How Microsoft Builds Software

Michael A. Cusumano and Richard W. Selby

Teams of programmers and testers frequently synchronize and periodically stabilize the changes they make to products in progress, yielding Excel, Office, Publisher, Windows 95, Windows NT, Word, Works, and more.

Since the mid-1980s, Microsoft and other PC software companies have gradually reorganized the way they build software products in response to quality problems and delayed deliveries [10]. Many have also found it necessary to organize larger teams in order to build up-to-date PC software products that now consist of hundreds of thousands or even millions of lines of source code and require hundreds of people to build and test over periods of one or more

years. As the world's largest producer of PC software, with approximately 20,500 employees, 250 products, and annual revenues of $8.7 billion (fiscal year ending June 1996), Microsoft has probably tackled more PC software projects than any other company in the industry. The complexity of some of its products, such as Windows 95 (which contains more than 11 million lines of code and required a development team of more than 200 programmers and testers), rivals that of many systems for mainframe computers and telecommunication systems.

Microsoft's philosophy for product development has been to cultivate its roots as a highly flexible, entrepreneurial company and not to adopt too many of the structured software-engineering practices commonly promoted by such organizations as the Software Engineering Institute and the International Standards Organization [6]. Rather, Microsoft has tried to "scale up" a loosely structured small-team (some might say hacker) style of product development. The objective is to get many small parallel teams (three to eight developers each) or individual programmers to work together as a single relatively large team in order to build large products relatively quickly while still allowing individual programmers and teams freedom to evolve their designs and operate nearly autonomously. These small parallel teams evolve features and whole products incrementally while occasionally introducing new concepts and technologies. However, because developers are free to innovate as they go along, they must synchronize their changes frequently so product components all work together.

We will summarize how Microsoft uses various techniques and melds them into an overall approach that balances flexibility and structure in software product development. We are not suggesting that the Microsoft-style development approach is appropriate for all types of software development or that Microsoft "invented" these development ideas. Nor do we suggest Microsoft's software development methods by themselves have caused the company's great financial success. We are saying there are several lessons to be learned from how Microsoft builds software products, some of which apply to other organizations, some of which do not. Software developers and managers from other organizations can decide which ideas may apply to them after considering such factors as their company's goals, marketing strategies, resource constraints, software reliability requirements, and development culture.[1]

Frequent Synchronizations and Periodic Stabilizations

We label Microsoft's style of product development the "synch-and-stabilize" approach. The essence is simple: Continually synchronize what people are doing as individuals and as members of parallel teams, and periodically stabilize the product in increments as a project proceeds, rather than once at the end of a project. Microsoft people refer to their techniques variously as the "milestone," "daily build," "nightly build," or "zero-defect" process. (The term *build* refers to the act of putting together partially completed or finished pieces of a software product during the development process to see what functions work and what problems exist, usually by completely recompiling the source code and executing automated regression tests.) Whatever the label, these techniques address a problem common to many firms in highly competitive, rapidly changing industries: Two or three people can no longer build many of the new, highly complex products; such products require much larger teams that must invent and innovate as they develop the product. Team members need to create components that are interdependent, but such components are difficult to define accurately in the early stages of the development cycle. In these situations, projects must pro-

> Without its synch-and-stabilize structured approach, Microsoft would probably never have been able to design, build, and ship the products it offers now and plans to offer in the future.

[1]This article is based on the authors' *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*, The Free Press/Simon & Schuster, New York, 1995.

