Agile Software Testing in a Large-Scale Project

David Talby and Arie Keren, *Air Force, Israel Defense Forces*
Orit Hazzan and Yael Dubinsky, *Technion–Israel Institute of Technology*

Quantitative and qualitative data from a large, real-world XP project validates agile testing practices and offers guidelines in four key development areas.

Agile software development in general, and Extreme Programming (XP)[1] in particular, promote radical changes in how software development organizations traditionally work. This is particularly true in software testing. Agile development completely redefines quality assurance work—from formal roles to day-to-day activities—making some traditional QA responsibilities and outputs irrelevant.[2–5] Agile development also eliminates the backbone of many organizations' software development cycle and its accompanying jargon, including "delivering to QA," "meeting the QA criteria for delivery," and "prioritizing bugs." Agile practices require full integration of testing and development, which can alter both organizational structures and high-level product-release policies. As a result, fully adopting agile quality practices is slower and more difficult than adopting agile programmer-oriented practices, such as pair programming and test-first programming, which are assimilated only within the development team.[6]

Currently, few real projects that implement full agile quality processes—and thus undergo these sometimes radical changes—have reported substantial findings. Here, we present and analyze new data from a real, large-scale agile project to develop a business-critical enterprise information system for the Israeli Air Force (IAF). Our results offer new evidence that agile testing practices actually work, dramatically improving development quality and productivity in line with Kent Beck's flat cost-of-change curve.[1] We describe the organization's successful practices and guidelines in four key areas: test design and activity execution, working with professional testers, planning, and defect management. This article closes several gaps in existing publications,[1,2,7] most notably by describing how to effectively use professional testers and how to thoroughly acceptance-test a system that's too large and complex for a single customer to specify.

Project overview

The IAF's enterprise information system is critical to daily operations and information security. As a result, the target system is highly complex and must be of the utmost quality. The IAF considered the project risky both because of its size and because it was the first project of this scope to be developed in the or-

0740-7459/06/$20.00 © 2006 IEEE

ganization using an XP-based method. Consequently, many different stakeholders closely scrutinized the development process.

To manage risks and provide full and timely information about the project's progress, the team developed a set of metrics,[8] consistently collected and analyzed data, and held regular reflection meetings and formal debriefings. External consultants helped with and augmented these efforts.

We derived our data from the study project's first four releases (over eight months). Our quantitative data is the same that the project team used for ongoing project tracking and decision making, so it's of high quality and validity. Our qualitative data includes recordings of formal reflection meetings and debriefings, along with data from questionnaires, interviews, and an analysis of formal organizational documents depicting and explaining project decisions. To our knowledge, this is the most comprehensive data set from an actual project using agile quality practices that has published its results to date.

Conforming to XP

The project team implemented many of the standard XP practices,[1,7] which we've italicized in this and the next section.

XP's basic practice is *short releases*. The team delivered working software at each iteration, which typically lasted two weeks. At the end of each iteration, the team formally presented the system to the customer for feedback. The team fully *tested* the system every two weeks. Team members fully specified, coded, and tested each new feature during the iteration. Moreover, developers performed full regression testing at every iteration, fixing all known defects. The goal here was to achieve several measurable testing benefits: few new and open defects, a short average time for defect fixes, and short average defect longevity.

The team used the *planning game* practice to plan project iterations and releases.[1] In the planning game, the entire team meets and the customer describes and prioritizes the stories that the system's next release should implement. The team then breaks the stories down into development tasks and estimates their cost. Planning's goal is to keep a *sustainable pace* and define *refactoring* activities, thus achieving *simple design*.

From the project's start, there was significant *customer collaboration* and fruitful communication among teammates. The customer participated in all planning games at both the release and iteration levels. In addition to reviewing the system during formal meetings, the customer regularly viewed in-development features and discussed them with analysts, developers, and testers. Formally, however, the customer belonged to a different organizational unit and had other duties.

