CHANGE: EMBRACE IT, DON'T DENY IT

Tools and techniques inspired by software development can introduce the flexibility needed to make changes during product development with minimal disruption.

Preston G. Smith

OVERVIEW: *Change midstream in a product development project often means cost and budget overruns, schedule delays and product defects. Understandably, managers dislike change and have installed systems, such as phased development, to inhibit it. On the other hand, change—from understanding the customer better, from market dynamics and from technology advances—is connected inescapably with innovation, which management seeks. This article proposes an alternative: build a set of tools and approaches to accommodate change without undue disruption, thus reintroducing flexibility into product development. The flexibility techniques described in this article were inspired by the recent reintroduction of flexibility into the parallel field of software development by the agile software movement. They can be used defensively to deal effectively with imposed change or offensively to actually lead change.*

KEY CONCEPTS: *product development, design changes, product innovation, flexible development.*

Change from plans during a new-product development project is a topic that increasingly places developers and their managers in a dilemma. On the one hand, change is becoming increasingly commonplace. Customers, who are presented with more and more options today and can turn to the Internet for competitive product information, change their minds more frequently and are more insistent on being satisfied. Such changes by customers put pressure on development programs to make changes accordingly.

In addition, markets shift more often and abruptly as the competitive arena becomes more turbulent and complex. For example, as globalization "flattens" the Earth, competitors appear from unexpected places, and they often bring with them new, disruptive business models. For example, Huawei appeared from nowhere in China to become a major threat to telecommunications equipment giants such as Cisco and Alcatel-Lucent, and Haier likewise has given Whirlpool a rough ride (*1*). Another market shift is the one in consumer goods regarding the relative power between manufacturers (Procter & Gamble and Rubbermaid, for example) and retailers (such as Walmart and Home Depot) (*2*). Such market shifts raise the likelihood of changes midstream in a development project.

Finally, technology—both the technology that goes into the product and the technology (like computer-aided design tools) used to develop it—is changing at an accelerating pace. New technologies appear and existing ones become obsolete or simply passé. Sometimes a new technology provides unexpected benefits that one would like to exploit during a project, such as the enthusiastic reception of portable music players by runners and others exercising physically, which, in turn, demands unexpected changes during product development to incorporate resistance to rain, perspiration and vibration. Alternatively, sometimes the benefits touted by the purveyors of the new technology don't pan out. This opens more opportunities for change in the midst of development.

On the other hand, many managers, at all levels, do not welcome change during a project. For them, mid-project changes open the door to product cost and development budget overruns, schedule slippage and product defects. Hard-pressed to deliver profit on a quarterly basis, managers, especially at higher levels, rightly see change as disruptive. Consequently, management has built development systems aimed at predictability and certain

Preston Smith is a Portland, Oregon-based consultant and trainer helping manufacturers to make their development processes more responsive. He is co-author with Donald Reinertsen of Developing Products in Half the Time *(Wiley, 1998), and this is his fourth article in* Research-Technology Management. *His latest book,* Flexible Product Development, *was published by Jossey-Bass in 2007. Smith has been an independent management consultant for 20 years and spent the preceding 20 years in engineering and management positions at IBM, Bell Laboratories, General Motors, and smaller companies. He holds a Ph.D. in engineering from Stanford University.*
preston@NewProductDynamics.com

success, such as: phased development (including Stage-Gate®), Six Sigma and Project Office. Although such systems clearly have benefits, their gains in predictability come with a corresponding side effect of rigidity.

In summary, although change during development is increasingly common, I instead see managements adopting systems that are increasingly resistant to change. This article shows how to introduce the flexibility needed to make changes during product development with minimal disruption, which I believe will separate the future winners from the losers.

Consider this example of turbulence encountered by Quadrus, a Calgary-based software development company, in developing an application for a Canadian online drugstore. This is a volatile market driven by ongoing supplier, political, regulatory, and legal thrusts. Extreme change was the essence of the management challenge Quadrus faced. In addition, its client was coming from behind in a bid to become a market leader. Quadrus responded by using very short (two-day) development iterations—each producing working software—and weekly online deployments, which not only kept up with the changing environment but aggressively led the change. By having a positive attitude toward change and employing systems that could reorient quickly, Quadrus' client could respond to competitive challenges and regulatory demands faster than competitors, thus leading the change to gain competitive advantage (*3*, p. 249).