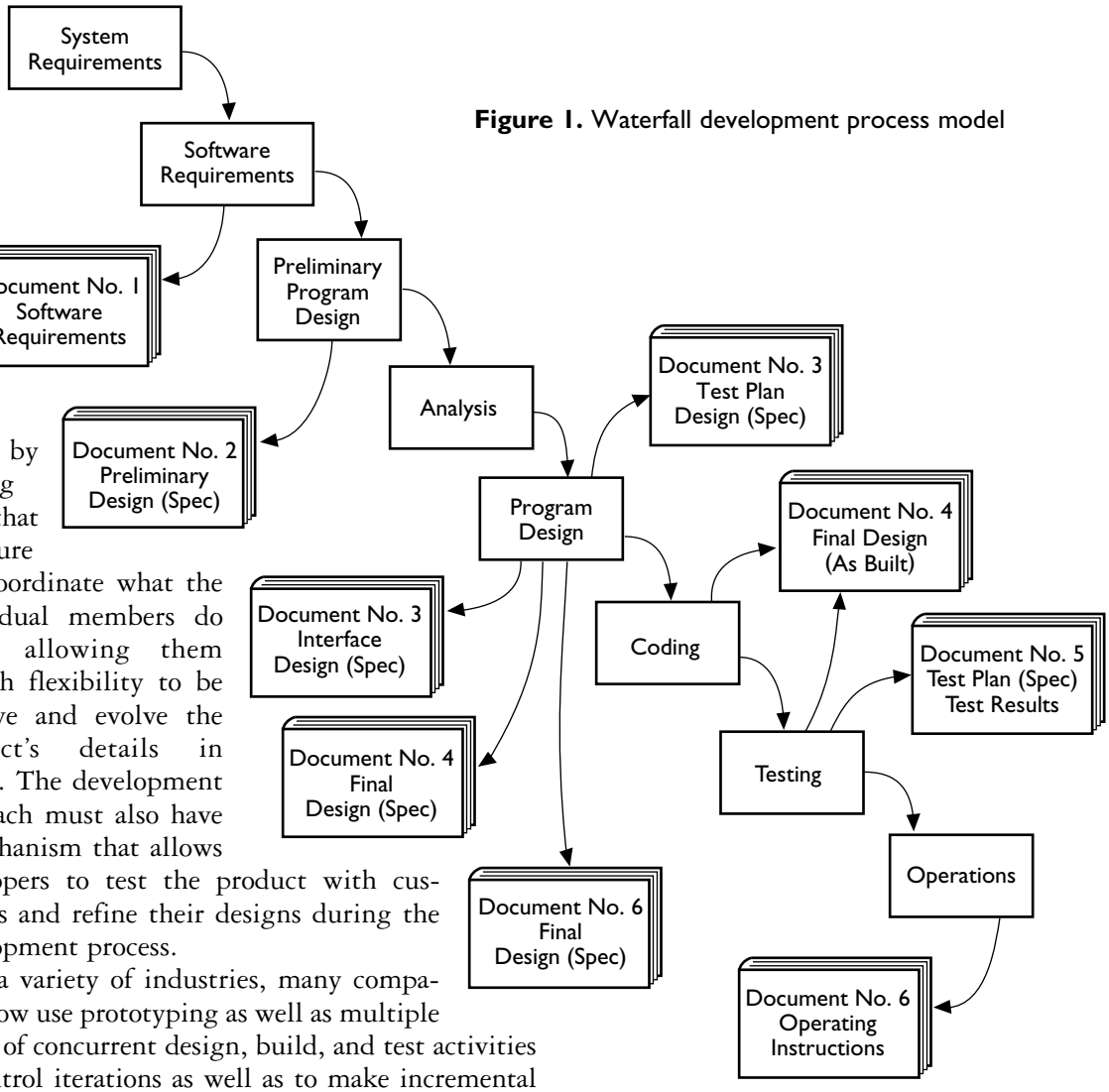

Figure 1. Waterfall development process model

ceed by finding ways that structure and coordinate what the individual members do while allowing them enough flexibility to be creative and evolve the product's details in stages. The development approach must also have a mechanism that allows developers to test the product with customers and refine their designs during the development process.

In a variety of industries, many companies now use prototyping as well as multiple cycles of concurrent design, build, and test activities to control iterations as well as to make incremental changes in product development [12]. In the software community, researchers and managers have talked about "iterative enhancement," a "spiral model" for iteration among the phases in project development, and "concurrent development" of multiple phases and activities for the past 20 years [1, 2, 3]. However, many companies have been slow to formally adopt these recommendations. Nonetheless, the basic idea shared by these approaches is that users' needs for many types of software are so difficult to understand and that changes in hardware and software technologies are so continuous and rapid, it is unwise to attempt to design a software system completely in advance. Instead, projects may need to iterate while concurrently managing many design, build, and testing cycles to move forward toward completing a product.

