SYNCHRONIZD-AND-STABILIZE: An Approach for Balancing Flexibility and Structure in Software Product Development

Michael A. Cusumano* and Richard W. Selby**
*MIT Sloan School of Management
**University of California, Irvine

Working Paper #3930-BPS-96
June 11, 1996

SYNCHRONIZE-AND-STABILIZE:

An Approach for Balancing Flexibility and Structure in Software Product Development

Michael A. Cusumano
Professor
Sloan School of Management
Massachusetts Institute of Technology

Richard W. Selby
Associate Professor
Department of Information and Computer Science
University of California, Irvine

June 11, 1996

Forthcoming, Communications of the ACM

This article is based on *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People* (New York: The Free Press/Simon & Schuster, 1995), by Michael A. Cusumano and Richard W. Selby

Since the mid-1980s, Microsoft and other personal-computer (PC) software firms have gradually been reorganizing the way they build software products in response to quality problems and delayed deliveries.[1] Many companies have also found it necessary to organize larger teams in order to build today's PC software products. These products now consist of hundreds of thousands and 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 18,000 employees, 200 products, and annual revenues of $6 billion (fiscal year ending June 1995), Microsoft has probably tackled more PC software projects than any other PC software company. Some of its products, such as Windows 95 (which contains more than 11 million lines of code and had a development team of more than 200 programmers and testers), rival the complexity of many systems produced by makers of software for mainframe computers and telecommunication systems.

Microsoft's general philosophy has been to maintain its roots as a highly flexible, entrepreneurial company, and *not* adopt too many of the structured software-engineering practices commonly promoted by organizations such as the Software Engineering Institute (SEI) and the International Standards Organization (ISO).[2] 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 (3 to 8 developers each) or individual programmers to work together as one relatively large team, in order to build large products relatively quickly, but still allow 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. Developers are free to innovate as they go along, however, so they must synchronize their changes frequently so that product components all work together.

In this article, we summarize how Microsoft uses various techniques and melds them into an overall approach for balancing 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 are we suggesting that Microsoft's software development methods have caused their great financial success. We are saying, however, that there are several lessons that can be learned from how Microsoft builds software products. Some of these lessons apply to other organizations – and some do not. Software developers and managers from other organizations can decide which ideas may apply to them, after considering various factors such as company goals, marketing strategies, resource constraints, software reliability requirements, and development culture.

FREQUENT SYNCHRONIZATIONS AND PERIODIC STABILIZATIONS

We have labeled 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 or what problems exist, usually by completely recompiling the source code and executing automated regression tests.) Whatever 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; they require much larger teams, who must also invent and innovate as they develop the product. Team members thus need to create components that are interdependent but difficult to define accurately in the early stages of the development cycle. In these situations, projects must find a way to proceed that structures and coordinates 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 allow a mechanism for 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 incremental changes in product development.[3] In the computer software community, since the mid-1970s, researchers and managers have talked about "iterative enhancement," a "spiral model" for iterating among the phases in product development, and "concurrent development" of multiple phases and activities.[4] Many firms have been slow to adopt these recommendations formally. Nonetheless, the basic idea shared among 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, that it is unwise to attempt to design a software system completely in advance. Instead, projects may need to iterate as well as concurrently manage many design, build, and testing cycles while they move forward to completing a product.

This iterative as well as incremental and concurrent-engineering style contrasts to a more sequential or "waterfall" approach to product development. In the waterfall approach, projects attempt to "freeze" a product specification, create a design, build components, and then merge these components together – primarily at the end of the project in one large integration and testing phase (Figure 1). This approach to software development was common in the 1970s and 1980s.[5] It also remains a basic model for project planning in many industries.[6] The waterfall model has gradually lost favor, however, because companies usually build improved 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 plus many others – now follow a process that iterates among designing, building components, and testing, as well as 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 different product "releases." In addition, many companies integrate pieces of their products together frequently (usually not daily, but often bi-weekly or monthly). This is useful to determine what works and what does not, without waiting until the end of the project – which may be several years in duration.