The project also employed XP's *whole-team* and *sit-together* practices. From day one, the project team met in the same room and was managed as a single, coherent team. This team included developers, architects, a tester, and business analysts. The exception to the whole-team practice was the customer, who had a workstation in the team room but used it only occasionally. These team-centered approaches resulted in two measurable test-related benefits. First, the defect-management overhead was lower than in traditional projects, where the QA and development teams are separate entities. Second, the project produced fewer false defects ("false alarms") than traditional projects do.

Team members promoted an *informative workplace* using story cards on the wall, which describe the details of the different development tasks and their status. They also published up-to-date design diagrams and held a *stand-up meeting* every morning. Finally, the team practiced *continuous integration*, achieving one to three integration builds on an average development day.

Diverging from classic XP

The project differed from classic XP projects in three respects. First, in addition to code and tests, the team was required to produce detailed specifications as a development-process output. This was required due to organizational policies and obligations, as well as because the project had to produce semiformal specifications for a metadata repository[9] to enable automatic code generation. So, completing a feature in this project meant completing its specifications, code, and tests. Team members were thus encouraged to use *pair programming*, *collective ownership*, and *coding standard* practices for not only code but also specifications and tests.

Second, unlike in classic XP, the project's acceptance testing (also known as user testing)

Few real projects that implement full agile quality practices have reported substantial findings.

How an organization promotes developer testing is a major issue in shifting to agile development.

was far too complex for the customer to fully specify. According to the project's leaders, acceptance testing would require the equivalent of two full-time personnel. This finding led the organization to use professional testers—a common choice in large-scale projects. So, the project modified the *customer tests* practice to retain its essence while making it practical: Do thorough acceptance testing, but not only by the customer.

Third, the project modified *test-driven development* to focus on the continuous creation of automated acceptance tests rather than unit tests. In classic XP, code is developed by first writing a (failed) unit test, then writing the code to make it pass, and so on. This project required a solution for automating the massive amount of constantly generated acceptance tests. The solution was a tool—well integrated in the project's development environment—that simulates an end user working on the complete multitier system.[10] Unlike classic XP, which automates acceptance tests as coded unit tests, in this critical project, the organization considered only real simulation to be adequate acceptance testing. So, while the project still used unit testing, writing test scenarios directly into the automated acceptance-testing tool was preferred whenever possible. In this respect, test-first development meant planning test scenarios before coding and never considering a feature "complete" before all its tests were written and passed.

Each of the three exceptions to standard XP procedures is a typical issue for large-scale projects. Nonetheless, such differences must be kept in mind when generalizing our agile development results.

Test design and execution

Making sure that testing is everyone's ongoing business is a big challenge. This project tackled it by making testing part of each team member's work, and a key measure of both team and personal productivity.

Everyone tests

In a traditional project, "everyone is responsible for quality,"[2] but in agile projects, everyone actually writes tests. This includes all developers, business analysts, and even the customer. For the IAF, this was a radical shift from existing practices. The organization implemented the practice after developers received the required extra training, but the customer took part only in defining and reviewing some test cases.

Having everyone test offered several project advantages. First, it eliminated the team's reliance on the single, assigned project tester, who would have otherwise constituted a major bottleneck. This proved highly important because—for reasons unrelated to the project—the tester was replaced twice during the first three releases. Second, because developers were responsible for writing tests for each new feature, their test-awareness increased and they prevented or quickly caught more edge cases while they worked.

Finally, because people knew that they were required to test their specifications and code, they designed them for testability.[11] That is, where appropriate, developers added special features or design considerations to the code to make hard-to-test features testable. Such coordination tasks are often difficult and neglected when developers and testers work as separate teams.

Product size = test size

Measuring progress is also crucial; the project team tackled it by defining the project's product size metrics as the number of regression test steps run at each iteration.[8] The "running and tested features" metric[12] inspired this approach. The project's leaders chose this approach for two reasons. First, because test size is more highly correlated with complexity than are lines of code or specifications, it's a better approximation of product size. Second, such a metric sends a strong message to the team: only features that have full regression testing at each iteration are counted as delivered product size.

