Application Integration

Lessons on developing and testing software suites

by Sam Guckenheimer

You are responsible for a strategic new product suite. You will have a "team of teams" consisting of eight development groups in five locations, each expert in its own domain. Each group is used to setting its own priorities independently and will be working on its own enhancements concurrently to your project. Product quality is essential, but time is also of the essence.

QUICK LOOK

- Risks and recommendations
- How to handle complex architectural dependencies
- Integrated development teams

That was the task assignment at Rational Software Corporation in early 1998. Customers in maturing markets want complete solutions—and that usually means integration of several originally separate products into suites. That was our team's challenge: we had a set of component products, originally from separate companies, that addressed specific functions in the software development lifecycle. Our goal was an integrated suite of tools that spanned the lifecycle of software development and deployment. This article is about what we learned along the way.

Any big development effort has risks, and it's a manager's job to prioritize those risks and deal with them. In this situation, your past experience as a manager can lead you astray. You may think that a particular risk has low priority, only to learn otherwise late in the project. Here we'll examine a series of risks you should beware of, as well some brief recommendations about handling these risks. Space doesn't allow more detail—but knowing the risk exists is more than half the battle. There are several key characteristics that make managing development of an integrated suite challenging:

Seamless Integration Each of the formerly separate products is still visible to the user. The user interfaces must be reconciled—a user performing a task that requires two of the components shouldn't have to deal with gratuitous differences. More importantly, the different components might use variants of the same underlying ideas. For example, we determined that a single context menu invoked from right mouse click in the Rational TestManager component would include options to view requirement details (in Rational RequisitePro), GUI script playback (in Rational Robot), and a load testing schedule (in Rational PerformanceStudio). All four of these formerly separate products needed to appear and work as part of a smooth user experience.

Geography You are managing a "team of teams" that's geographically distributed.

Legacy Each team has existing code that must be both updated and integrated into a whole. Typically, the originally separate products have infrastructure code that serves similar functions. The merged suite needs only one copy, but which one? And how should it be changed to serve all the components' needs? Furthermore, each team has a base of existing users who require a comfortable transition from the familiar versions of their component products to the new expanded suite.

RISK

Inconsistent View of Requirements and Process

It's tempting to think that a large group of talented people will work together cohesively without a lot of rules. In our project, we were reminded that this is not so simple. We were all used to working in teams small enough to fit in one location with minimal external interaction or dependency. We discovered that each team had evolved different processes, tools, styles—even vocabulary. Terms like *spec*, *test plan*, *beta*, and *test case* all had different meanings in different teams. Different products had presented the same terms within their user interfaces, yet with different semantic meaning. We started with different forms of project management, different source repositories, different tools and methods, and different formats and standards for the artifacts we used.

The natural tendency of such groups is to try to retain autonomy, a way of reducing risk that historically worked. But this would not allow the seamless integration we required.

☛ RECOMMENDATION ☚

Make Use Cases a Key Specification

We discovered that one of the most powerful ways to communicate a shared meaning between teams (and between developers and testers) was through use cases. These are statements of requirements as ordinary expected user activities, expressed as a sequence of user actions and the observable system responses. [For a Use Case example, see page 32 of Ross Collard's article on use cases and testing in this issue.]

A developer tends to think of features independently, but the user's perception of a feature is probably determined by the use cases that encompass it. As a project manager, you can cluster features according to use cases and prioritize them in groups. When a particular feature is late, you can determine the impact based on your knowledge of the use cases, and may reprioritize other features to make sure that the use cases that are delivered are complete.

Defining use cases *early* forced a common vocabulary and a common understanding of requirements. Consider for example the concept of defect workflow. In an integrated defect management system, it should be possible to launch test scripts, view requirements, or view source code when looking at a particular defect report. Further, changes in any one of the linked artifacts should be visible from any other. When product components are separate, different terminology and object models may be tolerable. But when they are being integrated, and a use case shows the integrated experience, discrepancies among concepts like "Open," "Pending," "Resolved," and "Submitted" become obvious and painful.

Use cases were a key input to prioritizing requirements. By tying development milestones to the successful implementation of groups of use cases, **the testing effort could be focused with every milestone** on those parts of the system that were intended to function. (And correspondingly, testers weren't frustrated by discovering late that they were blocked trying to test subsystems that weren't intended to work yet.)