STRATEGIES AND PRINCIPLES

We observed Microsoft over a two-and-a-half year period, conducted in-depth interviews with 38 key people (including Bill Gates), and reviewed thousands of pages of confidential project documentation and "postmortem" reports. Through this field research, we identified two strategies and sets of principles that seem critical to making the synch-and-stabilize style of product development work.

Microsoft teams begin the process of product development by creating a "vision statement" that defines the goals for a new product and prioritizes the user activities that need to be supported by the product features (Figure 2). Product managers (marketing specialists) take charge of this task, which they do while consulting program managers, who specialize in writing up functional specifications of the product. Next, the program managers, in consultation with developers, write a functional specification that outlines the product features in sufficient depth to organize schedules and staffing allocations. But the specification document does not try to decide all the details of each feature, or lock the project into the original set of features. During the project, the team members will 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 percent 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 sub-projects) that represent completion points for major portions of the product (Figure 3). All the feature teams go through a complete cycle of development, feature integration, testing, and fixing problems in each milestone sub-project. Moreover, throughout the whole 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 sub-project, the developers fix almost all errors that have been 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.

Defining Products and Development Processes: Focus Creativity by Evolving Features and "Fixing" Resources

To define products and organize the development process, leading product groups in Microsoft follow a strategy that 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 (about 20% to 50% of total project time) and no separate product maintenance group.
- Use a "vision statement" and outline specification of features to guide projects.
- Base feature selection and prioritization on user activities and data.
- Evolve a modular and horizontal design architecture, with the product structure mirrored in the project structure.
- Control by individual commitments to small tasks and "fixed" project resources.

These principles are significant for several reasons. While having creative people in a high-technology company is important, it is often more important to *direct* their creativity. Managers can do this by getting development personnel to think about features that large amounts of people will pay money for, and by putting pressure on projects by limiting their resources, such as staffing and schedule. Otherwise, software developers run the risk of never shipping anything to market. This risk especially becomes a problem in fast-moving industries, when individuals or teams have unfocused or highly volatile user requirements, frequently change interdependent components during a project, or do not synchronize their work.

Microsoft also gets around these problems by structuring projects into sequential subprojects containing prioritized features, with buffer time within each sub-project to allow people time to respond to unexpected difficulties or delays. It uses vision statements and outline specifications rather than complete product specifications and detailed designs before coding, because teams realize that they cannot determine in advance everything that the developers will 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, development teams also try to come up with features that map directly to activities that typical customers perform, and this requires continual observation and testing with users during development.

Most product designs have modular architectures that allow 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 (half day to 3-day chunks, for example) and asked developers to commit personally to the schedules they set. Managers then "fix" project resources by limiting the number of people they allocate to any one project. They also try to limit the time projects spend, especially in applications like Office or multimedia products, so that teams can delete features if they fall too far behind. (Cutting features to save schedule time is not always possible with operating systems or communications products, since reliability of the system is more important than features, and many features are closely coupled and cannot be so easily deleted individually.)

Developing and Shipping Products: Do Everything in Parallel with Frequent Synchronizations

To manage the process of developing and shipping products, Microsoft follows another strategy that we describe as *do everything in parallel with frequent synchronizations.* 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.
- 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 trying 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," such as to speed up 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 and sequential phases, and precise rules and controls. This approach, however, may excessively restrain innovation, and underestimate the importance of synchronizing work frequently. Communication and coordination difficulties across the functions and phases may also result in the project taking more time and people to complete than projects that overlap tasks and make people share responsibilities and work in small, nimble teams. What Microsoft tries to do, then, is allow many small teams and individuals enough freedom to work in parallel yet still function as one 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 their pieces of code, they must do so by a particular time, such as by 2:00 PM or 5:00 PM. This allows the team to put available components together, 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 go to bed at 9:00 o'clock.)* 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 actually resembles Toyota's famous production system, where factory workers stop the manufacturing lines whenever they notice a defect in a car they are assembling.[7])

