RETIRING LIFECYCLE DINOSAURS

Using Adaptive Software Development to meet the challenges of a high-speed, high-change environment

by Jim Highsmith

Contrary to what you've heard all of your life, form *doesn't* follow function. Form follows failure.

"The form of made things is always subject to change in response to their real or perceived shortcomings, their failures to function properly," writes Henry Petroski, civil engineering professor and author of *The Evolution of Useful Things*. "This principle governs all invention, innovation, and ingenuity; it is what drives all inventors, innovators, and engineers." In a similar vein, author Stuart Brand decries the oft quoted "form follows function" as an illusion, writing in *How Buildings Learn:* "[Louis] Sullivan's form-follows-function misled a century of architects into believing that they could really anticipate function."

►►QUICK LOOK

- Rethinking traditional project management practices
- 6 characteristics of an adaptive lifecycle

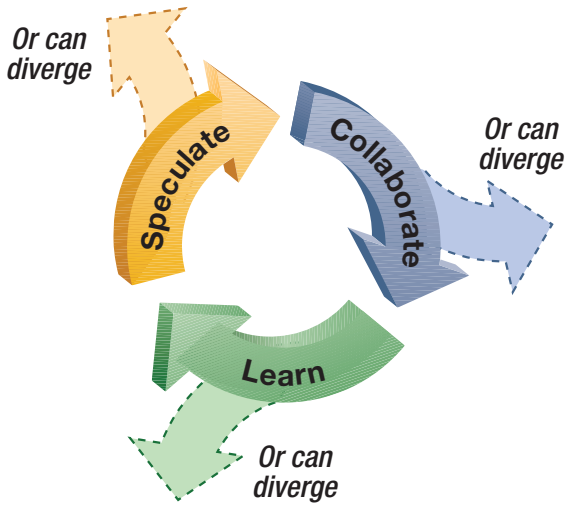
What can we learn from all this? First, as Brand points out, that none of us can really anticipate function very well. For all the many books on requirements engineering, the best way to determine how a product should evolve—Petroski writes about zippers and forks—is to use it. Effective, lean requirements analysis practices are still very necessary, as long as developers realize that those analyses are insufficient by themselves.

Second, iteration—building, trying, succeeding, failing, rebuilding—governs successful product development, particularly as the competitive environment becomes more extreme. "Good enough" requirements need to be followed by some form of usage (possibly utilizing a customer-driven prototype), and then followed by evolutionary changes to the requirements based on that usage.

Although a number of modern software development lifecycles have adopted an iterative approach, they still miss the mark in dealing with the messiness and unpredictability of complex (high-speed, high-change) environments. While the cycles are iterative, the fundamental assumptions are still deterministic—they convey the impression of short waterfall lifecycles strung together.

But as the world becomes increasingly dynamic—as technological and business changes accelerate—static management practices are overwhelmed. The old world was one of optimization in which efficiency, predictability, and control dominated. The *new* world is one of adaptation in which change, improvisation, and innovation rule. This dichotomy—optimization versus adaptation—provides a distinct way of viewing the future of software project management.

We, the purveyors of the most momentous technological change period in the history of business, still don't "get it." Every publication—from *ComputerWorld* to *Business Week*—trumpets to business leaders that "business will never be the same." Why then, do we insist that software development and management practices won't undergo dramatic changes also? We seem to accept the idea that *everything* is changing to fit this 21st Century Internet world; *everything* is changing, that is, except for you and me, and the ways we're comfortable doing what we do. Does that make sense?



ANNIE BISSETT

FIGURE 1 The adaptive lifecycle

In a Dilbert cartoon, Wally the co-worker bemoans the fact he has no impact on results, but he can take solace in having "process pride." As Wally says, "Everything I do is still pointless, but I'm very proud of the way I do it." **Maybe it's time for a new outlook.** It's time for an era of results over process, understanding over documents, collaboration over control, adaptation over optimization.

Adaptive Software Development (ASD) is one of a growing number of alternatives to traditional, process-centric software management methods. ASD, Extreme Programming (XP), Lean Development, SCRUM, and Crystal Light methods—although different in many respects—are tied together by a focus on people, results, minimal methods, and maximum collaboration. They are geared to the high speed and high change of today's e-business projects.