A Model for Flexibility

In this article, flexibility refers to the ability to make changes in the product being developed or in the process by which it is developed, even relatively late in development, without being too disruptive. Such flexibility is rare today because managements have opted for systems that actually restrict flexibility in favor of predictability (as mentioned earlier), but there is an important exception: software development. Over the past several years, software developers, such as those at Quadrus, have felt the need to accommodate change during development and have developed systems that reintroduce flexibility. These fall under the label of agile software development, and they stem from the Agile Manifesto (*AgileManifesto.org*). Although there are several variations in methodology, all agile methods employ some rather revolutionary approaches:

- They all develop software iteratively in loops of typically two weeks but never more than six weeks.
- They all deliver working software at the end of each of these iterations (in contrast to the more common deliverable of documentation).
- They all reassess and replan the product requirements at the end of each iteration.

Software developers have produced systems that reintroduce flexibility.

- They incorporate the customer in this frequent iterative planning.
- They depend on small, close-knit teams and will subdivide a large project until they can use such teams.
- Many employ pairing, in which developers write all production code using two programmers sitting at one computer with one keyboard and mouse, trading off between "driver" and "navigator" roles.
- They integrate one product feature at a time into the existing package and automate testing so that they can test continually as they integrate in order to detect problems early.
- Furthermore, many write these tests before developing the corresponding product feature and then design the feature to pass the test.

This is not business as usual in the software development world, in which the norm is a sequential (waterfall) process with extensive upfront documentation and many design reviews (code walkthroughs) to ensure that the software works properly when it is finally operational.

Although agile development has grown explosively in the software community, it depends on some unique characteristics of the software medium—such as object technologies and the ability to automate all testing—that are not available to the developers of other types of products. This does not mean that other fields cannot be agile, but it does mean that other developers and managers wishing to become more agile will have to rethink the basics of agility and find other tools and approaches for restoring flexibility to non-software development.

This rethinking of agile development is not straightforward, nor can one simply map the agile development characteristics in the bullet list above into flexibility techniques for non-software development projects. It requires a rebuilding, for instance, in recognizing that:

- There is value in making the product modular so that change can be contained within a module (as is done with object technologies in software).

• A key to flexibility is delaying decisions, (as agile software developers do by deferring decisions on a product feature until the iteration in which the feature is to be implemented).

• Small, close-knit teams do best at managing the heavy, highly responsive communication needs of a project subject to unrelenting change.

• There is great value in building and maintaining options to be available in case something changes.

From such recognitions, which have come largely from observing how agile software development projects work, I have assembled the set of flexible development tools and approaches described below for application to non-software products. Although aimed beyond software, I believe these techniques will also help software developers to better appreciate the essentials of what they are doing.

Although each of the following tools and approaches provides greater flexibility, each also has its costs, monetary or otherwise. Consequently, you should apply these tools with an eye toward both benefit and cost. Apply them selectively to only the parts of projects where you anticipate change or to only projects facing the prospect of great change. This assumes that you can, to some extent, anticipate where change is most likely to occur.

Conversely, if you can plan a project completely and do not expect it to change, these tools are unwise. If this is your situation, however—because change and innovation are inseparable—you might question whether you are innovating adequately, which many CEOs put high in their priorities (*4*).

Continually Monitor Customers

As noted, customers (or users) are a major source of change. Thus, in order to manage in an environment of change, you must find ways of staying abreast of changes in the customer's environment and in his/her perception of the product you are developing; it also helps to find ways of specifying product requirements that are less susceptible to change.

In 20 years of product development consulting, I have found that companies that are good at understanding their customers find ways unique to their business for their developers to keep in regular touch with the customer experience. For instance, Black & Decker sends design engineers out with customer support technicians as they make their rounds to construction sites and home centers (*5*). Toyota has its Japanese engineers cruise American freeways, rest stops, shopping malls, and even places as unusual as Disneyland to see first-hand how its American customers use the product (*6*, p. 30). Surgical instrument manufacturers put their designers into operating rooms.

This may simply seem like good business practice, but it becomes essential for flexible development because it gives developers a sense of what is going on in the user space so that they can anticipate a change, or at least recognize it when it happens. In short, it makes developers lighter on their feet because "they have been there before."

