trade-offs are a fact of life. Customer demands, market pressures, resource constraints, and all manner of surprises are inevitable. The question is not how to avoid making compromises, but how to manage them.

Managing trade-offs is especially acute in testing. As the final phase, this is where the bill comes due for every concession made along the way. Unfortunately, this causes the test process to appear flawed: When a cycle planned for four weeks balloons into eight, it appears as though the testers just aren't getting the job done when, in fact, they are probably working twice as hard as originally planned.

One way to manage this conundrum is to implement an early warning system that focuses on the effects of trade-offs at the time they occur, allowing enough time for mitigation. Edna Clemens, manager of development services at Tandem Telecom of Plano, Texas, a Compaq company, set out to do just that. Her goal was to measure product readiness as an objective metric that was understood, adopted, and assigned by the entire core team for a product, and then used as a basis for making decisions before the situation became critical.

This measurement, dubbed the "Confidence Rating," expresses the cumulative effect of decisions affecting product completion as a percentage. Ninety percent confidence was set as the minimum desired level for shipment, and anything below that signaled a risk that had to be mitigated. The beauty of this rating is that it is derived weekly, so that progress is measurable throughout the process, thereby eliminating unpleasant last-minute shocks.

Clemens started the process by assembling the core product team--product management, development, test, documentation, and support--and retaining an outside consulting firm to lead them through the development process. Each phase was defined as to its entry and exit criteria, with emphasis on what qualified the product to be promoted to the next phase. This experience in itself was revealing, as it helped each team member to understand the needs and objectives of the others.

A follow-up session focused specifically on the two final test phases: integration and system test. For these, the input and output items were defined in more detail and each was assigned a number of points, the total of which equal 100%. The input items include the test strategy, which contains the priority of tests and the planned execution sequence; defined test procedures; known problem documentation; a verified software build; preliminary release notes; and the installation guide.

The outputs include such measures as the percentage of test cases passed and failed; the percent deviation from the test plan (which indicates problem areas); the percentage of test cases executed; the percentage of core ("drop dead" or essential) test cases passed for the final build; and the number of errors outstanding by severity.

An interesting effect of these calculations can be seen in the way that the percentage of test cases executed is derived. For illustration purposes, say that there are 100 test cases in the inventory. After the first week 20 have been executed, but only 10 of them passed. Since the 10 that failed have to be re-executed, they are added back to the inventory, so the percentage completion is not 20%, it's only 18% (20/110). If the failures mean that a complete suite has to be redone, then all of them--even if some passed--are added back.

What this reveals is that no matter how long or hard the testers work, if the software is not complete and tests are failing, then their workload is increasing. This increase translates into a schedule delay, which reduces confidence in the product readiness and drops the Confidence Rating.

The key benefit of this approach, Clemens has found, is that it has shifted the team's focus from making the schedule to completing the product. And that's the real name of the game.

When to Automate

by Linda G. Hayes

Originally published in Datamation magazine, March 1998

Test automation is such a seductive idea that it's tempting to apply it everywhere. The question is not whether you should automate, but when and where automation makes the best sense. Choosing your battles carefully is the best way to win the war.

Choosing an application

The first thing to do is select the right application. Applications that are too unstable require excessive maintenance of test scripts; applications that are too stable don't need enough testing to yield a productivity payback.

Next, identify the ratio of change from one application version to the next. On average, 25% of most applications change in a year. Then determine how many test iterations will be executed per release; three is an absolute minimum.

Remember that it usually takes five to 10 times as long to automate a test as it does to perform it manually. Also consider that a change to the application can cause twice as much or more maintenance in the test library because a single rule change usually requires several test cases. On the plus side, automation typically saves 80% of the manual effort--the remaining 20% is spent on setup, supervision, and results analysis.

Most test groups careen from one crisis to another, and trying to automate under combat conditions is fruitless.

Finally, do the math. Let's say it initially takes you six times as long to automate a single manual iteration, so your investment is 600%. Your application will have two releases per year of 15% change each, with three test iterations per release, and each release will require 30% of maintenance to the test library. You'll get an annual return on your test investment of six iterations at 80% per iteration (or 480%), less maintenance for two releases at 30% each (or 60%), so you will receive a net return each year of 420%. This means you'll be in the productivity payback zone by the end of the first year. Not bad, as investments go.

It's a team effort

Once you've identified a candidate application, make sure you have the right team. If the majority of your testers are migrant workers borrowed or kidnapped from other departments, the odds are you won't recoup the investment in the automation learning curve. Although some amount of fluctuation in staffing is inevitable because of the cyclical nature of testing, make sure you have a permanent, dedicated core team. Otherwise, the maintenance effort of the test library skyrockets. New testers must learn the application and the tool, as well as find their way around the test library before they can make any changes. So the maintenance effort can go from twice as long per application change to 10 times or more, thus eroding the productivity payback.

Even with the right application and test team, you must choose your timing. The biggest problem in testing is schedule squeeze. Most test groups careen from one crisis to another, so making a significant investment of time and effort up front is a practical impossibility, and trying to automate under combat conditions is fruitless.