Whether you're managing testing, leading a development team, or running an entire project, it's time to reconsider the values and assumptions that underlie their management. The practices of Adaptive Software Development are driven by a belief in continuous adaptation—a different philosophy and a different lifecycle, one geared to accepting continuous change as the norm.

In ASD, the static **Plan-Design-Build** lifecycle is replaced by a dynamic **Speculate-Collaborate-Learn** lifecycle (see Figure 1). This is a lifecycle dedicated to continuous learning—and geared to constant change, re-evaluation, peering into an uncertain future, and intense collaboration among developers, testers, and customers. (Note that it's not always a simple circle; even in an iterative process, one shouldn't be adverse to diverging in order to explore areas not considered before.)

ASD, rooted in an underlying conceptual base of complex adaptive systems theory, was designed for extreme projects in which high-speed, high-change, and uncertainty reign. Many, many projects are *not* extreme; but for those that are, ASD fits better than traditional software development approaches. (See the Webinfolink feature at the end of this article for more information on adaptive systems theory.)

A Change-Oriented Lifecycle

A typical project management cycle of **Plan-Deliver-Review,** based on an assumption of a relatively stable business environment, becomes overwhelmed by high change. The first step in that cycle, planning, is one of the most difficult concepts for engineers and managers to examine thoughtfully. Raised on the science of reductionism (reducing everything to its component parts) and the near-religious belief that careful planning—followed by rigorous engineering execution—produces the desired results (we are in control), the idea that there is no way to "do it right the first time" remains foreign to many.

The word "plan," when used in most organizations, indicates a reasonably high degree of certainty about the desired result. The implicit and explicit goal of "making a plan" restricts the project manager's ability to steer the results in innovative directions. While operating in this mode produces the results you planned for, they may not be the results you now need.

Speculation gives us more room to explore, to make

clear the realization that we are unsure, to allow us to deviate from plans without fear. It doesn't mean that planning is obsolete, just that planning is acknowledgeably more tenuous than it was in the past. It means we have to keep delivery cycles short and encourage iteration. Speculating doesn't mean abandoning planning, it merely acknowledges the reality of uncertainty. Too often, deviations from traditional plans are considered mistakes to be corrected rather than opportunities for learning. Speculation recognizes the uncertain nature of complex problems, and encourages exploration and experimentation. Speculating (difficult for my banking clients) allows us to admit that we don't know everything—and once we admit to ourselves that we are fallible, then learning (the third step in this cycle) becomes more likely.

The second conceptual component of an adaptive lifecycle is that of *Collaboration*. Complex applications are not built; they evolve. Complex applications require a large volume of information to be collected, analyzed, and applied to the problem—a much larger volume than any individual can handle alone. "None of us is as smart as all of us," write Warren Bennis and Patricia Biederman in *Organizing Genius*. While there is always room for improvement, most software engineers are reasonably proficient in analysis, programming, testing, and similar skills. But turbulent environments are defined in part by high rates of information flow and diverse knowledge requirements. Building an e-commerce site requires greater diversity of technology, business skills, and knowledge than the typical project of five to ten years ago. In this high-information-flow environment, in which one person or small group can't possibly "know it all," collaboration skills—working jointly to produce results or make decisions—are paramount.

In his book *No More Teams, Mastering the Dynamics of Creative Collaboration*, author Michael Schrage defines collaboration as "an act of shared creation and/or discovery." The issue, says Schrage, isn't teams: it's "what kind of relationships organizations need to create and maintain if they want to deliver unique value to their customers and clients." Shared creation crosses traditional project team boundaries. It encompasses the development team, customers, outside consultants, and vendors. Collaboration then becomes the third key to building a more adaptable project management lifecycle. Teams must collaborate on technical problems and business requirements. Teams need to improve their joint decision-making ability, and more decisions must be delegated to the team level, because rapid change (and tight schedules) precludes the traditional **Command-Control** style of decision making.

That decision making depends on the third conceptual component of this cycle: *Learning*. We have to test our knowledge constantly—using practices like project retrospectives and customer focus groups. Furthermore, these review practices should be done after each iterative cycle rather than waiting until the end of the project. The quality of learning derived from practices like project retrospectives provides a key indicator about the true commitment to learning in an organization, and, therefore, a key to its adaptability.