To anticipate change, you can take this one step further than ordinary users by connecting your developers with *lead* users, that is, those who are using your products—and maybe even modifying them to suit—in advanced ways that the general user might need tomorrow. Eric von Hippel describes this technique (*7*) and cites a project he worked on with 3M to find new infection-control products (*8*). The traditional users here were surgeons working in advanced countries, but the lead users he found—who were forced to look at infection in dramatically fresh ways—were veterinary surgeons, Hollywood makeup artists, and surgeons working under challenging conditions in developing countries.

A challenge related to understanding the changing needs of customers is specifying product requirements in an environment of change. So-called best practice tells us that requirements should be specified carefully at a project's outset and "frozen" thereafter, but Don Reinertsen has found, by surveying a broad base of developers over several years, that this never happens in practice and that those who wait for complete specifications will probably be beaten to market by those willing to start with incomplete ones (*3*, p. 32). In other words, an environment of evolving, changing requirements is far more realistic for all products than the imagined one of frozen requirements. Furthermore, Alan MacCormack and Barry Boehm, from their research on software development projects, show us that, even if we could specify the product at the outset, this may be unwise, because, in a changing environment, the ability to make mid-course changes in requirements in response to customer feedback yields better products (*3*, pp. 34–35).

Accordingly, there are several ways of specifying a product at a higher level that is less susceptible to change, such as by specifying how the user will relate to it rather than by specifying features directly. These higher-level techniques include product visions, personas, use cases, and user stories (*3*, pp. 41–47).

Fence-in Change

Developers often suspect that certain parts of the product will change more than others. If so, they can divide the product into modules to isolate design change. Then, if change occurs, its effect on the whole design is limited; design changes will not ripple into areas that need not change.

The idea of modular design is to create strong barriers (interfaces) between modules. In application, this means that you are building fences to contain areas subject to change. Usually, you should draw the fence as tightly as possible around the area of suspected change to minimize the surrounding disturbance.

Black & Decker used this technique to manage change in a cordless screwdriver project. It proceeded with the highly engineered front end (motor, gearbox and chuck), which was unlikely to change and kept the handle as a separate module because its market research on handle shape was inconclusive. The company actually changed direction on handle shape six weeks into production (extremely late), which was enabled by the carefully chosen modular architecture (*3,* pp. 66–67).

However, Toyota (and others) uses this principle effectively in reverse: it fences in the areas it does not want to change. For example, the beams in car doors are part of its crashworthiness capability, which can be developed only with considerable engineering and lots of expensive, time-consuming testing. Consequently, Toyota fences in stability in the door beams as a module but allows designers great freedom to change any of the surrounding sheet metal as car styles change periodically (*6,* pp. 43, 245). Toyota does the same with the transmission, a complex, highly engineered unit whose reliability is critical; interfaces around the transmission allow it to remain constant while everything connected to it changes (*9*).

Although modular architectures have great power to accommodate change, they also have shortcomings relative to integral architectures. One negative of modularity is cost—interfaces usually add cost to the product. Another is a product performance burden—interfaces generally reduce product performance by adding weight and consuming space, or by introducing the possibility of weak or leaky joints in mechanical systems or crosstalk or phase shift in electrical ones. More fundamentally, interfaces introduce constraints on the design that limit designers' ability to optimize the system completely. Consequently, one should apply modular architectures selectively where they will contribute the most to flexibility without incurring undue penalties.

Try Things Out

If you believe the project will not change, you have the luxury of planning it completely and simply following your plan. To the extent that change might occur, you are wise to hedge your bets. Experimentation is an excellent tool for this, that is, trying things out intentionally to see what might happen. Such experimentation allows you to test alternatives, to broaden the design space in case change occurs, and to see how robust your design is against change.

Ability to make mid-course changes in requirements in response to customer feedback yields better products.

Experimentation takes many forms. It includes: building prototypes, mock-ups and breadboards; testing these; running simulations and building models; and overloading a system to see what fails first (a smoke test). The savvy experimenter looks for experiments that will return as much information and insight as possible for the investment in money and time. This cost–benefit equation has shifted enormously in recent years as computer-aided technologies have greatly reduced the cost of experimentation in many fields, such as exploring molecules in pharmaceutical development, building physical models of mechanical parts for customers to touch, and automating the testing of software and hardware.