The best time to get a time commitment is when the financial commitment to acquire the tool is made--the honeymoon period. Make a reasonable request to peel off two or three testers, or only one if that's reality, and have the testers begin the automation effort for the next release as soon as the most recent one is complete. Don't settle for part-time resources or expect workers to do their existing job and automate in their spare time. It doesn't work.

Be reasonable

The last caveat is simple: Don't expect miracles. If your test effort is random, undefined, and generally unpredictable, don't expect automation to save the day. Although tools are a great vehicle for establishing order and organization, they won't drive it. Get your act together before you memorialize it in scripts.

As a final note, realize that all of these calculations assume that you will only automate as much as you already test manually. The fact is that once automation is in effect, you open the test window and can get in more test cases per iteration and more iterations, which in turn increase coverage and improve quality. Quality, of course, yields a payback that has nothing to do with testing. It has to do with increased customer confidence, reduced time to market, and the reinvestment of resources otherwise wasted on support and rework--the real benefits of automation.

The Big Lie

by Linda G. Hayes

Originally published in Datamation magazine, February 1998

One of the first statistics I ever learned about software quality was that the cost of fixing a defect rises exponentially the later it is discovered. So, a problem identified during requirements costs a fraction to fix compared with one found during development, which costs a fraction compared with one found in test. The most costly are those found in production.

What's interesting about this statistic is that while no one seems to challenge it, no one seems to behave as though it's true, either.

There is no reason convincing enough to ignore quality.

This is how it goes. A critical development project is late and has entered the classic schedule squeeze: The deadline is looming, but the software is just not ready. As the pressure rises, negotiation intensifies. How good is good enough? How many bugs are too many? How critical is this problem or that problem, really?

We've all been there. But proponents of meeting the deadline at all costs never openly argue that quality is not important. Instead, they trot out all the reasons why the release date is so critical, as though the drama of the deadline somehow outweighs the reality of readiness.

Here are a few examples:

We have a contractual deadline. This is a great excuse. Who can argue with a legal obligation? But this stance fails to take into account that the contract no doubt contemplated delivery of a working product.

For example, I know of a company that met its contractual commitment to deliver software modifications to a huge financial institution. In turn, that company invested countless dollars and personnel hours rolling the update out to thousands of tellers at hundreds of sites, only to have the system pulled out of production within the first week.

The financial repercussions of this fiasco far outstripped the penalty that would have been attached to late delivery. But the loss of confidence and goodwill was even more costly. The customer immediately began searching for a replacement system.

We have to meet the market window. No doubt about it, tax return preparation software that becomes available on April 16th has little value no matter how high the quality is. Yet few situations are that black and white.

The most public example of this position is the ill-fated release of the graphical version of dBase. At the time, Ashton-Tate dominated the PC database market, but it was late with support for Windows. Senior management committed a delivery date to industry and market analysts, and that date became the overriding reason to ship the product.

The product, you may recall, was so unstable that resellers returned it in droves. Within a matter of months, stockholders lost hundreds of millions of dollars in value, and the company lost its reputation and its position in the market . . . permanently.

We're going public. The IPO process is so expensive, time-consuming, and stressful that it takes on a life of its own. Top management tours the country on a "road show," extolling the company story and, inevitably, making promises about new products.

I recall a situation where the pressure was so intense that the version that finally shipped had not even been through a complete test cycle. In a particularly heated exchange with the testing manager, the CIO actually exclaimed, "Why should you test it anymore? You just keep finding bugs."

What was the result of this attitude? The IPO was successful, but within six months the stock's value dropped 80%, and the entire senior management team was replaced within a year.

And that's the big lie. There is no reason convincing enough to ignore quality. But if you do come up with one, chances are that in the long run it won't matter because you may end up without your customer, your job, or your company.

Process or Perish

by Linda G. Hayes

Originally published in Datamation magazine, November 1997

If pressed, most IT managers will admit that the word "process" conjures up images of oppressive bureaucracy, restrictive regimentation, and pointless delay. In a faster-is-better environment, process is usually the first casualty. Who has time to get it right when we're just trying to get it done?

It pleases me enormously to expose this belief as myth. I offer for your inspiration two real-life examples where process led to success.

Testing thrives in app dev

The first case verges on the incredible. Can you imagine a development manager in the breakneck-speed atmosphere of financial services who actually voluntarily initiates a formal "unit testing and review" process? Everyone knows that unit testing--in which a "unit" of software, that is, part of an application, is tested by the app developers themselves to determine whether it does what it's supposed to do--is one of the last bastions of spontaneous, seat-of-the-pants testing. Indeed, code reviews often border on invasion of privacy in most IT shops.

As part of a companywide initiative to improve quality, John Marcante, principal with the Vanguard Group, marshaled some 200 developers and put them through a training program to educate them on testing concepts and practices. He formed a special interest group that developed templates and standards, and coached the various project teams through implementation and reviews of requirements, design, and code. A year later, he requested an independent assessment of this effort and the corresponding results.