Examining beliefs, assumptions, and mental models comes hard for many organizations. Learning about oneself—whether personally, at a project team level, or at an organizational level—can be painful. Project postmortems, for example, are simple in concept, asking periodically about what went right and what went wrong and what

PERSPECTIVE

Of Social Work and Software

Even as a child, John Sarkela saw that the "Plan-Deliver-Review" model wasn't always very realistic.

"My father was a social worker," he recalls, "and I remember there being parallels. Helping people build something workable out of chaotic unpredictable circumstances…in my father's line of work it was all about helping people learn and respond to cognitive dissonance in their world models, and make adaptive changes."

In that sense, it wasn't far off from software development. "It's the same process," he points out, "no matter which discipline you're in. You're dealing with constantly changing requirements, a shifting understanding of those requirements, and re-targeting your development efforts." The project—be it a new database application or a life in transition—must accommodate that new information, that new understanding.

Sarkela, chief technical officer for the new metamedia publishing development company Fourth Estate, acknowledged long ago that software invention—and life in general—is too unpredictable to work in a rigid textbook grid.

"I've embraced adaptive software development processes pretty much throughout my career," he says. "I've had to make compromises, depending on what projects and companies I've been working with, but the Speculate/Collaborate/Learn model has been pretty fundamental in my history as a developer and consultant."

***Speculating* is necessary to set a direction, focus on a deliverable, and start moving. And once you're in motion, *collaboration* is vital to deepening your initial understanding of the product and the process. Keep in mind, says Sarkela, that these are not singular events; speculation and collaboration are recurring activities throughout the project.**

changes need to be made in the future. Postmortems are difficult for most organizations, however, because they can degenerate into blaming and politics—becoming vehicles for "who do we blame," rather than "how do we learn." To become an adaptive, agile organization, one must first become a learning organization.

Characteristics of an Adaptive Lifecycle

An adaptive lifecycle has six basic characteristics: Mission Focused, Component Based, Iterative, Timeboxed, Risk Driven, and Change Tolerant.

For many e-business projects the final results may be fuzzy in the beginning, but the overall *mission* that guides the team is well articulated. Mission statements act as guides that encourage exploration in the beginning, but narrow over the course of a project. A mission provides boundaries rather than a fixed destination. Without a good mission and a constant mission refinement process, iterative lifecycles become oscillating lifecycles—back and forth with no progress. Mission artifacts (there may be several types) not only provide direction, but are used in making critical project trade-off decisions. Mission artifacts that don't help make decisions are mere fluff.

The adaptive lifecycle focuses on results, not tasks; and the results are identified as application components. *Component* in this context defines a group of features (or deliverables) that are to be developed during an iterative cycle. While documents (for example a data model) may be defined as a deliverable component, they are always secondary to a software feature that provides direct results to a customer.

Iterative cycles emphasize "re-doing" as much as "doing." In manufacturing, quality programs try to drive out "rework" as costly, the result of a broken process. But product development (software or other) varies considerably from stamping out the exact same thingamabob on an assembly line. Actually, the thingamabob was probably designed with an iterative process. Components normally evolve over several iterative cycles as customers provide feedback.

The practice of *timeboxing*, or setting fixed delivery times for iterative cycles and projects, has been abused by many rapid application development (RAD) proponents because they use time deadlines incorrectly. Time deadlines that are used to bludgeon staff into long hours or cutting corners on quality are a form of tyranny; they undermine the collaborative environment that adaptive development strives to achieve. It took several years of managing RAD projects (I'm slow) before I realized that timeboxing was minimally about time—it was really about focusing and forcing hard tradeoff decisions. In an uncertain environment in which change rates are high, there needs to be a periodic forcing function to get something finished. In addition, timeboxing forces a project team and their customers to continuously re-evaluate the validity of the project's mission profile—scope, schedule, resources, and defects. Project teams that can't—or won't—juggle tradeoffs will never succeed in extreme environments.

As in Barry Boehm's spiral development model, the plans for adaptive cycles are driven by analyzing the critical *risks*. Adaptive development is also *change tolerant*, viewing the ability to incorporate change as a competitive advantage, not something to be summarily viewed as a "problem."

"The goal is always—or should be—to get some working code as soon as you can. Anything that stands in the way of that is a problem." And analysis paralysis is often Problem Number One, says Sarkela. "The artificiality of the old development lifecycle methodologies, thinking that you could do a preliminary ninety-two-page analysis and thereby really understand something, really makes paralysis inevitable in traditional approaches." You end up with stacks of paper and no code, he warns, because you're afraid to move ahead without an ironclad understanding of the problem.