This iterative as well as incremental and concurrent-engineering style contrasts with a more sequential, or "waterfall," approach to product development. In the waterfall approach, projects seek to "freeze" a product specification, create a design, build components, and then merge the components—primarily at the end of the project in one large integration and testing phase (see Figure 1). This approach to software development was common in the 1970s and 1980s [8]. It also remains a basic model for project planning in many industries [11]. The waterfall model has gradually lost favor, however, because companies usually build better products if they can change specifications and designs, get feedback from customers, and continually test components as the products are evolving. As a result, a growing number of companies in software and other industries—including Microsoft—now follow a process that iterates among design, building components, and testing, and also overlaps these phases and contains more interactions with customers during development. Many companies also ship preliminary versions of their products, incrementally adding features or functionality over time in various product releases. In addition, many companies frequently integrate pieces of their

products together (usually not daily, but often on a biweekly or monthly basis). Frequent integrations help determine what does and does not work without waiting until the end of the project—which may be several years away.

Strategies and Principles

We observed Microsoft over a two-and-a-half year period ending mid-1995, conducted in-depth interviews with 38 key people (including chairman and CEO Bill Gates), and reviewed thousands of pages of confidential project documentation and postmortem reports. Through this research, we identified two strategies for defining products as well as development processes and sets of principles that seem critical to making the synch-and-stabilize style of product development.

Microsoft teams begin the process of product development by creating a "vision statement" defining the goals for a new product and orders the user activities that need to be supported by the product features (see Figure 2). Product managers (marketing specialists) take charge of this task while consulting program managers who specialize in writing up functional specifications of the product. The program managers, in consultation with developers, then write a functional specification outlining the product features in sufficient depth to organize schedules and staffing allocations. But the initial specification document does not try to cover all the details of each feature or lock the project into the original set of features. During product development, the team members revise the feature set and feature details as they learn more about what should be in the product. Experience at Microsoft suggests that the feature set in a specification document may change by 30% or more.

The project managers then divide the product and the project into parts (features and small feature teams) and divide the project schedule into three or four milestone junctures (sequential subprojects) representing completion points for major portions of the product (see Figure 3). All the feature teams go through a complete cycle of development, feature integration, testing, and fixing problems in each milestone subproject. Moreover, throughout an entire project, the feature teams synchronize their work by building the product and by finding and fixing errors on a daily and weekly basis. At the end of a milestone subproject, the developers fix almost all the errors detected in the evolving product. These error corrections stabilize the product and enable the team to have a clear understanding of which portions of the product have been completed. The development team may then proceed to the next milestone and, eventually, to the ship date.

Planning Phase Define product vision, specification, and schedule

- **Vision Statement** Product and program management use extensive customer input to identify and priority-order product features.
- **Specification Document** Based on vision statement, program management and development group define feature functionality, architectural issues, and component interdependencies.
- **Schedule and Feature Team Formation** Based on specification document, program management coordinates schedule and arranges feature teams that each contain approximately 1 program manager, 3–8 developers, and 3–8 testers (who work in parallel 1:1 with developers).

Development Phase Feature development in 3 or 4 sequential subprojects that each results in a milestone release

Program managers coordinate evolution of specification. Developers design, code, and debug. Testers pair with developers for continuous testing.

- **Subproject I** First 1/3 of features (Most critical features and shared components)
- **Subproject II** Second 1/3 of features
- **Subproject III** Final 1/3 of features (Least critical features)

Stabilization Phase Comprehensive internal and external testing, final product stabilization, and ship

Program managers coordinate OEMs and ISVs and monitor customer feedback. Developers perform final debugging and code stabilization. Testers recreate and isolate errors.