Reaching and increasing the product size metric is a nontrivial team goal; in this case, the team performed it successfully in 12 out of the 13 iterations we studied. Figure 1 plots the product size metric over this period of time.

Untested work = no work

Another related factor is how the organization promotes developer testing. This is a major issue in shifting to agile development, particularly in organizations where testing is poorly perceived. The key to success is similar to that with product size metrics. In this case, untested work equals no work—in all chan-

nels and under all conditions. The case is similar for task completion. At each iteration summary, the team reviews the iteration's tasks to investigate incompletion or time estimation errors. The team considers tasks lacking full written tests as incomplete—that is, failures—even if they already "work" and were presented to the customer.

In response to a written questionnaire administered after two releases, over 90 percent of team members ranked acceptance and system tests as "highly important"; the remaining 10 percent ranked them as "important." Furthermore, 33 percent were either "interested" or "very interested" in taking a leading role in acceptance testing, and almost 60 percent were "interested" in leading system testing.

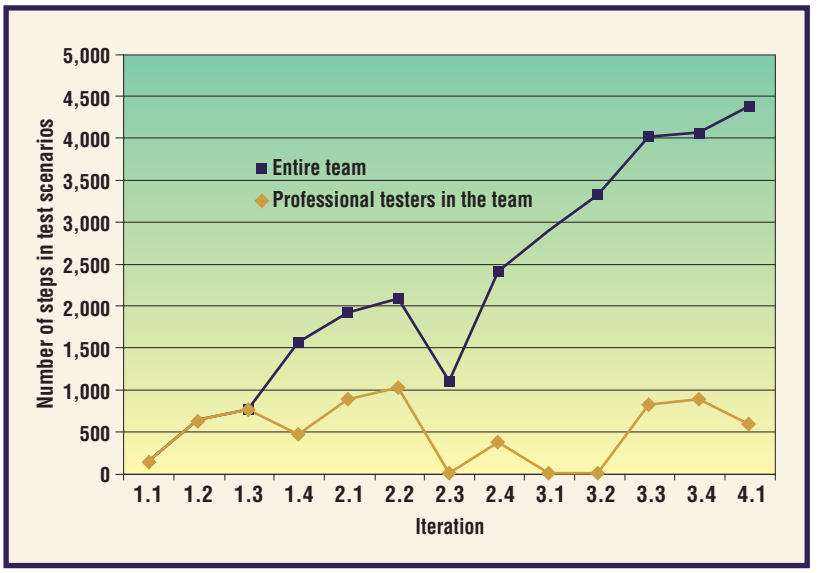

Figure 1. Product size per iteration. The team ran more passing test scenarios—reflecting a bigger product—on all iterations except one. Note that all tests must be rerun on each iteration and are not "accumulated."

Working with professional testers

In agile projects, everyone on the team fully tests their own work. So, professional testers add value not through more testing, but by writing some of the developers' tests, thus freeing them to code more new features. Professional testers can also create better test scenarios, although this ability varies greatly among individuals. Using professional testers nonetheless raises two key challenges for organizations adopting agile development: testing bottlenecks and coordinating the testing work among testers and programmers.

Easing bottlenecks: Code less, test more

Dealing with testing bottlenecks is relatively straightforward: developers simply code less and test more. Figure 1 illustrates this, comparing the product-size metric points attained by tests that the entire team ran to the points attained by the team's testers alone. The graph lines show the various phases the project went through. At first, only the tester was running tests. This became impractical within a few weeks as the software grew in size; at that point, most tests were handed over to other developers. The only iteration in which the total size decreased was when the tester was unexpectedly removed from the team (owing to an emergency in another project). However, the team recovered and took over testing the following week. Later, for the third release (3.3 in figure 1), a new tester joined the team and tester-generated points increased again. In this process, the team proved capable of coping with any tester availability—tester absence affected development speed but didn't compromise full, continuous testing.