I had the pleasure of interviewing a wide sample of developers and managers, ranging from old pros to new hires. What I discovered was astonishing. Far from feeling constrained by this process, they thrived on it! Reviews uncovered missing or misunderstood requirements and provided guidance to new developers on how their pieces fit into the whole. There was sincere appreciation for the training and consistent use of the templates and standards. Moreover, team members shared an unwavering belief that the time that was added in the early stages of the project (about 10%) was well spent, because it saved about 30% down the line by catching errors and omissions sooner. The end result? A higher quality product in less time.

Business testing

Our next case study is at the opposite end of the organization and development spectrum, but still in financial services--insurance. Although Foremost Insurance had outsourced its IT function, the business side of the house understood that it still owned the systems and remained committed to their quality. Thus, Marg Blouw, assistant vice president of project management and testing, stepped up to the plate and instituted a formal QA process that started at the earliest phase of planning and tracked all the way through final acceptance testing.

The end result of the QA process was a higher quality product in less time.

Instead of making do with borrowed resources, Ms. Blouw formed a permanent, professional test team consisting of experts in the business who applied their insurance expertise to analyzing the application requirements and design, developing matrices of functions, and writing detailed test cases. Documentation, standards, and templates support a formal process that yields optimized test coverage. Review and sign-off from the end users at each phase ensures that everyone has input and buy-in. The payoff? Post-production incidents are down 80 to 90%.

What's especially intriguing about these cases is that neither used any advanced tools or leading-edge technologies. The weapons used were word processors, spreadsheets, and--above all--management commitment.

The High Cost of Low Investment

by Linda G. Hayes

Originally published in Datamation magazine, October 1997

I am continually bombarded with questions about ROI on software test automation. People want to know how to convince management that the investment will be recovered and will even generate a profit. The issue is as perplexing as it is prevalent. Everyone, it seems, wants to know...but no one does.

Well, there's a reason for that: Nobody really measures the cost of bad software. I am still looking for a company that actually accounts for the full cost of system failure.

Not just the support, development, testing, and installation time for a fix, but the cost of downtime on the system. Especially the downtime.

Top-line downtime

Downtime costs can dwarf an entire QA budget because the loss of use of an essential system can affect the top line, not just the bottom. If you want to make a compelling case to senior management for increasing investment in testing, look at what the system you are testing is used for, then find out what it means to the business if the system crashes. A stock brokerage firm estimated downtime costs at $250,000 per hour in lost fees and profits.

I am still looking for a company that actually accounts for the full cost of system failure.

The ultimate intangible cost, of course, is customer satisfaction. In today's mercurial and competitive economy, a loss of confidence can become a loss of revenue that is not recoverable.

Cost-line uptime

But even if your company doesn't admit to, let alone account for, lost opportunity from downtime, there is usually an amount budgeted for "maintenance." Maintenance costs are what is spent on existing systems, as opposed to development costs, which are what is spent on new ones. In most IT shops, maintenance consumes 60-80% of the overall budget.

So what is maintenance, anyway? We're not changing the oil, after all: Software is an intangible that is not subject to wear and tear like a piece of equipment. Maintenance is really a catch-all for fixing problems and incorporating enhancements. The real question is, What is the ratio of fixes to enhancements? Even if you can only get a ballpark estimate--be charitable, say 50%--it's still a staggering number. If you can reduce maintenance, you can free up resources for new development, which can improve your organization's competitive agility and eventually have a positive impact on the top line.

Another fertile area for potential savings is the cost of distributing a fix. For a financial services company with 10,000 field locations, the cost to download and install a new version of an application was estimated at $7 million. In another case, a network software company spent $40 million just to copy and mail update disks around the world.

Forget worksheets

Some of the test tool vendors have proposed worksheets demonstrating a cost reduction from automation based on time savings from test execution. I have a problem with this because it assumes that quality and test coverage are already acceptable and the goal is simply to reduce the expense. This doesn't describe any company I've ever known. The point of automation is not to shave a few more bodies out of what is usually an embarrassingly small test group anyway, it is to leverage the resources you do have to get more done.

The real ROI of test tools--or any investment in testing and quality--cannot be explained in terms of spending less money in testing, but in making more and spending less everywhere else.

How does your company measure the ROI of testing tools? Please e-mail me at linda@worksoft.com.

The Three Faces of Testing

by Linda G. Hayes

Originally published in Datamation magazine, September 1998

One reason testing is such a difficult topic for IT shops is that the same term is applied to an activity, a department, and a phase of the development cycle. To add further confusion, there are many kinds of test activities performed by different departments at different phases.

The upshot is that testing is misunderstood and, as a result, mismanaged. It's like trying to work with someone who has multiple personalities: Unless you know which personality you're dealing with, you don't know how to respond.

Exercising the code

Most people understand that testing is an activity performed to assure that a system works as expected, but few understand that there are many factors that go into making a system work. From the development point of view, testing is performed to exercise the actual code--all of its internal algorithms and pathways, and all of its external interfaces to other code modules.

Performance testing, often left until the final testing phase, should really be done in the design phase.