Such computerized technologies permit experimenting prolifically at reasonable cost. Many managers employ these computerized tools to cut cost and simply pocket the savings, but Orion, a Massachusetts sensor technology firm, used computerized prototyping in a hand-held surgical laser project to explore seven times more design options than it would normally have while keeping its prototyping budget to only two-thirds of the previous amount (*3,* pp. 98–100). This gave Orion much more flexibility to find a comfortable, easy-to-use design.

Again, experiments cost money and consume time, so seasoned experimenters seek areas where change is likely and concentrate their experimentation there. Other trade-offs are involved. One is in deciding whether to run several experiments, in parallel (faster) or sequentially (usually cheaper). There are guidelines for making such choices, such as the amount of learning you can apply from one generation of experiments to the next and how cleanly structured you expect the design space to be (*3,* pp. 102–104).

Explore the Design Space

Experimentation is a good tool for exploring options, but we also need a strategy for applying it, that is, for knowing which experiments to run. Toyota has an excellent strategy, called set-based design, that amounts to a very different way of approaching design.

To illustrate the difference, I contrast the set-based approach with the more normal point-based one by using a non-product-development example attributable to Ward *et al.* (*10*). Suppose that you wish to convene a meeting. The traditional way of doing this (point-based) would be for the convener to contact a participant and negotiate a mutually acceptable time. Then the convener proceeds to the second participant, doing the same and perhaps returning to the first participant to renegotiate. This repeats with the other participants.

The set-based approach would be for the convener to request all participants' calendars first. Then the convener (today using modern meeting-scheduling software), looks for a common open time (the intersection of individuals' available times) and sets a time where everyone is free. This not only saves time, but more important, it exposes all possible solutions so that, should something change, the convener is in a strong position to make adjustments easily.

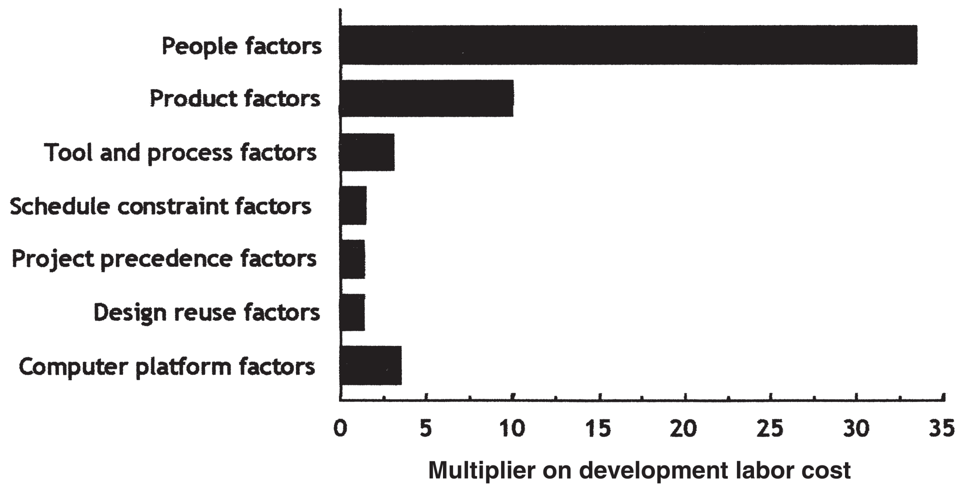
As you can see from this example, set-based design is a subtractive process; it employs logical intersections. You define the initial feasible space and proceed to impose constraints, for instance constraints on the design due to manufacturability, cost, weight, or physics. Each constraint reduces the feasible design space methodically. You thus maintain a *space* of feasible designs so that you can turn to another point in the space if a change renders the current design point unsatisfactory. In short, it is a continual pruning process.

This subtractive process and the design space it maintains are quite different from the normal design process that proceeds with one (or at most a few discrete) design that the designer hones to a final point. With such point-based design, if something changes, the designer has no information on adjacent designs and must usually retrace many steps. The set-based designer can merely shift to another place in the current design space.

Build Strong Teams

You have probably read much about strong development teams and have made improvements in your teams accordingly. Nevertheless, there is much we can learn from software development about the value of strong teams and how we might build them. For a starter, the initial statement of the Agile Manifesto is "Individuals and interactions over processes and tools . . . . That is, while there is value in the items on the right, we value the items on the left more."