RISK

Requirements Shift and Feature Creep

Requirements have a way of changing considerably over the course of a large project. For example, we found that product licensing was a considerably more complicated topic than anyone realized initially. At the same time, end-user demands, particularly in Web and e-commerce environments, continually drove pressure for new product features. Because of the need to maintain a seamless interface, feature creep affects all of the geographically distributed teams. You must keep everyone in sync.

☛ RECOMMENDATION ☚

Automate Requirements Management Across the Team

When requirements change and need to be shared across teams, it becomes very important that every team member have a consistent, current view of the project requirements. It is also essential for effective project management that you be able to **report test implementation and execution progress against the *current* set of project requirements.**

☛ RECOMMENDATION ☚

Communicate Frequently and Personally

In addition to regular weekly project meetings by conference call, we instituted a couple of key communication paths to help groups work together more smoothly than they could have otherwise. We instituted periodic face-to-face "summit meetings," which included key reviews of requirements, design, implementation, testing, and cross-group issues. We held these every one or two months, depending on project phase. Secondly, we instituted an on-call system in which each group designated a cell phone number that would be staffed during extended business hours for support of other groups during implementation.

Complex Intergroup Architectural Dependencies

We needed early agreement on architecture. This can be particularly painful when many of these choices involve change to the legacies of the teams involved. Geographical dispersion makes this especially difficult and increases the need for formal agreement.

Particularly tricky are the consumer-supplier relationships among the teams. We made several decisions to share engines for common functionality. For example, the test product team decided to base all activities on a common requirements engine—the COM server from our RequisitePro product. So the requirements product team (building the server) had to understand the needs of the test product team (building on *top* of that server), despite the two teams being located two thousand miles apart.

We found that **individual teams tend to significantly underestimate the risk in cross-team integration.** This can be exacerbated by the team's expert focus on the features in its own product area(s).

Most teams are used to planning their implementations with high probabilities of completion, say 90%, and will appropriately resist the cost of trying to get to even higher certainties. The difficulty in a large distributed project comes from a multiplicative effect. The probability of N independent teams, all with 90% probability of finishing on schedule, is .9 raised to the Nth power. The effect of putting together a suite increases N to the point where there is a diminishing chance of successful completion (see Figure 1). For example, if there are seven groups each with 90% confidence of on-time completion, the probability of the entire project's completion on target is less than 50%!

In a project with relatively independent components, one solution is a "train release," in which late teams just miss the release. It did not fit our need for seamless integration, where our objective was to support new use cases that spanned the previously separate products. If one

The Culture of Collaboration

The unified office suites that so many users take for granted today—the seamless series of information handoffs between word processing, databases, spreadsheets, multimedia, email, and Web applications—haven't always been the rule of thumb. Like any other "blended family," their creation was a complex dance of compromise, culture clash, and cooperation.

Jeanne Sheldon, Director of Microsoft Office Sustaining Engineering, has been chaperoning that dance for several years at the software giant, since the days before the first Microsoft "Office" suite. There are a number of priorities that are important for any integration effort, she says: change request management, linking the milestones across your applications, and using a common "check-in" build and verification procedure. But one of the vital components of suite building has less to do with code than it does with culture.

Especially in the early days, remembers Sheldon, the formerly autonomous project groups struggled from time to time with the culture shock of the new process. "Even with standard products such as Word and Excel," she says, "when it came time for design decisions, different teams would want to cling to their traditional ways of doing things." During disagreements about which infrastructure code or interface component to adopt as the standard, Sheldon says, one team would inevitably tell the other that "you just don't understand our users."

The key to making the process work, says Sheldon, was to understand the culture shifts involved in integrating the work of several teams. "Good-natured rivalry, group pride…some of the very dynamics that helped us when we were working on separate products sort of got in the way now that we were all working on the same project."

So how do you bring the new family together? There are two approaches, says Sheldon: centralization and cooperation. Each emphasis carries its own risks. "We strove for cooperation, rather than a sort of rigid centralization," she says. That got the cultural shift to work, by maintaining many of the individual application workgroups but redirecting their sense of purpose. The risk of the cooperative approach, and of keeping the structure decentralized, she points out, is that the Word person has to *want* to help the PowerPoint team…not because that's the team you work for, but because you see the bigger picture, and the common benefit to ALL your project teams. Fortunately, she says, at Microsoft that process worked, as team members concentrated on the greater good, the bigger process.