Encourage interaction over isolation

Initially, the project tester didn't work with other team developers but instead wrote cross-feature tests and automated the customer-defined use cases. The testing personnel manager—who wasn't an XP team member—required this separation for two reasons.

First, common wisdom states that testers must be independent of programmers to prevent implementation details from affecting their test designs. This *contamination* argument maintains that testers can't write adequate tests for a feature if a programmer has already explained the testing requirements. Second, one of testing's aims is to verify that detailed specifications are up-to-date. So, testers traditionally write tests according to the specifications, not to what someone has "unofficially" told them. An unspecified feature is viewed as a kind of bug.

These two claims form many QA professionals' core objection to agile testing.[13] The claims target basic XP principles—whole-team and developer testing, in particular—and insinuate that they're unsuitable for producing quality software. The counterargument isn't that these claims are false but simply that the alternative is better overall. Published reports on professional testers' experience in agile teams support this.[4,14,15] They also emphasize that testers must change their mind-set if

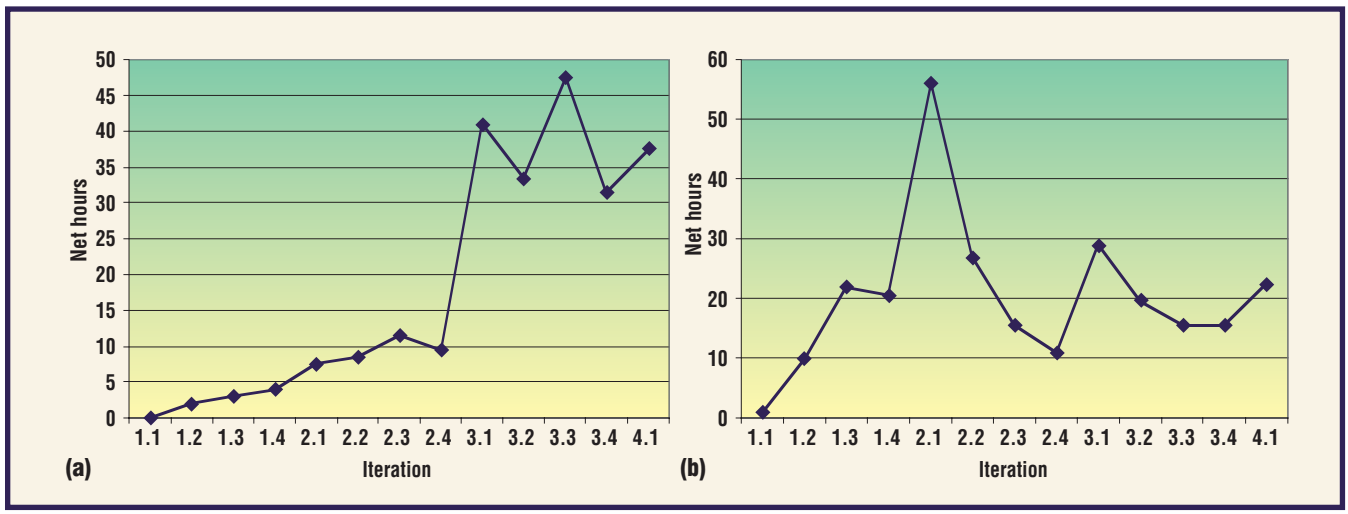

Figure 2. Testing and defect repair. The study project considered both regression testing and defect fixing as global overhead. (a) Net hours devoted to running and maintaining regression tests. (b) Net hours devoted to fixing defects.

they're to successfully integrate into an agile team; our findings support this as well.

In our study, testers working independently found many minor mismatches between written specifications and the system, and few actual bugs. The mismatches were tracked and resolved as bugs; they were usually trivial, taking only minutes to fix. However, because developers tested each task as they wrote it, few actual system bugs remained at this stage. So, working with testers in this way was highly inefficient: in one extreme case, the tester found only two actual bugs after three weeks of testing a complex feature.