This type of testing is necessary throughout the development phase and must be performed by developers who are familiar with the inner workings of the application with tools that access lower system layers. For some reason, this testing is usually just blurred into development as an informal sideline instead of managed as a separate activity. Developers are seldom trained in test techniques or tools, and schedules rarely account for test time. The result is applications that are more complex and less stable than ever.

The black box

Once the development phase is thought to be complete, the system moves into both the testing department and the testing phase. Typically, this includes system-level activities, or "black box" tests, that assure that the system meets functionality expectations. At this phase, no knowledge is presumed about how the application is constructed, only about what it is required to do.

It's only logical that these types of testers need to be skilled in the business purposes of the system. There are tools that automate on-line entry and others that verify batch processes, but both types deal with the inputs and outputs of the system, not the internals.

When the going gets tough

But accurate inputs and outputs aren't enough anymore. Systems must not only provide a certain feature or function, they must do so in the target operations environment, under certain conditions, and within certain tolerances. For example, a system may be able to correctly retrieve and display stock quotes, but a system that takes five minutes to do it is useless to a trader who makes money on tiny ticks in prices.

Making sure the system functions when the going gets tough is usually called stress or performance testing. It requires yet another set of skills and tools and is most likely performed by a different department. Although often left until the final testing phase, if it's performed at all, this testing should really be done earliest--during the design phase--because performance problems that are caused by fundamental architectural decisions may require a major restructuring.

To conduct an effective exercise of extreme conditions, performance testers must be familiar with the network topology, the configuration and distribution of rules and data, and the middleware glue that binds it all together. People with this type of skill usually make their living in the operations support department, not testing.

So the next time someone raises a hue and cry about testing, don't bear down on the testing department or initiate process improvements in the testing phase. Instead, look at the entire IT organization and the complete life cycle.

Forget about "quality"

by Linda G. Hayes

Originally published in Datamation magazine, August 1997

After cheerfully trashing regression testing in last month's column, I must admit that it's easy to criticize but not very constructive. I should find fault only if I'm able to offer a solution to the problem of how companies can ensure that their business operations continue to function smoothly even after they've made major changes to their IT systems. So here's my solution.

First, we lose regression testing. It not only conjures up negative images, it's also all about proving a negative, which is impossible.

Instead, I propose a new class of testing--Business Process Assurance--and here's its definition: BPA assures that critical business processes still function after changes have been made

Notice what we are going to assure: not quality, whatever that is, but that critical business processes still function. For example, when we install that new GUI front end to our order-entry application, we need to ensure that the system continues to accept and log orders properly. The problem with quality assurance, or QA, is that "quality" is undefined. Does it mean fast, reliable, easy, powerful, or all of those things? And who decides what constitutes quality, and how is it measured? If you don't know, how can you assure it?

The problem with quality assurance, or QA, is that "quality" is undefined.

Consider the word critical, which means that BPA is concerned with those aspects of an application that are essential. In other words, we're not talking about every possible error condition, every bug ever detected, or each and every combination, boundary, data type, etc. This distinction is crucial because it implies risk management. If you know you don't have enough time or resources to get complete coverage, then prioritization is key.

Now think about a business process. The "business" half of this term places this type of testing squarely outside of development. The "process" half says that these are user scenarios, examples of everyday tasks that run the business, not mathematically derived test cases. Taken together, this means that the business-user community is both a participant and a beneficiary in BPA.

In the context of my definition, the term function has a whole new meaning. Previously, "functional" testing begged the question of which functions needed to be tested. With BPA, we know we are talking about critical business processes, not about whether the buttons and keys work.

Finally, let's examine the phrase after changes have been made. Notice the word "changes" is not qualified: Instead of just modifications to application software, a change can be to the hardware or any aspect of the environment--even the business process itself. Critical applications reside in highly complex, interconnected environments, relying on multiple tiers and layers of functionality. It is unrealistic and downright naive to apply application-centric test techniques to integrated IT environments.

The implication is that the test environment should be a microcosm of your production environment--not just in its configuration, but in its processing cycles. Thus, BPA testing can't be cadged from a corner of a development system, making do with volatile data and unstable software configurations. BPA requires a tightly controlled and well-managed test environment.

The last detail to settle here is exactly where Business Process Assurance fits in your organization. The most logical place is as a condition prerequisite to promotion into production. In other words, BPA should be the gateway to operations, the point at which the business confirms that it can continue to carry on uninterrupted. It happens last. Why is that so important? Someone in a deadline crunch might slight a process as obscure as regression testing, but who would dare to fail to assure that critical business processes still function? Perish the thought!

Don't Let Your Business Process Regress

by Linda G. Hayes

Originally published in Datamation magazine, July 1997

I must confess: I don't believe in regression testing. Not that I don't believe it can work, just that it doesn't.

After spending 15 years in software testing and working with hundreds of IT shops, I have to report that effective regression testing--which is supposed to ensure that, for example, your order-entry application keeps on working properly even after you've added a snazzy GUI to it--is about as common as development projects that are ahead of schedule and under budget.