"We brought everyone along piece by piece," recalls Sheldon, "and some of it was subtle adjustments in the culture. We changed the way we talked about our product…now we talked about shipping 'Office,' and improving the 'suite.'" At the time, she says, you don't notice the big shifts in your teams' viewpoint—all you see is the step-by-step integration, the increasing creativity of collaborative decisions, the difficulties and the triumphs. "But you notice it when you bring in a new application with a new team," Sheldon says. "When that new team member looks skeptical about some cross-application function, that's when you can smile across the table at another old-timer, and say 'I remember being there…'"

—A.W.

team missed the train, a great many use cases wouldn't work.

✍ RECOMMENDATION ✍ Iterative Development with Formal Internal Releases

We found that it is essential to have **an iterative development plan with internal milestones.** We also found that the intermediate milestones must have "teeth"—clear and measurable acceptance criteria that drive each component project and the integration to closure. Project leads could use these to accurately chart progress on the calendar and respond in a timely way to slips and missed dates. It is key to get the development teams to deliver the integration technology at milestones in early iteration milestones, so that implementation and testing can uncover risk and drive rework and resource reallocation early in the lifecycle.

Internal milestones need to be paired: one milestone for delivery, a second for acceptance. A classic difficulty was cross-site miscommunication and debugging. Suppliers thought that their components were performing as spec'd, but consumers were still not satisfied. A lot of this misunderstanding was avoided **when the consumer supplied acceptance tests to the supplier.** In other industries this may be a common practice, but for many of our software teams it was a novelty to have an "outside" group provide them.

Iterative development provides an important flexibility by allowing you to see explicit phases of work and assess project status at clear milestones. Using iterations allowed us to identify serious misunderstandings early in the project, test early and continuously, assess the project's realistic status, tune the process as we went, and spread the work load of the teams (especially in testing) more evenly across the project.

The practice of iterative development has particular implications for testing. Actual feature completion may vary within a considerable range, and you have to accept that the precise deliverable feature set may not be known until quite late in the iteration. (Indeed, it may depend on testing results!) You have to match testing plan priorities and schedules closely to the prioritization of features and be ready to reprioritize frequently. This is sometimes hard for testing teams to accept initially, but the payback is significant. You see quickly that the effort spent in test implementation is used effectively and has a clear impact on the project deliverables.

✍ RECOMMENDATION ✍ Unified Change Request Management

I can't stress enough the **importance of a common defect handling process and database** with very disciplined submission, reporting, and status tracking. These let everyone see the same picture of the state of the project and eliminated many misunderstandings that came from having separate bug databases and ranking schemes. It took some up-front work among the project managers to agree on a sufficiently rich schema to accommodate the needs of all the component teams. Of course, the payoff is that all member teams can use the same project metrics for assessing status. Project meetings can then focus on what actions to take, not on whose data to believe.

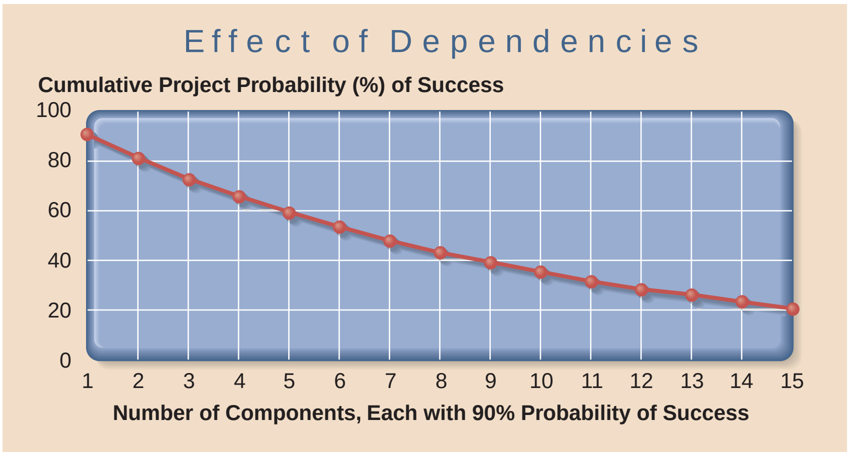


FIGURE 1 **The difficulty in completing a large distributed project comes from the multiplicative effect of component and team dependencies.**

✍ RECOMMENDATION ✍ Project Structure

The structure of intergroup dependency required designating a team with very clear ownership of the integration and certification of the complete suite (see Figure 2). This "Suite Integration Team" had responsibility for certification of the complete suite and the interoperation of all components. Each "consumer" team would deliver tests upstream to supplier teams to certify incoming quality before delivery. The Suite Integration Team would follow a similar process, but for all components in the broader context.