Microsoft's daily build process has several steps. First, in order to develop a feature for a product, a developer checks out private copies of source code files from a centralized master version of the source code. He implements his feature by making changes to his private copies of the source code files. The developer then creates a private build of the product that contains his new feature, and tests it. He then checks in the changes from his private copies of the source code files into the master version of the source code. The check-in process includes an automated regression test to help assure that his changes to the source code files do not cause errors elsewhere in the product. A developer usually checks his code back into the master copy at least twice a week, but he 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 creates a new internal release of the product and includes many steps that compile source code. The build process automatically translates the source code for a product into one or more executable files, and also may create various library files that allow end users to customize the product. The new internal release of the product built each day is called the "daily build." Daily builds are generated for each platform, such as Windows and Macintosh, and for each market, such as U.S. and the major international versions.

Product teams also test features as they build them from multiple perspectives, including bringing in customers from "off the street" to try prototypes in a usability lab. In addition, nearly all Microsoft teams work on a single physical site with common development languages (primarily C, with some C++ and assembler, as well as Visual Basic for user interfaces), 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 or 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 (Figure 4).

STRUCTURING THE "HACKER" APPROACH

Some people may argue that Microsoft's key practices in product development – daily synchronizations through product builds, periodic milestone stabilizations, and continual testing – are no more than process and technical "fixes" for a "hacker" software organization that is 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 sit down at a computer with no formal plans, designs, or processes, and just "bang on" a keyboard and "hack away" at coding.[8] This approach worked for small computer programs that one person or a small handful of people could write – such as the first versions of DOS, Lotus 1-2-3, WordPerfect, Word, or Excel. It became unworkable as PC software programs grew into hundreds of thousands and then millions of lines of code.

Formal plans and processes existed first in the mainframe computer industry, where software systems had grown to this million-line-plus size even by the end of the 1960s.[9] Yet PC software companies have been unwilling to give up their traditions and cultures completely. 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, we believe, 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 that 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, however, because of how 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 Publisher, Works, Excel, Word, Office, Windows NT, Windows 95, and other products. Of course, the synch-and-stabilize process does not guarantee on-time or completely bug-free products. Creating new, large-scale software products on a precisely predicted schedule and with no major defects are extremely difficult goals in the PC industry. Microsoft and other PC software companies also try to replace products quickly and usually announce overly ambitious deadlines, which contribute to their 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 now offers and plans to offer in the future.

We have noted that 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 the hard way that accumulating historical metric data is useful to analyze bug trends and establish realistic project schedules[10]). 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 as well as 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 anti-bureaucratic 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.

BENEFITS OF SYNCH-AND-STABILIZE

Although 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 will solve major problems with a single simplistic 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 more rigid styles of product development (Table 1).

Microsoft does have its weaknesses. For example, the company needs to pay more attention to 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 their development methods. In particular, new product areas such as video-on-demand have many tightly linked components with "real-time" constraints that require precise mathematical models of when video, audio, and user data can be delivered reliably and on time. Many existing and new products have an extremely large or infinite number of potential user conditions or scenarios to test, based on what hardware and applications the customer is using. These new products can benefit from some incremental changes in the development process. They will 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 approach provides a framework that has several benefits for managers and engineers engaged in product development:

- It *breaks down large products into manageable pieces* (a prioritized 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 continuously synchronizing changes, stabilizing the product in increments, and continuously finding and fixing problems.
- It *facilitates competition on customer input, product features, and short development times* by providing a mechanism to incorporate customer inputs, set priorities, complete the most important parts first, and change or cut 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 have been completed, and preserving process and product flexibility as well as opportunism throughout the lifecycle.

The ideas and examples revealed here provide *useful lessons for firms 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 life cycles, and competition based around evolving product features and de facto technical standards. In particular, how to coordinate 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 this 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.