You've got to have the courage to go ahead, Sarkela stresses, in spite of the real world's uncertainties. "And you can do that only when you don't have so much invested in your code that if it's wrong you're afraid to throw it out and start over, if need be." Sarkela likes to have just enough spec to know what he's building, and what the user's needs are.

Once that's in hand, he collaborates with the user to get something in place, using a "review-as-you-write" approach, with developers working in pairs. "Two people sharing ownership relieves some of the stress of being perfect," Sarkela says. "Giving up strict code ownership takes the ego out of the equation, and means that anyone on the team can go in and fix code the minute defects are found."

All of that helps get code into users' hands faster, and helps open up the *learning* process. "That may be the most important part of this model," says Sarkela. "You learn a lot faster, and a lot smarter, with something tangible to examine." With this working code and early feedback the hope is that—in software development, as in real life—subsequent iterations in our projects benefit from an ever-deepening functional understanding.

—A. W.

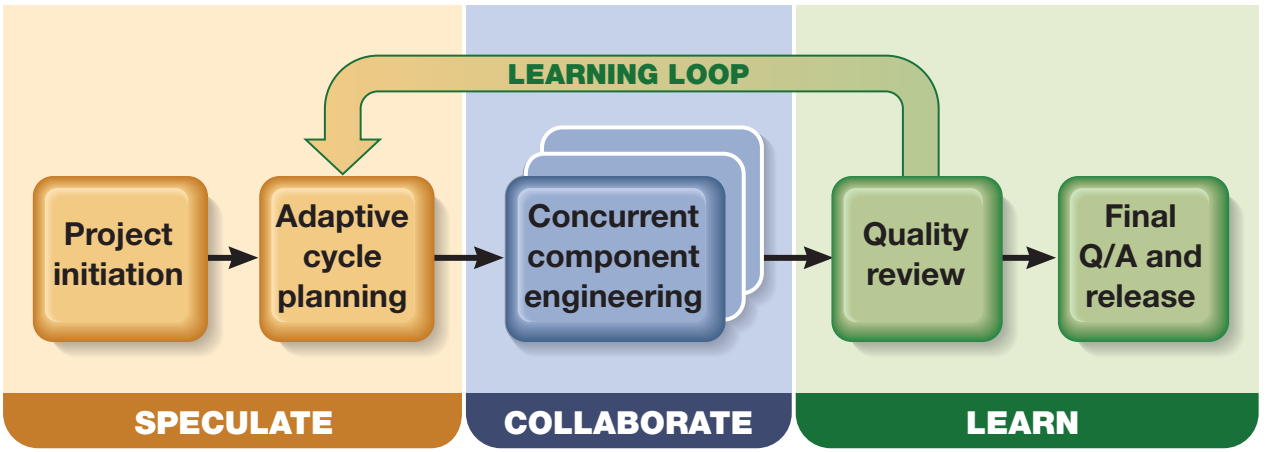


FIGURE 2 Adaptive lifecycle activities

The Basic Adaptive Lifecycle

While the iterative "Speculate/Collaborate/Learn" cycle is useful for outlining overall concepts, specifics are necessary to actually deliver a software application. Figure 2 shows the basic adaptive lifecycle and each of its components.

SPECULATE: Initiation and Cycle Planning

There are seven steps in adaptive cycle "speculating":

1. Conduct the project initiation phase.
2. Determine the project timebox.
3. Determine the optimal number of cycles and the timebox for each.
4. Write an objective statement for each cycle.
5. Assign primary components to cycles.
6. Assign technology and support components to cycles.
7. Develop a project task list.

Project initiation is similar to that used in any competent project management approach. It involves setting the project's mission and objectives, understanding and documenting constraints, establishing the project organization and key players, identifying and outlining requirements, making initial size and scope estimates, and identifying key project risks. Since speed is usually a major consideration in using an adaptive approach, much of the project initiation data should be gathered in preliminary Joint Application Development (JAD) sessions. For small projects, initiation can often be completed in a concentrated weeklong effort. Larger projects may require a "Cycle 0" for more extensive initiation work. (Cycle 0 setups differ from other cycles that have to deliver a tangible piece of an application to the customer; Cycle 0 involves preparatory deliverables but no pieces of the application.)