There are exceptions; some IT shops no doubt do a great job. But it's also true that some people have survived jumping off the Golden Gate Bridge, which is not a strong recommendation for doing it.

So why doesn't regression testing work? I think it has to do with the way it is perceived and practiced.

Most IT shops are rewarded for what they create--new apps, more features--not for keeping the old stuff running.

For starters--and I'm only being halfway facetious here--what's with that name, anyway? Regression sounds bad--too much like repression, depression, and suppression. It sounds pejorative. And it doesn't seem to add a lot of value: Just makes sure that what used to work, still works. Most IT shops are rewarded for what they create--more features, new applications, cooler technologies. What glory is there in maintaining the status quo?

On the other hand, there is plenty of ignominy when things that used to work quit working. The Year 2000 is a great case in point. The entire purpose of testing for the millennium is to make sure that applications still work after the date change is accommodated. This is classic regression testing on a global scale, which is why we're in such deep trouble.

Furthermore, what's the difference between regression, system, or acceptance testing? Sure, there are subtle distinctions between the definitions of each, but at the end of the day, aren't they all about making sure the stuff works? Regression testing is about proving a negative: making sure that there is no unexpected impact from changes. How do you prove that something you don't expect to happen, didn't happen?

What's in a change?

Regression testing assumes you know what the application used to do before you make any changes, which implies a known set of requirements. But the fact is, few IT shops maintain current application requirements, relying instead on the individual expertise of testers conscripted from the ranks of business users. Thus, regression testing is only as complete as the knowledge of the person(s) performing the tests.

Aside from the inconsistency of this approach, there is a deeper problem. A "change" to an application's functionality is not necessarily the result of actual modifications to its code. In today's complex environment of interdependent and multitiered applications, there are myriad factors that can affect operations: a new version of the operating system, middleware, database, or change to the network topology or hardware configuration can result in downtime.

For example, I recently reviewed the production trouble tickets for a large IT shop and found that fewer than 25% of the incidents could truly be described as defects in the software itself. The rest were caused by ancillary system resources. Regression testers borrowed from the ranks of users can hardly be expected to know about, let alone understand and test for, the potential impact from changes of this nature.

But if regression testing sounds bad and is poorly practiced, what's the answer? Well, how about renaming, redefining, and repositioning it? But that's a topic for next month.

Testing: No Easy Way Out

by Linda G. Hayes

Originally published in Datamation magazine, June 1997

The good news for testers is that, with the staggering volume of software fixes required for Year 2000 projects, testing has suddenly been promoted from the back of the bus to the driver's seat. The bad news for everyone else--especially IS managers--is that there aren't nearly enough drivers.

Here's the situation: Hordes of vendors are riding to the rescue to help you fix your applications and data to become Y2K compliant, and although the vendors usually promise that their changes will work, even they estimate that they will find and fix only 70-80% of the problems.

If there are 12,000 code modules in an inventory and if 30% of them are at risk, that's 3,600 potential land mines still out there. Pretty scary.

To make matters worse, there's a paucity of resources to help you deal with the flip side of all these changes: Regression testing. Because testing is at least half the overall effort, why aren't there more resources?

For two reasons. First, after years of career genocide, people with testing expertise are scarce. The second reason has to do with what regression testing really means and how it does--or doesn't--get done.

Only the business that relies on an application can be relied on to test it.

Regression is the type of testing that you must perform to ensure that the functions that used to work haven't been broken--or regressed--by changes. It is meant to catch the unexpected effect of modifications, and is critical any time changes are made to applications already in production. With fixes of the scale and scope of Y2K, and the accompanying risk to your ongoing operations, regression testing is essential. Period.

The hidden assumption here is that these functions that need verification already reside in a library of regression test suites. But the embarrassing truth is, functional requirements usually exist only in the minds of expert application users, and so regression testing is a spontaneous, stream-of-consciousness event on the part of conscripted and largely migrant business workers.

Now I may invoke the wrath of the testing community here, but I don't think you can hire professional testers off the street who can sweep into a large, diverse computing environment and materialize a comprehensive regression testbed. Not because they don't know how to test, but because they don't know what to test.

These legacy systems are huge, encompassing decades of development and representing extensive expertise in the particular business requirements of the companies they serve. No amount of education in the science of testing will reveal the 49 ways to purchase, transfer, or sell shares in a retirement plan, for example, or explain the impact of each on tax reporting. Yet it is precisely this type of knowledge that is critical to effective regression testing.

If I haven't committed outright testing heresy yet, this will do it: If you follow traditional test-case design techniques, you will quickly find that you have hundreds of thousands--probably millions--of test cases to develop. Given the size and number of applications involved, even the most elementary approaches will yield an absurd volume of test conditions.

What to do?

All hands on deck: This is survival mode. Admit that you can't palm off this problem to outsiders; only the business that relies on an application can be relied on to test it. For each application, form a task force comprising representatives from the user community, development, production support, and testing. Identify the inputs and outputs, including all data files, that are involved, and organize the flow of this information into operating cycles--daily, weekly, monthly, annually, etc.--that mimic what should happen when the switch is thrown on the software after the Year 2000. Now, leave the rest to your testers--if you can find any.