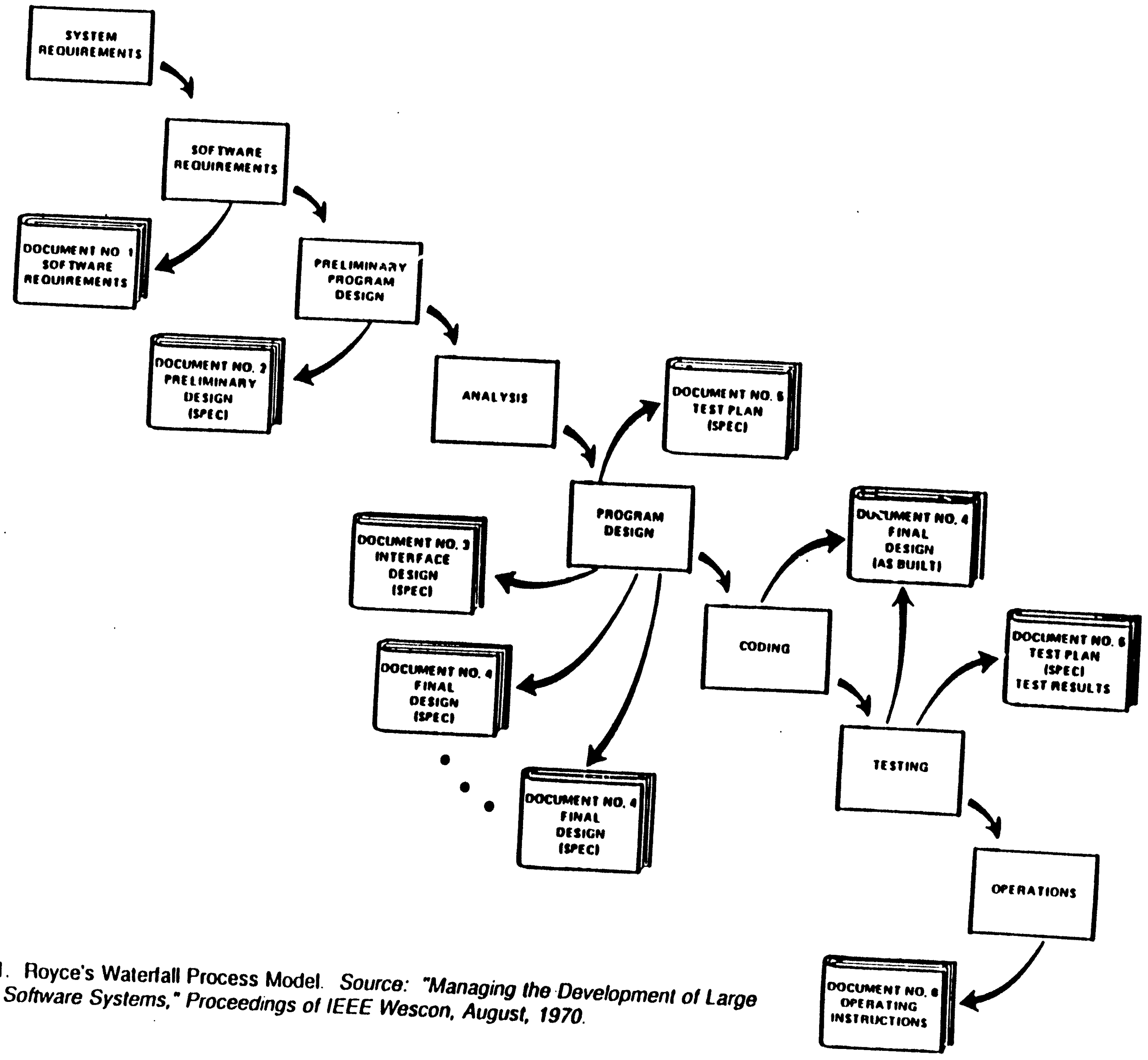

SYSTEM REQUIREMENTS
SOFTWARE REQUIREMENTS
DOCUMENT NO 1 SOFTWARE REQUIREMENTS
PRELIMINARY PROGRAM DESIGN
DOCUMENT NO. 2 PRELIMINARY DESIGN (SPEC)
ANALYSIS
DOCUMENT NO. 5 TEST PLAN (SPEC)
PROGRAM DESIGN
DOCUMENT NO. 3 INTERFACE DESIGN (SPEC)
DOCUMENT NO. 4 FINAL DESIGN (SPEC)
DOCUMENT NO. 4 FINAL DESIGN (SPEC)
CODING
DOCUMENT NO. 4 FINAL DESIGN (AS BUILT)
TESTING
DOCUMENT NO. 5 TEST PLAN (SPEC) TEST RESULTS
OPERATIONS
DOCUMENT NO. 6 OPERATING INSTRUCTIONS

Figure 1. Royce's Waterfall Process Model. *Source: "Managing the Development of Large Software Systems," Proceedings of IEEE Wescon, August, 1970.*