- **Internal Testing** Thorough testing of complete product within the company
- **External Testing** Thotough testing of complete product outside the company by "beta" sites, such as OEMs, ISVs, and end users
- **Release preparation** Prepare final release of "golden master" disks and documentation for manufacturing

Figure 2. Overview of the synch-and-stabilize development approach

Defining Products and Development Processes

To define products and organize the development process, leading Microsoft product groups follow a

strategy we describe as "focus creativity by evolving features and 'fixing' resources." Teams implement this strategy through five specific principles:

- Divide large projects into multiple milestone cycles with buffer time (20%–50% of total project time) and no separate product maintenance group.
- Use a "vision statement" and outline feature specifications to guide projects.
- Base feature selection and priority order on user activities and data.
- Evolve a modular and horizontal design architecture, mirroring the product structure in the project structure.
- Control by individual commitments to small tasks and "fixed" project resources.

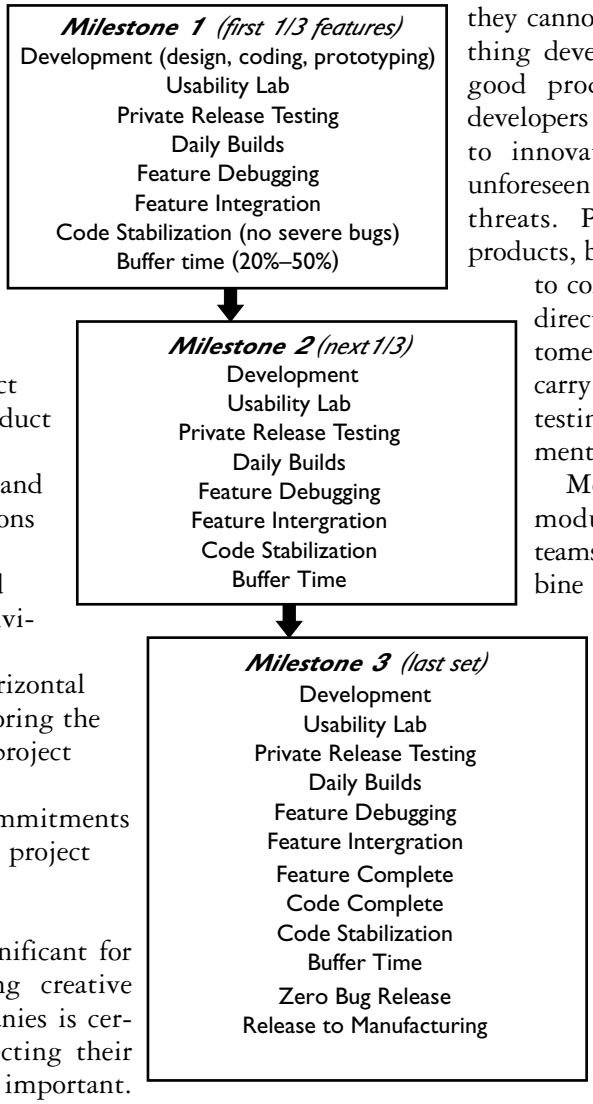

Figure 3. Milestones in the synch-and-stabilize approach (each taking two to four months)

These principles are significant for several reasons. Employing creative people in high-tech companies is certainly important, but directing their creativity is often more important. Managers can do this by getting development personnel to think about the features large numbers of customers will pay money for and by pressuring projects by limiting the resources, such as staffing and schedule, the company will invest in their development. Otherwise, software developers risk never shipping anything to market. This risk is especially troublesome in fast-moving industries in which individuals or teams have unfocused or highly volatile user requirements, frequently change interdependent components during a project, or fail to synchronize their work.

Microsoft gets around these problems by structuring projects into sequential subprojects containing priority-ordered features; buffer time within each subproject gives people time to respond to changes and to unexpected difficulties or delays. Microsoft projects use vision statements and outline specifications rather than detailed designs and complete product specifications before coding, because teams realize they cannot determine in advance everything developers need to do to build a good product. This approach leaves developers and program managers room to innovate or adapt to changed or unforeseen competitive opportunities and threats. Particularly for applications products, because development teams try to come up with features that map directly to activities typical customers perform, the teams need to carry out continual observation and testing with users during development.