The second step is to set the timebox for the entire project. This timebox should be based on the scope, feature set requirements, estimates, and resources that result from project initiation work. Speculating doesn't abandon estimating; it just means accepting that any estimates are tenuous.

The third step is to decide on the number of cycles and assign a timebox to each one. For a small- to medium-sized application (<5000 function points), cycles usually vary from four to eight weeks. Some projects may need two-week cycles, while others might require more than eight weeks (although this is rare). The overall project schedule and the degree of uncertainty are two of the factors that determine individual cycle lengths.

After establishing the number of cycles and a schedule for each, team members perform the fourth step: developing a theme or objective for each of the cycles. Just as it is important to establish an overall project objective, each cycle should have its own theme. Cycle milestones are intended to force product visibility, because without visibility the product's defects and mistakes remain hidden. Without visibility, learning suffers.

A *Cycle* delivers a demonstrable set of components to a customer review process—it makes the product visible to the customer. Within the cycles, *Builds* deliver working components to an integration process—they make the product visible to the development team. With this increased visibility, testing is an ongoing, integral part of each development cycle—not some activity tacked on at the end.

The fifth and sixth steps assign components to cycles. The most important criteria for component assignment are that every cycle must deliver a visible, tangible result to an end user. Assigning components to cycles is a multi-dimensional exercise. Factors in the decision include:

- Making sure each cycle delivers something useful to the customer
- Identifying and managing high-risk items early in the project
- Scheduling components to accommodate natural dependencies
- Balancing resource utilization

ANNIE BISSETT

A spreadsheet is often the most effective tool for component-based cycle planning. (See Figure 3.) The first column contains all the identified components. There is also a column for each cycle. As shown in the figure, as decisions are made on the scheduling of each component, a team member checks the cycle column. Experience has shown that this type of planning—done as a team rather than performed exclusively by the project manager—provides better understanding of the project than a traditional task-based approach. Component-based planning reflects the uniqueness of each project. While there may be, and should be, "best practices" for activities such as analysis, object modeling, or design, what makes these activities unique to a project are the components to be delivered to the customer.

Finally, for those who may be uncomfortable without a task list, each component can become the target of a task: *Develop component A*. Additional tasks, not directly related to components but necessary for project completion, can also be added. There may be certain tasks added to the project plan in addition to the components, but the bulk of the plan is a "component breakdown structure," rather than a "work breakdown structure."

COLLABORATE:
Concurrent Component Engineering

Concurrent component engineering delivers the working components. Managers are more concerned about collaboration and dealing with concurrency than about the details of designing, testing, and coding. As mentioned earlier, for many projects today—involving distributed teams, varying alliance partners, and broad-based knowledge—how people interact and how they manage interdependencies are critical issues. For smaller projects, especially those in which team members work in physical proximity, concurrency can be handled informally. Dependencies can be handled by hallway chats and whiteboard scribbling. In larger projects, however, managing concurrency requires an advanced adaptive lifecycle whose description is beyond the scope of this article.

For small single-site teams, collaborative development can be enhanced by practices promoted by Extreme Programming. Pair programming, for example, encourages two people to work closely together: each drives the other a little harder to excel.

Collective ownership, another Extreme Programming practice, provides another level to the collaboration begun by pair programming. This approach, in which anyone on the team can change the code, encourages the entire team to work more closely together. Everyone—each individual, each pair—strives a little harder to produce high-quality designs, code, and test cases.

Relationships are the new bottom line in business, as Roger Lewin and Birute Regine write in their new book *The Soul at Work: Embracing Complexity Science for Business Success.* Relationships are important, they point out, "not simply for humanistic reasons, but as a way to promote adaptability and business success." Building alliances across organizations, collaborative problem solving and decision making, and sharing tacit knowledge are replacing rigor, control, and process as the building-block skills required to deliver successful projects.

LEARN:
Quality Review

The basic adaptive management approach is to maintain a focus on the project scope and objective, assign components to the delivery team, stand back and let the team figure out how to deliver the components, and maintain accountability through quality reviews. Quality evolves—not from micromanaging the process, but from establishing appropriate exit criteria and review practices. These feedback practices are key to learning.