Figure 2 *Overview of Synch-and-Stabilize Development Approach*

Planning Phase: Define product vision, specification, and schedule.

• **Vision Statement** Product and program management use extensive customer input to identify and prioritize 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 up 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.

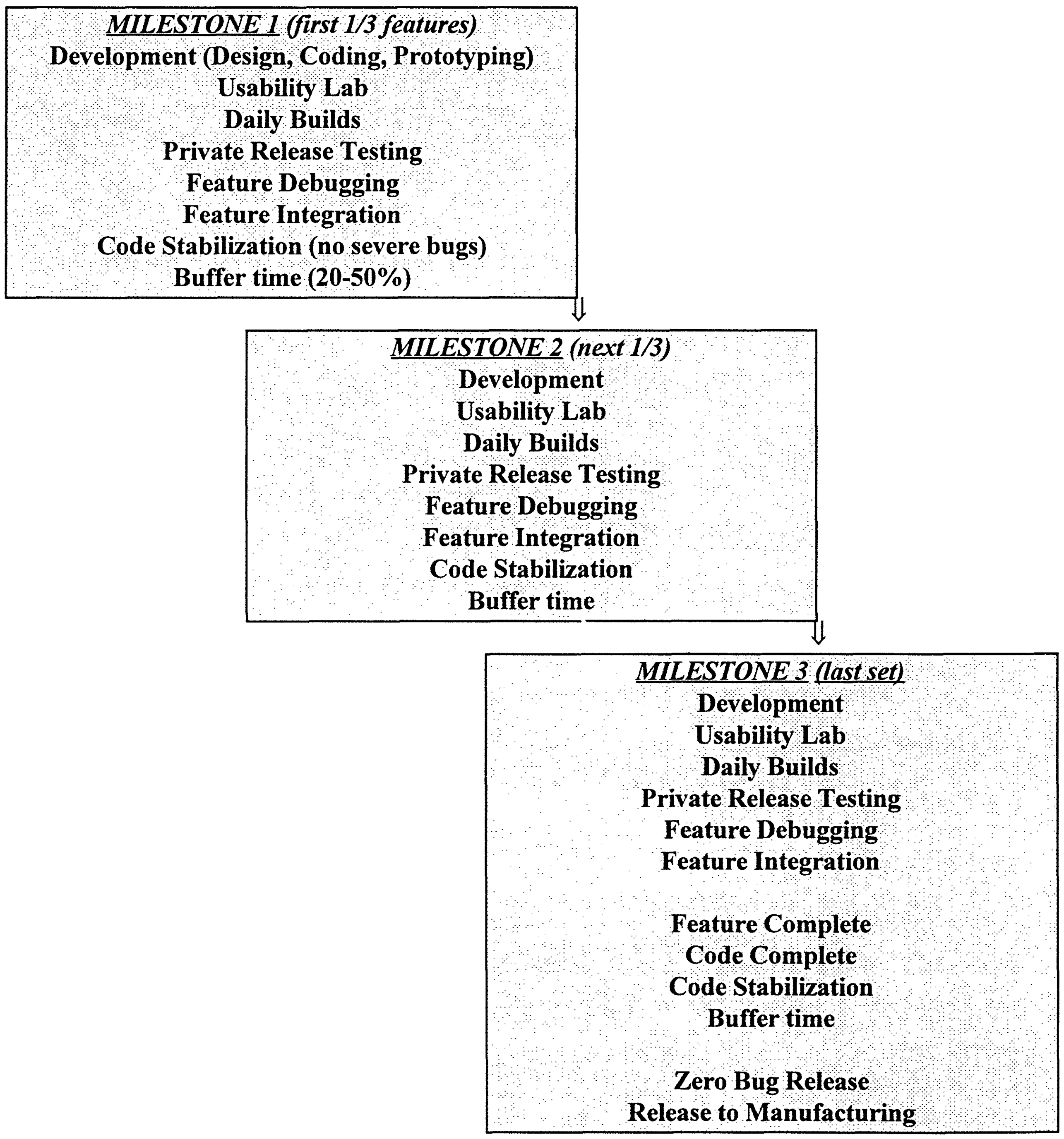
• **External Testing** Thorough testing of complete product outside the company by "beta" sites such as OEMs, ISVs, and end-users.