These results reassured the project's management that developers could effectively test the software themselves. In turn, the organization decided to integrate the tester into the project team. The tester began to work on tasks with pairs of developers, writing tests in parallel with coding after they had all inspected the feature's specifications and generated a set of test cases accordingly. Although we haven't yet collected data on the method's effectiveness, it does seem clear that, in this agile project, continuous interaction is the more productive choice.

Activity planning

Planning for quality activities involves time-allocation challenges in feature testing, regression testing, and repairing defects.

Integrate feature testing and coding

In agile development, feature testing is always an integral part of each feature's development and time estimation. Testing and coding time are usually equal: if feature A is estimated to require five hours of specifications and 10 hours of coding, for example, it would typically be allocated 10 hours for writing and running tests. This estimation can change on a per-task basis, such as when a team knows in advance that a given feature is more difficult to test than others or requires refactoring of existing tests. As a rule, features are time-estimated to completion. Feature A, for example, would be planned as a single, 25-hour task. This again stresses that no task is considered complete until its tests are written and running.

Regression testing: Divide and conquer

In our study project, regression testing was considered as global (rather than personal) overhead and allocated as a global time period. This works because regression tests are usually run on each iteration's last day, with the testing work divided among team members. Figure 2a plots the net number of hours for running and maintaining regression tests in each iteration. As the figure shows, during the first two iterations, the variance between iterations was small enough to make this global approach the correct choice. The net time required of the entire team at each iteration was about half a day; this changed over time, owing mainly to substantial growth in team size in the third release.

Allocate bug-fix time globally

The project also allocated a global time pool for fixing defects; experience and available data thus far indicates that this was a good choice for two reasons. First, estimating the time to fix a defect is extremely difficult

because defect complexity is unpredictable. At the same time, as figure 2b shows, the law of averages applies here, indicating that we can reasonably predict the time required for fixing all defects in a forthcoming iteration. We can also adjust this much-more-stable estimate at each iteration's planning game according to the number of open defects at the time (see, for example, iterations 2.1 and 2.4).

Second, planning defect resolution as an individual task results in high overestimates. In the single iteration that allocated time on a per-defect basis, each defect was estimated to require at least four hours, with the highest single defect estimate being nine hours. Such overestimation occurs because a single, unexpectedly complex defect might well require that much time. But, as the data shows, it's highly improbable that many unrelated defects will simultaneously require that much time.

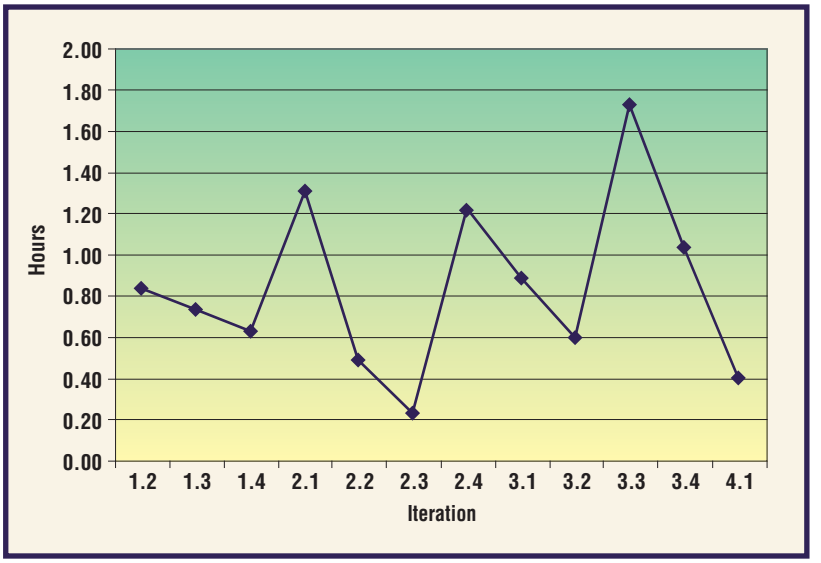

Figure 3. Average net time to fix a defect per iteration.