There are four general categories of things to learn at the end of each development cycle:

- Result quality from the customer's perspective
- Result quality from a technical perspective
- The functioning of the delivery team and the practices they are utilizing
- The project's status

Providing visibility and feedback from the customers is the first priority in most projects. One vehicle for this is a

	C1	C2	C3	C4
Cycle Delivery Dates	**1-Jun**	**1-Jul**	**1-Aug**	**1-Sep**
Primary Features				
Order Entry	X			
Order Pricing		X		
Warehouse Picking		X		
Partial Order Ship			X	
Calculate Reorders			X	
System Interfaces			X	
Pricing Error Handling			X	
Security & Control				X
Technology Components				
Install Visual Basic	X			
Install Comm Lines			X	
Support Components				
Client/Server Arch		X		
Develop Conversion Pln		X	X	
High-Level Data Model	X			

FIGURE 3 A sample Component Cycle Plan

customer focus group. Derived from the concept of marketing focus groups, these group sessions are designed to explore a working model of the application and record customer change requests. They are facilitated sessions, similar to JAD sessions, but rather than generating requirements or defining project plans, customer focus groups are designed to review the application itself. In the end, customers relate best to a working application, not documents or diagrams. Where prototyping sessions involve individuals or small groups in helping to define an application, focus groups are more formal cycle milestones.

A second key to delivering quality products is reviewing technical quality. A standard vehicle for technical quality assessment is the technical review. A design review might occur at the end of a cycle, whereas code reviews or test plan reviews might occur during the cycle. The focus of reviews should be to learn—not to find fault.

The third feedback process is to monitor the team's performance. This might be called the people-and-process review. Project postmortems and end-of-cycle mini-postmortems are needed.

There are four basic postmortem questions:

- What's working?
- What's not working?
- What do we need to do more of?
- What do we need to do less of?

Postmortems tell more about an organization's ability to learn than nearly any other practice. Good postmortems force us to learn about ourselves and how we work.

The fourth category of review is not strictly related to quality, but is instead a review of project status. This review leads directly into a re-planning effort at the beginning of each subsequent cycle.

The basic questions are:

- Where is the project?
- Where is it versus our plans?
- Where should it be?

Determining a project's status is different in a component-based approach. In a waterfall lifecycle, completed deliverables mark the end of each major phase (a complete requirements document, for example, marks the end of the specification phase). In a component-based approach, rather than having a single completed document or other deliverable, the status often reflects multiple components in a variety of states of completion.

Recognize that not everyone will feel comfortable with this method. Many managers experienced in traditional approaches, for example, may feel a distinct loss of control and influence. Their "gut feelings" about progress have to be re-initialized.

The last of the status questions is particularly important: Where "should" the project be? Since the plans are understood to be speculative, measurement against them is insufficient to establish progress. The project team and sponsors need to continually ask, "What have we learned so far, and does it change our perspective on where we need to be?"

Toward the Future

Optimizing cultures believe in efficiency, control, and process rigor. But the Internet era has altered the fundamental premise on which these beliefs depend: predictability. No one would labor under the delusion that the results of a twelve-month e-commerce project (why one would be this long is another issue) could be predicted with any degree of accuracy—there are too many variables, changing too fast.

Optimizing cultures tend to see the world as black or white. *Adaptive* cultures, on the other hand, recognize gray. They understand that planning is tenuous and control nearly impossible. Adaptive cultures understand that success evolves from a succession of trying different alternatives, and learning from both success and failure. Adaptive cultures understand that learning about what we don't know is often as important as doing things we already know how to do.

Not every project is a complex one. In fact, in many organizations complex projects may be less than twenty percent of the total (but a very critical twenty percent). Solving those complex software development and testing problems does not mean abandoning good software management and engineering practices, but it does mean adopting a new perspective on their use. It means understanding that software development is not a mechanical process, but an organic, non-linear, non-deterministic one. It means altering some basic principles of how organizations work to solve complex problems, and adopting new practices geared to those beliefs. Shifting from an optimizing culture to an adaptive one means gearing up for the future, rather than lounging in the past. STQE

Jim Highsmith (jimh@adaptivesd.com) *is President of Information Architects, Inc., author of* Adaptive Software Development: A Collaborative Approach to Managing Complex Systems, *editor of* E-business Application Delivery, *and helps IT organizations and software companies adapt to the accelerated pace of development.*