• **Release preparation** Prepare final release of "golden master" diskettes and documentation for manufacturing.

Source: Microsoft Secrets, p. 194.

Figure 3: SYNCH-and- STABILIZE Development Phase Breakdowns

Time: Usually 2 to 4 months per Milestone



MILESTONE 1 (first 1/3 features)
Development (Design, Coding, Prototyping)
Usability Lab
Daily Builds
Private Release Testing
Feature Debugging
Feature Integration
Code Stabilization (no severe bugs)
Buffer time (20-50%)

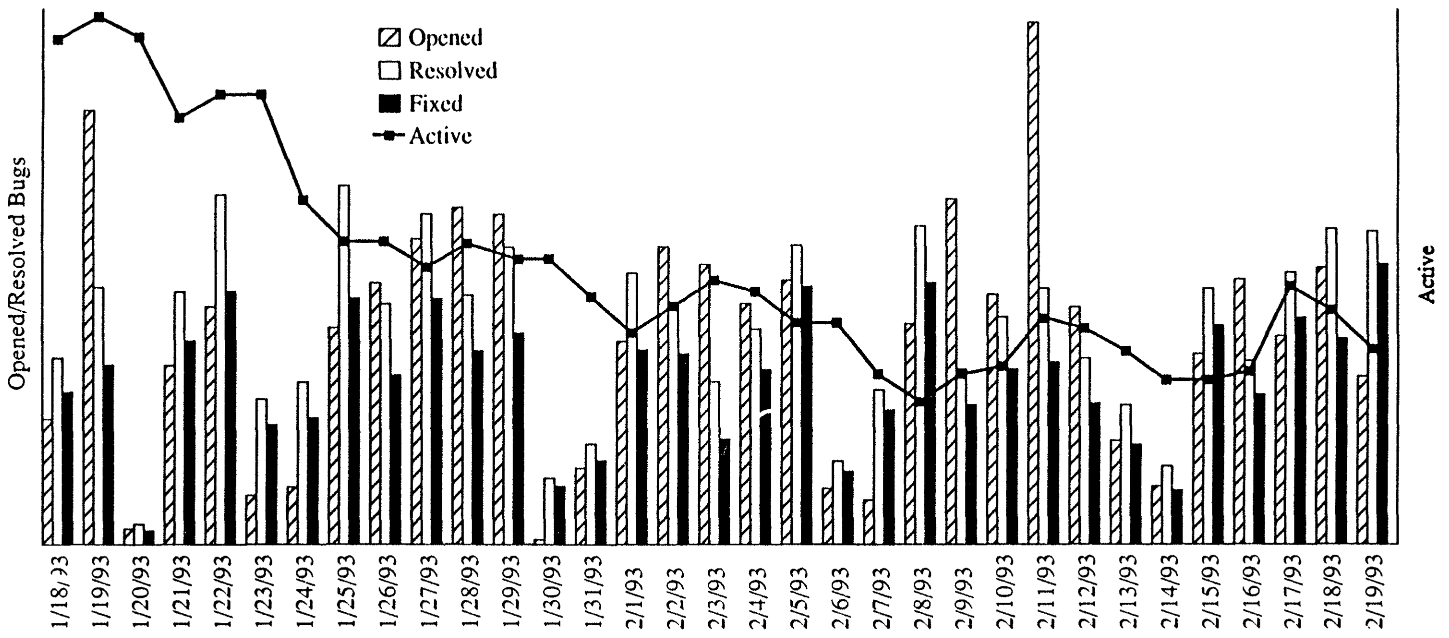
MILESTONE 2 (next 1/3)
Development
Usability Lab
Daily Builds
Private Release Testing
Feature Debugging
Feature Integration
Code Stabilization
Buffer time