Most product designs have modular architectures allowing teams to incrementally add or combine features in a straightforward, predictable manner. In addition, managers allow team members to set their own schedules, but only after the developers have analyzed tasks in detail (for example, half-day to three-day chunks) and have been asked to personally commit to the schedules they set. Managers then "fix" project resources by limiting the number of people they allocate to each project. They also try to limit the time spent on projects, especially for applications like Office and multimedia products, so teams can delete features if they fall too far behind. (However, cutting features to save schedule time is not always possible with operating systems projects in which reliability is more important than features and in which many features are closely coupled and cannot easily be deleted individually.)

Developing and Shipping Products

To manage the process of developing and shipping products, Microsoft follows another strategy we describe as "do everything in parallel with frequent synchronization." Teams implement this strategy by following another set of five principles:

- Work in parallel teams but "synch up" and debug daily.
- Always have a product you can ship, with versions for every major platform and market.

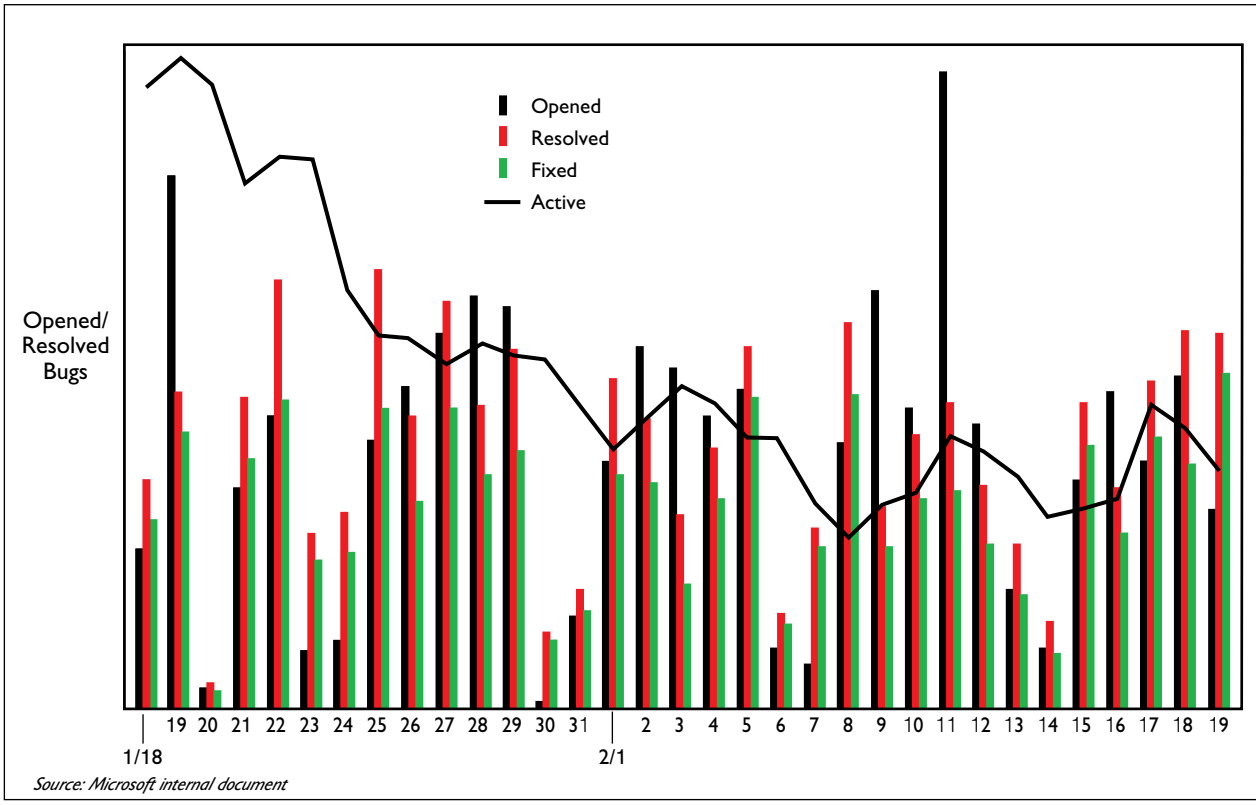

Figure 4. Milestone 2: Bug data and daily builds from Excel/Graph

- Speak a "common language" on a single development site.
- Continuously test the product as you build it.
- Use metric data to determine milestone completion and product release.