In building a model to estimate the time and effort needed to develop a piece of software, software methodology researcher Barry Boehm identified 22 multipliers on project labor cost and collected data for these multipliers over a broad range of development projects (*11*). I have grouped these multipliers into categories in the illustration below. Thus, the labor cost multipliers asso-



Multipliers on project labor cost span the indicated ranges for each category of project factors. Although the data come from a broad variety of software development projects, all but the last one also apply to non-software projects. (Computer platform factors relate to specific characteristics of the host computer, which do not have a clear analog for non-software products.) From Flexible Product Development *by Preston G. Smith, Jossey-Bass. © 2007 by John Wiley & Sons. Used with permission.*

ciated with the people assigned to the project combine to produce a possible range of 33 to 1 in project labor cost, the multipliers associated with the product being developed span a range of 10 to 1, and so forth.

This illustration suggests strongly that the factors associated with people far outweigh the others, so devoting effort toward improving how the team works is likely to pay far bigger dividends than investments anywhere else. This stands in contrast to the great attention and investment that firms usually devote to processes, methodologies and tools.

New data from Don Reinertsen, Gary Olson, Judith Olson, and the agile software development community show that co-location is a powerful factor in improving team performance (*3,* pp. 142–146). This is disturbing news in an era in which globalization and "virtualization" are pushing teams away from co-location. However, there is much you can do to obtain many of the benefits of co-location even if your team is not completely co-located. These include co-locating all members in each metropolitan area and establishing norms for using tools like e-mail—for instance, a rule that any team e-mail must be answered within four hours.

Strong teams are vital to flexibility because a turbulent environment presents much bigger communication and coordination challenges than a stable one, and high-performance teams are the prime means of coping with these challenges.

Make Decisions at the Last Responsible Moment

If you dissect a product development project to see what occurs inside, you will find that the core activity is decision making—thousands of little decisions that cumulatively create the product. It follows that you should concentrate on this "inner loop" of the process if you wish to improve product development. For instance, if you want to speed it up, find ways, such as co-location, to accelerate decision making, as programmers do when they find the inner loop of their code and rewrite it in a low-level language that runs faster.

Alternatively, if you wish to be more flexible, find ways to make decisions more flexibly. A major opportunity here is not to make a decision until you *must* make it—what we call making decisions at the *last responsible moment.* This might seem like procrastination, which is basically being lazy about a decision, but it is actually a proactive process of identifying when the decision must be made and scheduling it, then proceeding to collect information to help make a better decision when its last responsible moment arrives.

Making decisions at the last responsible moment has two advantages. First—most importantly for flexibility—it keeps your options open longer, and second, it allows you to make the decision using fresher information.

Improving how the team works is likely to pay far bigger dividends than investments anywhere else.

However, there are some decisions, such as ones where the outcome is not likely to change or where the outcomes are nearly equal, that you should make early so you can dismiss them.

Although this delayed decision-making may seem obvious, it is not the way management normally operates. Managers usually are paid to make decisions, not to put them off. In contrast, managers at Toyota are paid to delay decisions (*10*).

Note that popular phased development processes tend to force organizations into making many decisions unnecessarily at the project's outset in order to "nail things down." Unfortunately, these nailed-down items constitute a loss of flexibility.

Plan Piecemeal and Constantly Consider Risk

Project management has become quite popular over the past decade. However, the project management profession has its roots in the construction industry where predictability is valued highly and major change is relatively uncommon. Consequently, you must handle project management quite differently when change is the norm. I mention just two areas to refocus:

Project planning presents a dilemma for projects undergoing heavy change. The temptation is to replan the project in response to change, but this can lead to paying constant attention to replanning rather than to developing the product itself. When turbulence is high, you can shift to two other means of planning. One is rolling-wave planning, in which you plan the next segment in detail and leave the rest of the project planned only at the top level. As you progress, the wave rolls forward and the next segment undergoes detail planning. This way, you are not investing much in long-term plans that are likely to change anyway.

The other is loose–tight planning in which you alternate periods of tight planning and control with more relaxed periods in which to regroup. This is what the iterative approach of agile software development does by delaying planning of features to be implemented until the beginning of an iteration and then planning only those features to be implemented in the next iteration. During an iteration, planning is tight, but between iterations all remaining work is reassessed. This is also how Boeing developed its 777 airliner by alternating periods of design with periods of stabilization (*12*).