MILESTONE 3 (last set)
Development
Usability Lab
Daily Builds
Private Release Testing
Feature Debugging
Feature Integration

Feature Complete
Code Complete
Code Stabilization
Buffer time

Zero Bug Release
Release to Manufacturing

Figure 4 Bug Data and Daily Builds from Excel/Graph 5.0, Milestone 2



Source: Microsoft internal document.

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 separate functional departments

Source: Microsoft Secrets, p. 407

Michael A. Cusumano is Professor, Sloan School of Management, Massachusetts Institute of Technology, 50 Memorial Drive, Cambridge, MA 02139. email: cusumano@mit.edu

Richard W. Selby is Associate Professor, Dept. of Information and Computer Science, University of California, Irvine, CA 92717. email: selby@ics.uci.edu

END NOTES

[1] See Stanley A. Smith and Michael A. Cusumano, "Beyond the Software Factory: A Comparison of 'Classic' and 'PC' Software Developers" (Cambridge, MA, MIT Sloan School Working Paper #3607-93/BPS, September 1993),

[2] See Watts S. Humphrey, Managing the Software Process, (New York, Addison Wesley, 1989).

[3] See examples in Steven C. Wheelright and Kim B. Clark, Revolutionizing Product Development (New York, Free Press, 1992).

[4] See Victor R. Basili and Albert J. Turner, "Iterative Enhancement: A Practical Technique for Software Development," IEEE Transactions on Software Engineering, Vol. SE-1, No. 4 (December 1975); Barry W. Boehm, "A Spiral Model of Software Development and Enhancement," IEEE Computer, May 1988; and Mikio Aoyama, "Concurrent-Development Process Model," IEEE Software, July 1993.

[5] See Winston W. Royce, "Managing the Development of Large Software Systems," Proceedings of IEEE WESCON, August 1970, pp. 1-9.

[6] See Wheelwright and Clark as well as, for example, Glen L. Urban and John R. Hauser, Design and Marketing of New Products (Englewood Cliffs, N.J., Prentice Hall, 199?).

[7] See Michael A. Cusumano, The Japanese Automobile Industry: Technology and Management at Nissan and Toyota (Cambridge, MA, Harvard University Press, 1985).

[8] See Steven Levy, Hackers: Heroes of the Computer Revolution (New York, Anchor/Doubleday, 1984).

[9] See Michael A. Cusumano, Japan's Software Factories: A Challenge to U.S. Management (New York, Oxford University Press, 1991).

[10] See Richard W. Selby, "Empirically Based Analysis of Failures in Software Systems," IEEE Transactions on Reliability, Vol. 39, No. 4, October 1990, pp. 444-454