Boost Your Test Team's Value

by Linda G. Hayes

Originally published in Datamation magazine, May 1997

Last month I promised to tell you how to implement test tools successfully.

You may recall my point that these tools are basically specialized languages with all the attributes of programming, yet the most valuable test resources are typically recruited from the business community. Thus, a disconnect. But you can overcome this disconnect by smart project management.

Think about it this way: Testware is software. Automating the testing process is no different than automating accounting or any other application. If you were doing your books manually and needed to automate, would you buy your bookkeeper a C compiler? This is essentially what happens when test tools are handed to testers whose primary skill is application expertise. Even if they are willing and able to learn the scripting language, the odds are they won't have the experience to develop tests that are structured for easy maintenance and manageability.

What to do? Divide and conquer. Divide the project in two places: at the skill and task levels.

Different disciplines

At the skill level, this means recognizing that testing and automation are really two different disciplines, and you need both of them to succeed. The testing aspect covers what to test--the inputs and expected outputs that demonstrate functionality. Automation, on the other hand, deals with how the tests will be executed and managed--the script commands and logic that will apply the inputs, evaluate the outputs, and report the results, as well as the library control procedures that keeps it all straight. To return to our accounting analogy, the bookkeeper is responsible for coding and entering the accounts and transactions, the accounting system handles the posting and reporting, and the controller manages the finances.

The implication of this approach is that only a small core team of people--a couple of developers really--need to have test tool skills, and they are responsible for building the script engine. This engine is designed to execute test cases, which are supplied as sets of data; almost all test tools accept data from external files. Another team member serves as the test librarian--someone who is comfortable with change management, version control, and configuration management issues. The rest of the team--the business experts--supplies the test cases, which can be created in their utility of choice: a familiar word processor, database, or spreadsheet.

There are numerous advantages to this technique. By using technical people to work with the tool, you can honor the power and complexity of the language instead of trying to hide it through capture/playback. Maintenance is reduced because test cases are not hard-coded into scripts. A larger population can contribute to the automated test library because participation is not constrained by specialized skills, and test cases are easier to read since they are presented as data instead of as scripts.

Just like app dev

At the task level, divide the development of the script engine just as you would the development of any application. Identify utility functions, such as log-on, data setup, log-off, and cleanup, and create them as routines that can be shared across the test library. Use modularity to segregate and resolve common issues like error recovery and synchronization checking. Focus on a script library architecture that takes advantage of reuse and thus minimizes the impact of later changes to the application under test.

Also, divide the development of test cases. Identify and prioritize requirements according to risk, and assign related areas to testers with pertinent expertise. Organize the test cases into suites and cycles that mirror the application under test: daily, weekly, monthly, quarterly, and annual processing sequences. Package a subset of critical tests into a build verification cycle, which can serve as an entry criteria into the test phase, and give developers access to it.

In other words, you need to treat your test automation effort with the same respect you would any other development and implementation project. Organize and manage it to take maximum advantage of the specialized skills of each member of the team and the specific purpose of each element of the test library.

The Truth About Automated Test Tools

by Linda G. Hayes

Originally published in Datamation magazine, April 1997

Yes, there is justice. After being one of the original purveyors of automated test tools, I now find myself in the karmic position of helping companies actually implement them.

Unfortunately, I have to spend the first session deflating the hopes and dreams left behind by the salesperson, who has skillfully presented the tool as an instant panacea for increasing quality while reducing cycle time, resources, and the federal deficit.

Although my experience has been with automated tools for testing software, we could be talking about any kind of IT tool--whether for enterprise resource planning, Year 2000 projects, or on-line analytical processing--that you've paid big bucks for in expectation of a big payoff.

Management just dropped big bucks on this tool, and people are waiting impatiently for miracles to occur.

Here's the automated test tool promise: It is able to merely observe the typical manual test process as it is being performed, then magically reproduce the same effort, effortlessly. What could be easier?

The Siegfried and Roy magic act for one, because it's what you don't see that makes it all work. So let's look behind the curtain.

Foreign languages

Virtually all automated test tools for graphical applications are really just specialized languages. Some resemble C, others Basic or Assembler, but all offer the basic syntax and constructs of a programming language. Since the best testers are typically business users instead of programmers, this is a fundamental disconnect.

As a vendor, you obscure this awkward truth through the seductive simplicity of capture/playback. You activate a "record" mode that captures the tester's manual activities and converts them into the script language behind the scenes. Pretty slick, actually, except for a few million details.

One of these details is that this type of recording is "blind" in the sense that it captures the activities without any sense of context. For example, if you enter data into a series of fields, you'll get back the sequence of input values and any navigation keys you pressed, but you won't be able to tell by looking at the script which value went into which field. Not only is the script impossible to read, but any inconsistency or change in the order of the fields--which is inevitable--causes the script to shift out of context or fail altogether.