We also used a cross-team "Change Control Board" (CCB), including program managers from the component teams. For the few weeks leading to every cross-team milestone, the CCB would convene and serve as the communication funnel for any changes intended by any group. This gave every "consumer" the opportunity to influence supplier priorities and to review possible effects of change in advance. The CCB used the same priority ranking scheme as the component teams to triage incoming defects and requests.

RISK

Testers Don't Have Knowledge of the Suite

It is always difficult for testers to gain a deep enough understanding of a product to test complex functionality. When the application under test is a suite composed of previously separate products, and the goal is to enable

CHART BY ANNIE BISSETT

broader usage models than have been possible before, this problem is compounded. Most of the testers probably come from individual product groups, so they know their respective pieces well but don't have a clear perspective of the broader suite.

The natural tendency of each team is to prioritize individual-product feature testing over integration testing. It is usually easier to test the individual features, because they are better understood, require less setup, and can reuse existing test assets more. This pattern makes it likely that key functionality will quickly fall between the cracks.

It is usually difficult for teams to review each other's test plans effectively, because the reviewers often lack the necessary context to see the implications of details and omissions.

☞ RECOMMENDATION ☜
Use Cases

Use cases (again) can provide master recipes for how the products are to fit together and how the user interacts with them. We found that these were by far the most valuable form of requirements for extending an individual tester's knowledge of what product areas to explore and what tests to develop.

R I S K
Shaky Build and Release Process

A structural effect of this kind of distributed project is that the many teams who are working in parallel need to merge their code frequently, prepare for specific milestones, and at the same time be able to branch to begin new work. It would be typical for a component team to deliver iteration *A* for integration and start iteration *B*. Meanwhile, the integration team would be working with the bits delivered from iteration *A* and require bug fixes of the component team. This required each team to have the ability to manage separate workspaces for the different iterations and to be able to merge fixes back to the mainline.

Along the way, there are risks that bug fixes made in iteration *A* do not get merged safely into iteration *B*. At the same time, each team assumes interface functionality long before it is available. Regressions may not be apparent to the supplier team, because they may only affect consumer interfaces or cross-component use cases. All of these pose the further risk that different teams look at inconsistent data and waste time trying to reconcile facts, rather than decide an appropriate course of action. We found that four practices helped to mitigate these problems.

☞ RECOMMENDATION ☜
Unify Configuration Management Across the Teams

Multi-site, geographically distributed development requires the ability to share a common source repository. We found that it was essential for each team to have a local build and test process that could simultaneously draw on other teams' source modules. Each team needed to verify its bits against the public components and then promote and publish its source to the public repository. This was essential to avoid version skew across the components. ("Are we sure that Team B isn't using a different version of **foo.dll** than Team A?") As we approached milestones, complete system builds were a daily process. This required significant configuration management automation.