These principles bring considerable discipline to the development process without the need to control every moment of every developer's day. For example, managers in many different companies talk about making their companies less bureaucratic, more innovative, and faster to react through organization and process "re-engineering" and "restructuring" to speed product development. But complex products often require large teams of hundreds of people, not small teams of a dozen or fewer engineers; and large teams can make communication and coordination extremely difficult and slow. Large-scale projects are simpler to schedule and manage if they proceed with clearly defined functional groups, sequential phases, and precise rules and controls. This approach, however, may restrain innovation and undervalue the importance of frequently synchronizing work. Communication and coordination difficulties across the functions and phases may also result in a project's taking more time and people to complete than projects that overlap tasks and require that people share responsibilities and work in small, nimble teams.

What Microsoft tries to do is allow many small teams and individuals enough freedom to work in parallel yet still function as a single large team, so they can build large-scale products relatively quickly and cheaply. The teams also adhere to a few rigid rules that enforce a high degree of coordination and communication.

For example, one of the few rules developers must follow is that, on whatever day they decide to "check in," or enter their pieces of code into the product database, they must do so by a particular time, say, 2 p.m. or 5 p.m. This rule allows the project team to assemble available components, completely recompile the product source code, and create a new "build" of the evolving product by the end of the day or by the next morning and then start testing and debugging immediately. (This rule is analogous to telling children that they can do whatever they want all day, but they must be in bed at 9 p.m.) Another rule is that if developers check in code that "breaks" the build by preventing it from completing the recompilation, they must fix the defect immediately. (This rule resembles Toyota's famous production system, in which factory workers are encouraged to stop the manufacturing lines whenever they notice a defect in a car they are assembling [4].)

Microsoft's daily build process has several steps. First, in order to develop a feature for a product, developers can check out private copies of source code files from a centralized master version of the source code. They implement their features by making changes to their private copies of the source code files. The developers then create a private build of the product containing the new feature and test it. They then check in the changes from their private copies of the source code files to the master version of the source code. The check-in process includes an automated regression test to help ensure that their changes to the source code files do not cause errors elsewhere in the product. Developers usually check in their code back to the master copy at least twice a week but may check it in daily.

Regardless of how often individual developers check in their changes to the source code, a designated developer, called the project build master, generates a complete build of the product on a daily basis using the master version of the source code. Generating a build for a product consists of executing an automated sequence of commands called a "build script." This daily build creates a new internal release of the product and includes many steps that compile source code. The build process also automatically translates the source code for a product into one or more executable files and may create various library files, allowing end users to customize the product. The new internal release of the product built each day is the daily build. Daily builds are generated for each platform, such as Windows and Macintosh, and for each market, such as the U.S., and for the major international versions.

Product teams also test features as they build them from multiple perspectives, including bringing in customers "off the street" to try prototypes in a Microsoft usability lab. In addition, nearly all Microsoft teams work at a single physical site with common development languages (primarily C and C++), common coding styles, and standardized development tools. A common site and common language and tools help teams communicate, debate design ideas, and resolve problems face to face. Project teams also use a small set of quantitative metrics to guide decisions, such as when to move forward in a project and when to ship a product to market. For example, managers rigorously track progress of the daily builds by monitoring how many bugs are newly opened, resolved (such as by eliminating duplicates or deferring fixes), fixed, and active (see Figure 4).

The Hacker Approach

Some people may argue that Microsoft's key practices in product development—daily synchronization through product builds, periodic milestone stabilizations, and continual testing—are no more than process and technical fixes for a hacker software organization now building huge software systems. We do not really disagree, but we also think that Microsoft has some insightful ideas on how to combine structure with flexibility in product development. It is worthwhile to note that the term hacker is not necessarily a bad word in the PC industry. It goes back to the early days of computer programming in the 1960s, when long-haired, unkempt technical wizards would work at their computers with no formal plans, designs, or processes, and just "bang on a keyboard" and "hack away" at coding [7]. This approach worked for relatively small computer programs that one person or several people could write, such as the earliest versions of DOS, Lotus 1-2-3, WordPerfect, Word, and Excel. It became unworkable as PC software programs grew into hundreds of thousands and then millions of lines of code.