So what do you do? You either throw the script out and start over, and eventually throw the tool out, or you take a deep breath and dive into the script language to create a test you can live with.

The trap

Now you're caught. Developing test scripts that are readable, maintainable, and reliable is just as challenging as developing the system you're testing, because your test scripts have to stay in perfect harmony with the application they're testing. This is no mean feat. A single change to a single window can ripple throughout dozens--dare I say hundreds?--of test scripts. And your test scripts have all the attributes of source code: They require structure, documentation, version control, change control, configuration management--well, you get the picture.

The most painful part is that management just dropped big bucks on this tool and people are waiting impatiently for miracles to occur. Meanwhile, you feel as though you've just been catapulted into a bad video game.

Is it hopeless? No. There is a way to realize the benefits of these tools, which, in spite of all this, are profound. It's just that there's no free lunch, no silver bullet, no Santa Claus. If we could just own up at the outset and set realistic expectations, get the right resources, and invest the necessary time, we would enjoy a tremendous payback. I should know--I'm doing it--and next month I'll tell you how.

The Tyranny of the Schedule

by Linda G. Hayes

Originally published in Datamation magazine, March 1997

As far as I can tell, the No. 1 scourge of IS shops is The Schedule. The Schedule, it seems, is an intractable and irresistible force that exists outside of reality. It routinely demands unnatural acts, bursts of heroism, and sacrifices on all sides.

Why is that? With so many smart people with such powerful project management tools to subdue The Schedule, how does it get away from us?

Either The Schedule is never in control to begin with because it's driven by an external event--a critical customer need or a marketing window--that establishes an arbitrary deadline not based on the actual effort required and resources available, or The Schedule starts out in control and events conspire against it. Where it's first conceived with reason and planning, severe problems are discovered in the production system, or resources are lost to higher priorities, or any of a host of typical, but unpredictable, hurdles must be overcome.

The marketing department screams about forgone profits for every day The Schedule slips.

But no matter how it happens, all schedules seem to have one thing in common: They seem to get off track at the end. Have you ever noticed that a project can be swimming along until it hits system test and then, suddenly, it's not just late, but no one can even say when it will be ready. Meanwhile, the marketing department is screaming about lost opportunities and forgone profits for every day The Schedule slips.

I have a theory about this. Consider the software development process, which comprises a series of phases, like a relay race: The baton must be passed--the requirements understood, the design developed, the code written and integrated, and the final product tested--before the next runner can take off.

Now, think about trying to save time in a race, so the next runner starts before actually getting the baton, betting that the baton will catch up later. Each successive runner gets farther and farther ahead of the baton, so you can't tell whether you're winning the race by watching the runners.

This is how it happens: The requirements are a little hazy, but time is wasting, so you forge ahead. The design looks okay except the requirements are still not finalized, but development proceeds anyway. Then the requirements expand, the design is massaged on the fly, and your IS staff begins to work inhuman hours. Unit and integration testing get short shrift and system testing commences with a struggle to get the system even to install.

In other words, none of the phases has a clear beginning or end--except for testing, whose end means that the system is "ready."

According to The Schedule, of course, each phase started on time --it's just that the one before it wasn't necessarily finished. So, at the end of the race--the system test, where the judges are waiting to flag the winner--you have to wait for the baton to catch up. If the baton is still back in requirements or design, you're in real trouble because you truly don't know how far you have to go. To make matters worse, no one is expecting a delay, because The Schedule says everything's okay.

It's true that the process isn't perfect: You can't stop everything while you clean up some prior details. But unless you have some objective basis for declaring a promotion to the next phase, you don't know whether you're on The Schedule or not. Until the end.

So why is it important to know earlier that you're off The Schedule? In a word: mitigation. If you know you're behind, the sooner you admit it the faster you can do something about it: add resources, cut requirements, plead for extensions. The classic pass-the-buck approach that sticks the entire weight of The Schedule on the final system test phase is not only unfair, it's bad management.

The Year 2000 and the S&L Crisis

by Linda G. Hayes

Originally published in Datamation magazine, March 1997

What do the Year 2000 and the S&L crisis have in common? Both are credited with horrendous costs that escalate with each report, and neither is at all what it seems.

The so-called S&L crisis was not the result of an assault on the S&L industry by outside forces, an all-encompassing conspiracy, or a cataclysmic event. What it was, really, was a profound and massive regulatory failure.

Congress deregulated the industry and opened the door to speculative investments, and what looked like a good thing during boom times turned into a catastrophe when the economy hit the skids. Ultimately, the blame did not belong to the greedy opportunists who took advantage of the situation; the real culpability was with the legislature that set up the possibility. That's what I think, anyway.

And the same thing is true with the so-called Year 2000 crisis. It's not that a malevolent millennium has attacked our corporate IS systems, leaving destruction in its wake. What's really going on here is that the bill is finally coming due for decades of sloppy source control, negligent configuration management, and generally irresponsible development and testing practices that have valued delivery schedules above discipline and have mortgaged the future to make today's numbers.

At many companies, the Year 2000 budget is like a bill going through Congress: Riders are being attached left and right.