Defect management

Defect management involves two major challenges: managing workflow and selecting and scheduling the defects to fix.

Use a team-centered defect-management approach

Defect management workflow is much simpler in agile projects than in traditional ones in three ways:

- Anyone can open a defect (once its developers declare that the defective feature is "done," it's considered a defect).
- Anyone can close a defect, after fixing it and running the related tests.
- Anyone who finds a defect also assigns it to someone to be fixed.

Team members routinely view their own defects list and fix the defects. If they're assigned a defect by mistake—for example, a defect related to a subject they know nothing about—they can reassign it. This eliminates a sizable overhead from team leaders,[4] who traditionally are responsible for this task. For example, the team leaders on two comparable traditional projects at the IAF estimated that they devote one to three hours per week to defect management, compared to zero for the agile project's leaders.

Team-centered, XP-based development is key to enabling project members to successfully assign defect fixes to each other. XP's whole-team and sit-together practices require that everyone, including the tester, sit in the same room. The informative-workplace practice requires that everyone participate in the planning game and daily stand-up meetings, so people know each other and everyone's respective knowledge areas. Given this, team members can usually identify the right person to fix a given bug. This doesn't occur when the testing team is separate from the development team.

The same practices also reduce false defect reports. Such reports either duplicate existing defect reports or are created owing to developer misunderstanding. During our study period, only 29 false defects were recorded, constituting 7.5 percent of all defects. Sixteen of these false defects were recorded in a single iteration (2.3), in which the tester worked separately from the team. These numbers are low; during the same periods, our two reference projects at the IAF reported four to eight times more false defects.

Fix defects as soon as possible

Letting team members assign bug-fixing responsibilities requires that the average bug-fix time is short—in this study, it was slightly over an hour (see figure 3). This speed is crucial to ensure that precise load balancing of bugs among team members is unimportant. When a bug could require a day or more to fix, team leaders must assign such work to prevent bottlenecks and ensure that deadlines are met. In this case, the average bug-fix time remains

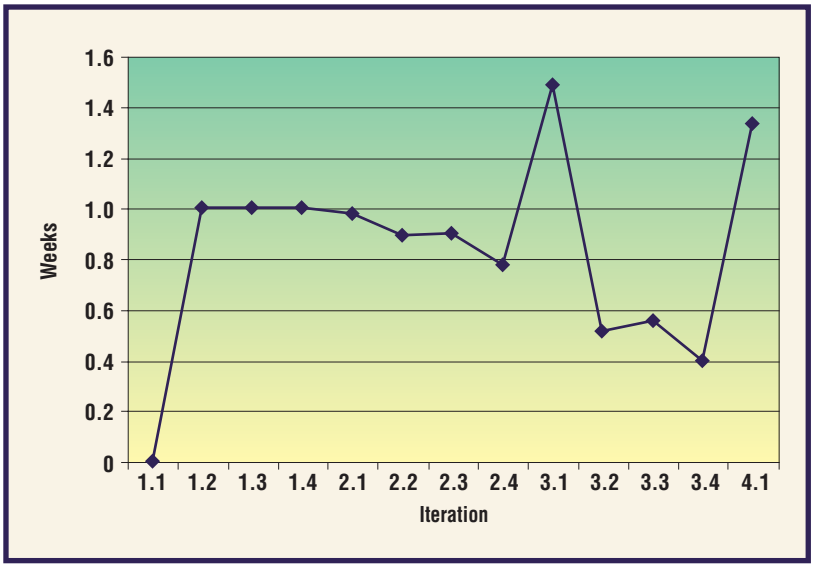

Figure 4. A defect's average longevity per iteration. The average is for all defects, regardless of severity.

relatively constant, even as the application grows significantly in size and complexity. This agrees with the flat cost-of-change curve,[1] a central benefit of agile development.