Another part of project management you must handle differently under heavy change is risk management. Good practice under light change is to *integrate* the risk management process with the rest of the project management process so that risk management actions are identified, planned and executed methodically as part of the project plan. When change dominates the project, risk management instead must be *intrinsic*, that is, everything you do to manage the project is done to manage its risk: you keep close to customers to manage the risk of customer change, you fence in modules to manage the risk of pervasive design modification, and so forth.

Maintain Flexibility in Upper Layers of Process

Although the illustration suggests that the development process is not the place to devote improvement effort, there is nevertheless much you can do to make your development process more flexible. First, recognize that because a project has many dimensions, you may find that, in one project, you need flexibility in certain areas while you will need strict control and accountability in certain other areas. For instance, a project may have disastrous consequences if the product has defects but its essence may be to extend a new technology through market adaptation. Consequently, in this project you will need tight control over quality while your technology experimentation program should be open and extensive to provide flexibility. In the next project, these areas of control and openness may shift completely. Boehm and Turner show how to combine flexibility with tight control based on a project's specific needs (*13*).

Software developers have learned a useful lesson about building flexible development processes: standardize in the lower layers of process (*13,* p. 152). For example, standardize how you run a test, how you assess its results, and how you apply tolerances to a certain type of component. Then maintain flexibility by leaving freedom in the upper layers of your process, that is, in how you assemble the basic activities. We do this because the upper layers provide the flexibility while the lower layers provide the quality control.

Consider language as an analogous situation. We do not change the letters; there are exactly 26 in English. Words are mostly standardized, but new ones do arise or are borrowed from other languages: blog, biosphere and *ciao*. There are rules for constructing sentences, but many variations are acceptable too. Looser still are paragraphs, and almost anything goes in the document layer.

Don't make a decision until you must—the last responsible moment.

Observe that most product development processes attempt to specify the upper layers at project outset, for instance, by listing which activities must occur in which phases, which unnecessarily constrains flexibility. Instead, for flexibility, leave many of these upper-layer decisions to be decided at the last responsible moment.

Out-Innovate the Competition

Change fits perfectly with innovation, so why make it hard to change? Instead, use these tools and approaches as well as many others available to you (*3*) to improve your ability to change, to the point that you can out-change and thus out-innovate your competitors. ⦿

References

1. Zeng, Ming and Williamson, Peter J. 2007. *Dragons at Your Door*. Boston: Harvard Business School Press.
2. Mitchell, Alan. 2004. Why Retailers' Power Has Reached the Tipping Point. *Marketing Week*, August 5, pp. 32–33.
3. Smith, Preston G. 2007. *Flexible Product Development*. San Francisco: Jossey-Bass.
4. Boston Consulting Group. 2006. *Innovation 2006*. Boston: Boston Consulting Group, Inc.
5. Smith, Preston G., and Reinertsen, Donald G. 1998. *Developing Products in Half the Time* (Second Edition). New York: Wiley, pp. 95–96.
6. Morgan, James M., and Liker, Jeffrey K. 2006. *The Toyota Product Development System*. New York: Productivity Press.
7. von Hippel, Eric. 1988. *The Sources of Innovation*. New York: Oxford University Press.
8. von Hippel, Eric, Thomke, Stefan and Sonnack, Mary. 1999. Creating Breakthroughs at 3M. *Harvard Business Review*, September–October, pp. 47–57.
9. Sobek, II, Durward K., Ward, Allen C. and Liker, Jeffrey K. 1999. Toyota's Principles of Set-Based Concurrent Engineering. *Sloan Management Review*, Winter, pp. 67–83.
10. Ward, Allen, Liker, Jeffrey K., Cristiano, John J., and Sobek, Durward K., II. 1995. The Second Toyota Paradox: How Delaying Decisions Can Make Better Cars Faster. *Sloan Management Review*, Spring, pp. 43–61.
11. Boehm, Barry W., et al. 2000. *Software Cost Estimation with COCOMO II*. Upper Saddle River, New Jersey: Prentice Hall.
12. Thomke, Stefan H. 2003. *Experimentation Matters*. Boston: Harvard Business School Press, pp. 168–169.
13. Boehm, Barry, and Turner, Richard. 2004. *Balancing Agility and Discipline*. Boston: Addison-Wesley.