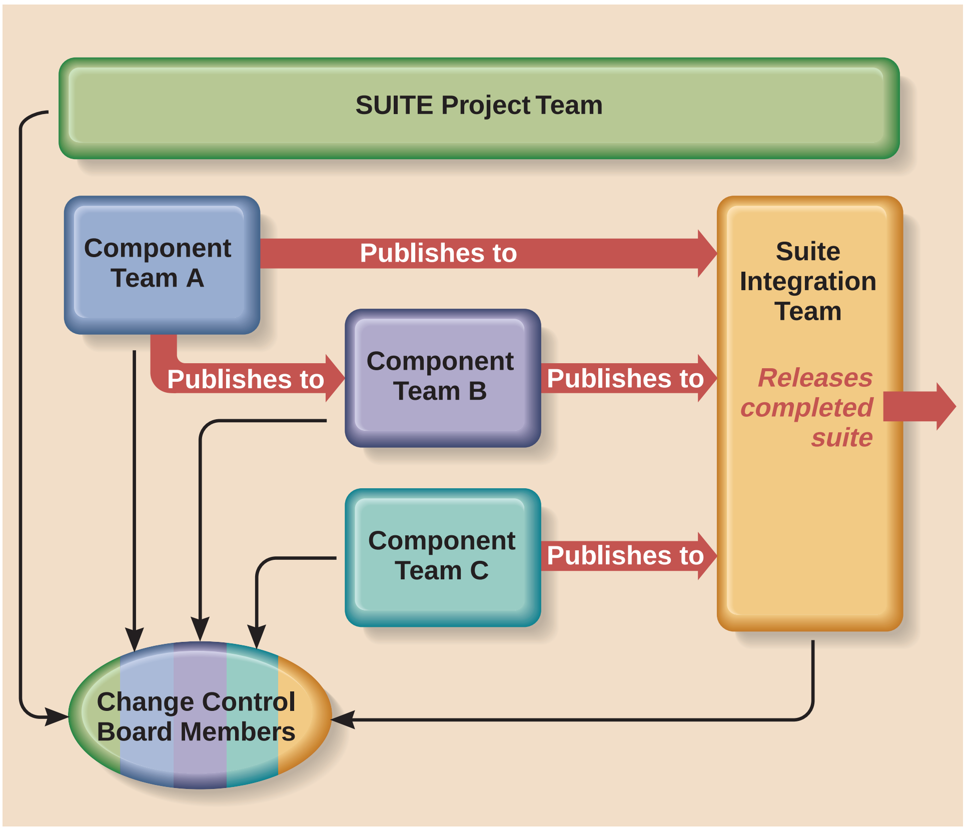


FIGURE 2
The Suite Integration Team has responsibility for certification of the complete suite.

☚ RECOMMENDATION ☛
Nightly Builds

We found that, as part of our build and release process, we needed the nightly heartbeat of test results. Each morning's email held two key broadcasts that provided an important psychological impact: the countdown to the next milestone and the summary status report from the regression tests of the previous night. While regressions were uncommon, they did happen, and catching them immediately was an enormous benefit to making forward progress. We also needed to see continually where testing was blocked due to incomplete integrations or missing but expected functionality. When the morning's report showed unexpected failures, it had the effect of warning beeps on the EKG—everyone scrambled to understand them and fix the problem. [For more information on nightly builds, see "From the Front Line," in Volume 1/Issue 3 of STQE.]

☚ RECOMMENDATION ☛
Use Simple, Consistent Reporting

One of the most useful tools for communicating project state was a simple graph of defect find and fix rates. This proved to be an intuitive forecasting tool and an easy-to-grasp representation for management. The idea is to show weekly rates of newly found defects and newly fixed defects on one graph. You can look at this for the project as a whole and drill down into suspect problem areas—by component, by team, by developer, by defect source, by configuration, by problem priority, etc.

This graph gives an objective presentation of project status that everyone on the team can understand and that works at all levels of the data (see Figure 3, previous page). I caution that projects vary in complexity and duration, so there are no absolute norms here. And whenever the bits are exposed to a new test group (e.g., integration teams, beta users), there should be a burst in the find rate. **The most important thing is to compare actual findings to expectations** and to use this as a basis for discussion and analysis.

RISK
Complex Configuration Dependencies

When you are building a suite from existing products, the issue of configurations becomes quite tricky. **Not all the products coming into the mix will have the same supported configuration matrices** (platforms, service-pack levels, required and related third-party tools, supported database configurations, memory, CPU and disk requirements, minimum display requirements, internationalization and localization requirements, etc.). Moreover, not all of the teams will have the same dependence on these specifics or appreciation of these details. For example, most products will be shielded from dependence on specific service pack levels, while others may depend critically on differences among them.

☚ RECOMMENDATION ☛
Nail This Down Early

Specifying the configurations and the responsibilities for verifying configuration support takes on a new importance. Because the target configuration choices affect all of the component teams, they need to know the list clearly—and early in the project.

☚ RECOMMENDATION ☛
Spend Enough Time and Money on Configuration Testing

Installation environments can also make a huge but subtle difference. For example, we found that we were limited in our ability to update common runtime **dll**s, because the update could affect third-party products that had to be present on the same machines. There was no shortcut to discovering these issues—they required careful configuration testing.

RISK
Reuse

A component product that evolves from a standalone to a suite has to confront new issues. For example, if two products use the same graphing control, they obviously need to ship with the same version, and—less obviously—be tested against other versions that might be installed already. A product that evolved with its own user interface, now interacting as a COM server, may encounter all sorts of data that were previously impossible. Many of the boundary conditions and error cases have probably never been considered before. Prior experience with the standalone product will not prepare you to think about these new cases thoroughly.