What's brought us to this sorry state of affairs? Take a look around. End users have been coddled as incapable of articulating their needs formally enough to support requirements. Programmers have been treated like temperamental artists whose creative juices will dry up if anyone dares to impose order and organization on their work. Testers have been tolerated as obstacles, nags, and whiners whose only purpose is to screw up the schedule. The battle cry of "time to market" or "competitive agility" or any other of a host of excuses has been trotted out to sidestep structure, bypass processes, and generally slap systems together. Maybe I'm exaggerating. But maybe not.

The biggest tragedy about the Year 2000 saga is that management is now setting up SWAT teams, armed to the teeth with tools, to hunt down and eliminate the insidious date problem. It's like the S&L trials--if we can just punish the guilty, everything will be okay. Nonsense. This is not about date fields, this is about software management and control. This is about knowing what your systems do, how they do it, and where. This is about being able to make changes to your software, test it effectively, and deliver it with confidence. What the SWAT teams are finding is that some of the source code is missing in action, the rest is a crazy patchwork of inter- and intrasystem dependencies, and effective regression testing is a conceptual term bandied about by academics. How did this happen?

Banks without vaults

It's not like we can claim surprise, either. The rising tide of maintenance has been slowly engulfing development budgets, consuming the vast majority of funds. And what is maintenance, exactly, in the context of software? Software is not like your lawn, falling into disrepair out of neglect; nor is it an engine, requiring oil to keep the wheels turning. Software maintenance is a giant rug that we sweep rework under--the work we didn't finish when we supposedly met the schedule.

Until now it's been possible to stay ahead of the wave. Patching systems, replacing them, or escaping to client/server have all served to dodge and obscure the real issue: Most IS organizations are run like a bank without a vault. Would you do business with a bank where the tellers kept the money in their pockets, purses, and money clips? Even if we assume the tellers are honest and trustworthy, people make mistakes. They leave. They forget. I know if my teller starts making change from his wallet, I'm out of there.

Here's a news flash: The amount of money spent on major IS systems is no less an asset than the money in a bank--and a pretty big bank, at that. Yet that asset--in this case, the source code and all supporting documentation--is like so much loose change in the programmers' pockets. There is no accounting, no audit, no ownership. And no wonder that we can't put our finger on each and every date routine and dependency.

I'm not faulting the programmers, the users, or the testers. They are all creatures of their culture. The guilty parties occupy the highest levels of management, who will budget hundreds of millions of dollars to build systems but won't part with pennies to invest in the very infrastructure that supports those systems. I've seen huge, wildly profitable companies that can't seem to afford the tools, training, and time to help their staff do a professional job--even though the systems they're building are mission-critical to the top line.

No one intentionally sets out to short quality, and at some level almost all managers understand that quality is key to long-term success. Where it breaks down is in the short term. This deadline, that contract, the next sale, a critical market window, just this one fix or feature--all of these conspire to put pressure on the best of intentions. Quality is the battle lost by daily compromise, not by sudden surrender. Schedules loom large, and the price for missing the deadline is more immediate, and somehow more real, than the price for skipping the discipline.

What should be happening is a merciless evaluation of the practices that created the Year 2000 predicament in the first place so that it won't happen again. What's happening instead is that the mass migration to client/server is opening up new frontiers, a virtual Wild West of workstations, servers, and tangled interconnections that defy control and standardization. If you thought it was hard to keep a handle on a mainframe that was corralled in a glass-fronted room, just wait for servers that can hide under desks. It's scary.

Like a Congressional bill

But there are bright spots. At many companies, the Year 2000 budget is like a bill going through Congress: Riders are being attached left and right. These riders are allotments to invest in measures that have been put off for years and will offer residual value long after the Year 2000 has come and gone. Boring infrastructure investments that would never have seen the light of day before are now being heralded as heroic measures necessary to save the corporation from certain embarrassment.

Just gathering an inventory of all the applications that have sprouted throughout the company has value: It can be downright amazing to discover the true state of application deployment throughout the enterprise. Some companies are discovering redundancy on a monumental scale; in one case, there were a dozen different systems all trying--without any measurable success--to track software issues. Think about it. This type of information alone can spark a reevaluation of the IS architecture and its purpose within the company.

Also, the process of identifying date fields and their dependencies can be used to create complete data dictionaries and dataflow diagrams, which shed light on the flow of information through the corporation and enable future modifications. Information flows reveal other issues: duplication, omission, and complexity are often uncovered, leading to opportunities for streamlining both systems and the processes they support.

And, of course, preparing to test applications after the date changes are implemented is a wonderful opportunity to get a grip on requirements and to shore up regression test libraries. This is an especially sore spot in most companies, since they are always too busy trying to make changes to figure out what is already there. Tools are another beneficial side effect of the Year 2000, and so is documentation. At least the tools will remain, after the consultants have gone, and the documentation this time around is bound to be better than what was left previously, the poor quality of which exacerbated many companies' Year 2000 efforts.

If these efforts are successful, then the Year 2000 can be credited with, in fact, getting us ready for a new millennium.