> No PC software company has done a better job of keeping some basic elements of the hacker culture while adding just enough structure to build today's, and probably tomorrow's, PC software products.

Formal plans and processes were first used in the mainframe computer industry where software systems had grown to the million-line-plus size by the end of the 1960s [5]. PC software companies have been unwilling to completely give up their traditions and cultures. Nor would it be wise for them to do so, given the rapid pace of change in PC hardware and software technologies

and the need for continual innovation.

No company has taken advantage of the exploding demand for PC software better than Microsoft. Similarly, no PC software company has done a better job of keeping some basic elements of the hacker culture while adding just enough structure to build today's, and probably tomorrow's, PC software products. It continues to be a challenge for Microsoft to make products reliable enough for companies to buy, powerful enough so the products' features solve real-world problems, and simple enough for novice consumers to understand. To achieve these somewhat conflicting goals for a variety of markets, Microsoft still encourages some teams to experiment and make lots of changes without much up-front planning. Projects generally remain under control because teams of programmers and testers frequently synchronize and periodically stabilize their changes.

Since the late 1980s, Microsoft has used variations of the synch-and-stabilize approach to build Excel, Office, Publisher, Windows 95, Windows NT, Word, Works, and other products. However, the synch-and-stabilize process does not guarantee on-time or bug-free products. Creating new, large-scale software products on a precisely predicted schedule and with no major defects is an extremely difficult goal in the PC industry. Microsoft and other PC software companies also try to replace products quickly and usually announce overly ambitious deadlines, contributing to the appearance of being chronically late. Nonetheless, without its synch-and-stabilize structured approach, Microsoft would probably never have been able to design, build, and ship the products it offers now and plans to offer in the future.

Microsoft resembles companies from many industries that do incremental or iterative product development as well as concurrent engineering. It has also adapted software-engineering practices introduced earlier by other companies (such as various testing techniques) and reinvented the wheel on many occasions (such as concluding that accumulating historical metric data is useful for analyzing bug trends and establishing realistic project schedules [9]). Microsoft is distinctive, however, in the degree to which it has introduced a structured hacker-like approach to software product development that works reasonably well for both small- and large-scale products. Furthermore, Microsoft is a fascinating example of how culture and competitive strategy can drive product development and the innovation process. The Microsoft culture centers around fervently antibureaucratic PC programmers who do not like a lot of rules, structure, or planning. Its competitive strategy revolves around identifying mass markets quickly, introducing products that are "good enough" (rather than waiting until something is "perfect"), improving these products by incrementally evolving their features, and then selling multiple product versions and upgrades to customers around the world.

Table 1. Synch-and-stabilize vs. sequential development

Synch-and-Stablize	Sequential Development
Product development and testing done in parallel	Separate phases done in sequence
Vision statement and evolving specification	Complete "frozen" specification and detailed design before building the product
Features prioritized and built in 3 or 4 milestone subprojects	Trying to build all pieces of a product simultaneously
Frequent synchronizations (daily builds) and intermediate stabilizations (milestones)	One late and large integration and system test phase at the project's end
"Fixed" release and ship dates and multiple release cycles	Aiming for feature and product "perfection" in each project cycle
Customer feedback continuous in the development process	Feedback primarily after development as inputs for future projects
Product and process design so large teams work like small teams	Working primarily as a large group of individuals in a separate functional department

A Semblance of Order

The principles behind the synch-and-stabilize philosophy add a semblance of order to the fast-moving, often chaotic world of PC software development. There are no silver bullets here that solve major problems with a single simple solution. Rather, there are specific approaches, tools, and techniques; a few rigid rules; and highly skilled people whose

culture aligns with this approach. As we have suggested, several elements distinguish synch-and-stabilize from older, more traditional sequential and rigid styles of product development (see Table 1).