☚ RECOMMENDATION ☛
Reuse Tests Through Test Automation

In this case, test automation plays a particular role. It is the consumer team that is usually best able to design and implement effective tests. Automating the tests and delivering them upstream to the supplier team makes it much easier for the supplier team to reproduce problems. While this requires investment in automation of test set-up and execution, it saves enormously in the "cannot reproduce" phenomenon that may haunt cross-product compatibility bugs.

RISK

Unreproducible bugs

The last thing you want in a distributed testing and implementation effort is for *"Cannot reproduce"* to become the most common defect resolution. This is an enormous risk. Defect reports have to be reproducible to be actionable, especially when they involve complex dependencies.

☛ RECOMMENDATION ☚
Test Automation

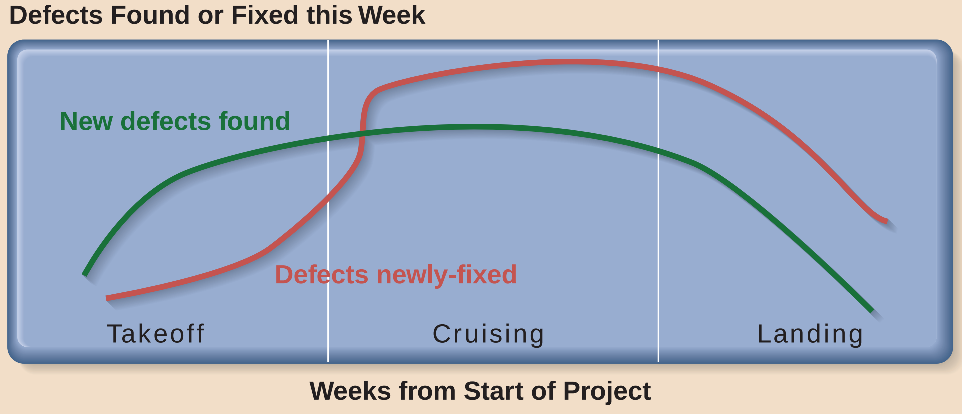
We found that **it was essential for teams to share test plans, test tools, and test assets.** This can be a difficult cultural transition when some teams are used to less formal processes.

The increased challenges of putting together a suite can be overcome by a combination of careful planning, attention to process, implementation of best practices, and use of common tools. This approach facilitates team communication and allows planning, development, and testing artifacts to be shared between disparate groups. Don't expect it to be easy, but the rewards make it worthwhile. STQE

Sam Guckenheimer (samg@rational. com) *has been involved in software quality engineering, development, and marketing for twenty years. As Senior Director, Marketing, Automated Testing at Rational, he was a member of the product management team for Rational Suite TestStudio.*

Using Find and Fix Rates to Monitor a Project

This graph shows the number of weekly newly-found and newly-fixed defects, either within an iteration or for the project life. Note that these are number of defects opened and closed. Accordingly, when the find rate is above the fix rate, outstanding defect backlog is increasing and conversely, when fixes exceed finds, the backlog is shrinking.



Before code freeze in a project iteration, you expect to see a steeply-rising find rate. A high rate may indicate inattention to unit testing and quality problems to come. A low rate may indicate that testing is blocked or misfocused; risk of defect detection bottleneck later.

At the same time, you expect to see a much lower fix rate, also positively sloped. A high rate may indicate that developers are focusing on defects rather than completing necessary features. A low rate may mean developers are missing easy opportunities for early defect removal.

After code freeze, you expect to see the find rate level off (until the bits are exposed to new test groups).

Look for the crossover point, when fix rates start exceeding find rates. Obviously, this is essential to clear the defect backlog. If the crossover doesn't happen when expected, you may need to triage bugs more aggressively. If cross-over happens too soon and the fix rate greatly exceeds the find rate, it may be a sign of inadequate testing.

As you approach completion, find rates should begin falling steeply.

Fix rates should also decline as the backlog is cleared. The slope should appear roughly parallel to the find rate. At this point, changes should be carefully considered and approved to guard against introducing new defects. However, a fix rate that declines too fast may be a warning sign that outstanding defects are complex or subtle.

CHART BY ANNIE BISSETT

FIGURE 3 **A sample find and fix rate graph**