The average time is based on developer reports of the total net time they spent working on defects (developers complete daily reports on time spent on each task). The team achieved this very low average—an order of magnitude or more lower than most projects, including those in the same organization—for two reasons. First, there's little overhead to reproduce defects because testers and programmers work on the same baseline. Second, team members fix defects as soon as they're found, when the relevant code is still fresh in the developers' mind.

In traditional projects, a feature-complete version is sent to the QA group, which sends back a large stream of defects of varying importance. Because the effort to repair defects always greatly exceeds the available resources, a lengthy process of prioritizing defects begins. The team fixes the most urgent defects first, during time slots preallocated for this activity during project planning.

Agile projects replace this process with a simple rule: fix every defect as soon as you discover it. This is required because the team releases software to its customers at every iteration, and releases shouldn't, in principle, contain known bugs. In our case study, the rule was that defects should be fixed as soon as possible—this does not necessarily mean immediately, however, simply because the right person to fix a defect wasn't always immediately available. In addition, teams sometimes find defects during regression testing, at the end of an iteration, that they can't fix in time. Given this, our project's team devoted the beginning of each iteration to fixing the previous iteration's open defects.

Figure 4 shows a defect's average longevity over time. During the first four iterations, the team fixed all defects within a week (usually at the start of the next iteration after their discovery). The team relaxed the pace slightly in the next two releases. However, the average remained low and relatively constant relative to traditional projects (in the organization and elsewhere). Moreover, these averages relate to *all* defects, not only critical or urgent ones.

Fixing all defects as soon as possible has several advantages. First, as we discussed earlier, defects require far less time to fix with this approach. Second, working on a clean and highly stable code base makes new development faster. Third, besides avoiding the overhead of prioritizing and planning defect fixes, this rule avoids unpleasant customer negotiations over which defects to solve. Such negotiation is often forced upon both sides when a project's delivery date approaches and—because traditional development projects test late in the process—it becomes clear that not all defects will be fixed before the deadline. In the project studied here, the team fixed all defects routinely and, given its perceptions about product quality, planned no "stabilization period" toward the official delivery date.

Another significant difference from common practice that often surprises traditional practitioners is that agile projects have no concept of defect severity. This relieves developers of both the pressure to stabilize the system just before the delivery date and the need to coerce the customer into dictating which defects to fix (and which to not fix). So, the only consideration regarding a given defect is whether to fix it. If the answer is no, the defect isn't opened. If the answer is yes, it's fixed when it's cheapest to do so—right away.

Our study's data and analysis are useful both to practitioners wishing to establish new agile teams and to researchers looking for high-quality, real-world data. In addition to illustrating applicable tactics, our

data shows that agile testing works. The project team cut by an order of magnitude the time required to fix defects, defect longevity, and defect-management overhead. Even on such a large-scale project, the team achieved full regression testing at each iteration and developer testing. It also resolved all defects over a significant time period that included both personnel changes and team growth.

Implementing the presented process required a new mind-set at personal and organizational levels, and it remains a challenge to expand its adoption to other IAF projects. One key motivator for such expansion, in the IAF and industry-wide, is the continued publication of quality data on large-scale agile projects. We strongly encourage and call for further investigation on these issues, and, in particular, on long-term agile projects, projects spanning multiple coordinating teams, and projects that are concurrently developing and maintaining production systems.