Microsoft also has weaknesses. The company now needs to pay more attention to, for example, product architectures, defect prevention mechanisms, and some more conventional engineering practices, such as more formal design and code reviews. New product areas also pose new challenges for its development methods. For example, some new areas, such as video on demand, have many tightly linked components with real-time constraints that require precise mathematical models of when video/audio/user data can be delivered reliably and on time. Many existing and new products have an extremely large or even infinite number of potential user conditions or scenarios to test based on what hardware and applications each customer is using. These new products can benefit from some incremental changes in the development process. They also require more advance planning and product architectural design than Microsoft usually does to minimize problems in development, testing, and operation.

Nonetheless, the synch-and-stabilize process described here provides several benefits for product developers:

- It breaks down large products into manageable pieces (a priority-ordered set of product features that small feature teams can create in a few months).
- It enables projects to proceed systematically even when they cannot determine a complete and stable product design at the project's beginning.
- It allows large teams to work like small teams by dividing work into pieces, proceeding in parallel but synchronizing continuously, stabilizing in increments, and continuously finding and fixing problems.
- It facilitates competition on customer input, product features, and short development times by providing a mechanism for incorporating customer inputs, setting priorities, completing the most important parts first, and changing or cutting less important features.
- It allows a product team to be very responsive to events in the marketplace by "always" having a product ready to ship, having an accurate assessment of which features are completed, and preserving process-product flexibility and opportunism throughout the development process.

These ideas and examples provide useful lessons for organizations and managers in many industries. The synch-and-stabilize approach used at Microsoft is especially suited to fast-paced markets with complex systems products, short lifecycles, and competition based around evolving product features and de facto technical standards. In particular, coordinating the work of a large team building many interdependent components that are continually changing requires a constant and high level of communication and coordination. It is difficult to ensure that such communication and coordination take place while still allowing designers, engineers, and marketing people the freedom to be creative. Achieving this balance is perhaps the central dilemma that managers of product development face—in PC software as well as in many other industries. C

References

1. Aoyama, M. Concurrent-development process model. *IEEE Software 10*, 4 (July 1993).
2. Basili, V.R., and Turner, A.J. Iterative enhancement: A practical technique for software development. *IEEE Trans. Software Eng. SEI-1*, 4 (Dec. 1975), 390–396.
3. Boehm, B.W. A spiral model of software development and enhancement. *IEEE Comput. 5* (May 1988), 61–72.
4. Cusumano, M.A. *The Japanese Automobile Industry: Technology and Management at Nissan and Toyota*. Harvard University Press, Cambridge, Mass., 1985.
5. Cusumano, M.A. *Japan's Software Factories: A Challenge to U.S. Management*. Oxford University Press, New York, 1991.
6. Humphrey, W.S. *Managing the Software Process*, Addison-Wesley, New York, 1989.
7. Levy, S. *Hackers: Heroes of the Computer Revolution*. Anchor/Doubleday, New York, 1984.
8. Royce, W.W. Managing the development of large software systems. In *Proceedings of IEEE WESCON* (Los Angeles, 1970), pp. 1–9.
9. Selby, R.W. Empirically based analysis of failures in software systems. *IEEE Trans. Reliab. 39*, 4 (Oct. 1990), 444–454.
10. Smith, S.A., and Cusumano, M.A. Beyond the Software Factory: A comparison of "classic" and "PC" software developers. Working Paper 3607-93/BPS, Sloan School, Massachusetts Institute of Technology, Cambridge, Mass., 1993).
11. Urban, G.L., and Hauser, J.R. *Design and Marketing of New Products*. Prentice-Hall, Englewood Cliffs, N.J., 1993.
12. Wheelwright, S.C., and Clark, K.B. *Revolutionizing Product Development*. Free Press, New York, 1992.

Michael A. Cusumano (cusumano@mit.edu) is a professor of strategy and technology management in the Sloan School of Management at the Massachusetts Institute of Technology, Cambridge, Mass.

Richard W. Selby (selby@ics.uci.edu) is an associate professor of computer science in the Department of Information and Computer Science at the University of California, Irvine.