References

1. K. Beck and C. Andres, *Extreme Programming Explained*, 2nd ed., Addison-Wesley, 2005.
2. L. Crispin and T. House, *Testing Extreme Programming*, Addison-Wesley, 2002.
3. C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing: A Context-Driven Approach*, John Wiley & Sons, 2002.
4. R. Leavitt, "Let's End the Defect Report-Fix-Check-Rework-Cycle," 2005, presentation, 2005; www.rallydev.com/kportal.jsp?doc=wp6.
5. B. Pettichord, "Agile Testing Challenges," *Proc. Pacific Northwest Software Quality Conf.*, 2004; www.io.com/~wazmo/papers/agile_testing_challenges.pdf.
6. V. Ramachandran and A. Shukla, "Circle of Life, Spiral of Death: Are XP Teams Following the Essential Practices?" *Proc. XP/Agile Universe*, LNCS 2418, Springer, 2002, pp. 166–173.
7. K. Beck, *Test Driven Development: By Example*, Addison-Wesley, 2002.
8. Y. Dubinsky et al., "Agile Metrics at the Israeli Air Force," *Proc. Agile 2005 Conf.*, IEEE Press, 2005, pp. 12–19.
9. D. Talby et al., "The Design and Implementation of a Metadata Repository," *Proc. Int'l Council on Systems Eng. Israeli Chapter Conf.*, Incose-IL, 2002.
10. D. Talby et al., "A Process-Complete Automatic Acceptance Testing Framework," *Proc. Int'l Conf. Software—Science, Technology and Eng.* (SwSTE 05), IEEE CS Press, 2005, pp. 129–138.
11. B. Pettichord, "Design for Testability," *Proc. Pacific Northwest Software Quality Conf.*, 2002; www.io.com/~wazmo/papers/design_for_testability_PNSQC.pdf.
12. R. Jeffries, "A Metric Leading to Agility," *XP Magazine,* 14 June 2004; www.xprogramming.com/xpmag/jatRtsMetric.htm.
13. M. Stephens and D. Rosenberg, *Extreme Programming Refactored: The Case against XP*, Apress Publishers, 2003.
14. L. Crispin, "Extreme Rules of the Road: How an XP Tester Can Steer the Project toward Success," *STQE/Better Software Magazine*, July/Aug. 2001, pp. 24–29; www.testing.com/agile/crispin-xp-article.pdf.
15. B. Pettichord, "Don't Become the Quality Police," StickyMinds.com, 1 July 2002, www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=ART&ObjectId=3543.

About the Authors

David Talby is a software team leader in the Israeli Air Force's MAMDAS operational software development unit, and a doctoral student in computer science at Hebrew University, Jerusalem. His research interests are in parallel computer scheduling and workload modeling, and software process improvement initiatives focusing on agile methods. He received his MSc in computer science and MBA in business administration from Hebrew University. Contact him at the School of Computer Science and Eng., Hebrew University, Givat Ram, Jerusalem 91904, Israel; davidt@cs.huji.ac.il.

Orit Hazzan is an associate professor in the Department of Education in Technology and Science at Technion–Israel Institute of Technology. Her research interests are in human aspects of software engineering, particularly relating to agile software development and XP development environments. She is coauthor (with Jim Tomayko) of *Human Aspects of Software Engineering* (Charles River Media, 2004). She received her PhD in mathematics education from the Technion–Israel Institute of Technology. She's a member of the ACM and the ACM Special Interest Group on Computer Science Education. Contact her at the Dept. of Education in Technology and Science, Technion–Israel Inst. of Technology, Haifa 32000, Israel; oritha@techunix.technion.ac.il.

Yael Dubinsky is a visiting member of the human-computer interaction research group at the Department of Computer and Systems Science at La Sapienza, Rome, and an adjunct lecturer in the Computer Science Department at the Technion–Israel Institute of Technology. Her research examines the implementation of agile software development methods in software teamwork, focusing on software process measurement and product quality. She received her PhD in science and technology education from the Technion–Israel Institute of Technology. Contact her at 7 Pinchas Rozen St., Haifa 34784, Israel; yael@cs.technion.ac.il.

Arie Keren leads the Magnet division's systems-engineering group in Israel Aircraft Industries, and previously led the MAMDAS software development unit's C4I engineering group in the Israeli Air Force. His research interests are in systems analysis, architecture, and development processes. He received his PhD in computer science from Hebrew University, Jerusalem. Contact him at 43 Lilach St., Modiin-Reut 71908, Israel; ariekk@netvision.net.il.