Microsoft

BEST PRACTICES

HOW WE TEST SOFTWARE AT MICROSOFT

Alan Page, Ken Johnston, Bj Rollison

Microsoft®

How We Test Software at Microsoft®

Alan Page
Ken Johnston
Bj Rollison

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2008940534

Printed and bound in the United States of America.

ISBN: 9780735624252

Second Printing: July 2014

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Acquisitions Editor: Ben Ryan
Project Editor: Lynn Finnel
Editorial Production: Waypoint Press
Cover Illustration: John Hersey

Body Part No. X14-71546

To my wife, Christina, and our children, Cole and Winona, who sacrificed their time with me so I could write this book; and for my parents, Don and Arlene, for their constant support, and for giving me a sanctuary to write.

—Alan Page

To my children, David and Grace, for allowing their dad the time to write; and to my wife, Karen, who while I was working on a presentation for a testing conference first suggested, "Why don't you just call it 'How we test at Microsoft.'" Without those words (and Alan leading the way), we would not have started or finished this project.

—Ken Johnston

To my mother and father for their unending love, sage wisdom, and especially their patience. I also want to thank my 6-year-old daughter Elizabeth whose incessant curiosity to learn new things and persistent determination to conquer diverse challenges has taught me that the only problems we cannot overcome are those for which we have not yet found a solution.

—Bj Rollison

This book is dedicated to the test engineers at Microsoft who devote themselves to the most challenging endeavor in the software process, and who continue to mature the discipline by breaking through traditional barriers and roles in order to help ship leading-edge, high-quality software to our customers. For us, it is truly a privilege to mentor and work alongside so many professional testers at Microsoft, because through our interactions with them we also continue to learn more about software testing.

Contents at a Glance

Table of Contents

Acknowledgments

This book never would have happened without the help of every single tester at Microsoft. Many helped directly by reviewing chapters or writing about their experiences in testing. Others helped by creating the legacy of software testing at Microsoft, or by continuing to innovate in the way we test software.

Including the names of all 9,000 testers at Microsoft would be impractical (especially when many former employees, people from other disciplines, and even external reviewers contributed to the completion of this book). On the other hand, we do want to call out the names of several people who have contributed substantially to the creation of this book.

This book grew out of the opinions, suggestions, and feedback of many current and former Microsoft employees. Some of the most prominent contributors include Michael Corning, Ed Triou, Amol Kher, Scott Wadsworth, Geoff Staneff, Dan Travison, Brian Rogers, John Lambert, Sanjeev Verma, Shawn McFarland, Grant George, Tara Roth, Karen Carter-Schwendler, Jean Hartman, James Whittaker, Irada Sadykhova, Alex Kim, Darrin Hatakeda, Marty Riley, Venkat Narayanan, Karen Johnston, Jim Pierson, Ibrahim El Far, Carl Tostevin, Nachi Nagappan, Keith Stobie, Mark Davis, Mike Blaylock, Wayne Roseberry, Carole Cancler, Andy Tischaefer, Lori Ada-Kilty, Matt Heusser, Jeff Raikes, Microsoft Research (especially Amy Stevenson), the Microsoft Test Excellence Team, the Microsoft Test Leadership Team, and the Microsoft Test Architect Group.

We'd also like to thank Lynn Finnel, the Project Editor for this book, who continued to give us encouragement and support throughout the creation of this book.

Introduction

I still remember the morning, sometime late in the fall of 2007, when my manager at the time, Ken Johnston, uttered these five words, "You should write a book."

He had just come back from delivering a talk at an industry test conference (not coincidentally titled, "How We Test Software at Microsoft,") and was excited by the audience reception. Ken loves to give presentations, but he somehow thought I should be the one to write the book.

I humored him and said, "Sure, why not." I went on to say that the book could cover a lot of the things that we teach in our software testing courses, as well as a smattering of other popular test approaches used at Microsoft. It could be interesting, but there are a ton of books on testing—I know, I've probably read a few dozen of them—and some of them are really good. What value to the testing community could yet another book provide?

I was about to talk the nonsense out of Ken when I realized something critical: At Microsoft, we have some of the best software test training in the world. The material and structure of the courses are fantastic, but that's not what makes it so great. The way our instructors tie in anecdotes, success stories, and cool little bits of trivia throughout our courses is what makes them impactful and memorable. I thought that if we could include some stories and bits of information on how Microsoft has used some of these approaches, the book might be interesting. I began to think beyond what we teach, of more test ideas and stories that would be fun to share with testers everywhere. I realized that some of my favorite programming books were filled with stories embedded with all of the "techie" stuff.

The next thing I knew, I was writing a proposal. An outline began to come together, and the form of the book began to take shape, with four main themes emerging. It made sense to set some context by talking about Microsoft's general approaches to people and engineering. The next two sections would focus on how we do testing inside Microsoft, and the tools we use; and the final section would look at the future of testing inside Microsoft. I sent the proposal to Microsoft Press, and although I remained excited about the potential for the book, part of me secretly hoped that Microsoft Press would tell me the idea was silly, and that I should go away. Alas, that didn't happen, and shortly thereafter, I found myself staring at a computer screen wondering what the first sentences would look like.

From the very beginning, I knew that I wanted Ken to write the first two chapters. Ken has been a manager at Microsoft for years, and the people stuff was right up his alley. About the time I submitted the proposal, Ken left our group to manage the Office Online group. Soon after, it became apparent that Ken should also write the chapter on how we test Software plus Services. He's since become a leader at the company in defining how we test Web services, and it would have been silly not to have him write Chapter 14, "Testing Software

Plus Services". Later on, I approached BJ Rollison, one of Microsoft's most prominent testers, to write the chapters about functional and structural test techniques. Bj Rollison designed our core software testing course, and he knows more about these areas of testing than anyone I know. He's also one of the only people I know who has read more books on testing than I have. Ken, Bj and I make quite a trio of authors. We all approach the task and produce our material quite differently, but in the end, we feel like we have a mix of both material and writing styles that reflects the diversity of the Microsoft testing population. We often joke that Bj is the professor, Ken tries to be the historian and storyteller, and I just absorb information and state the facts. Although we all took the lead on several chapters, we each edited and contributed to the others' work, so there is definitely a melding of styles throughout the book.

I cannot begin to describe how every little setback in life becomes gigantic when the task of "writing a book" is always on your plate. Since starting this book, I took over Ken's old job as Director of Test Excellence at Microsoft. Why in the world I decided to take on a job with entirely new challenges in the middle of writing a book I'll never know. In hindsight, however, taking on this role forced me to gain some insight into test leadership at Microsoft that benefitted this book tremendously.

My biggest fear in writing this book was how much I knew I'd have to leave out. There are over 9,000 testers at Microsoft. The test approaches discussed in this book cover what most testers at Microsoft do, but there are tons of fantastically cool things that Microsoft testers do that couldn't be covered in this book. On top of that, there are variations on just about every topic covered in this book. We tried to capture as many different ideas as we could, while telling stories about what parts of testing we think are most important. I also have to admit that I'm slightly nervous about the title of this book. "How We Test Software at Microsoft" could imply that everything in this book is done by every single tester at Microsoft, and that's simply not true. With such a large population of testers and such a massive product portfolio, there's just no way to write about testing in a way that exactly represents every single tester at Microsoft. So, we compromised. This book simply covers the most popular testing practices, tools, and techniques used by Microsoft testers. Not every team does everything we write about, but most do. Everything we chose to write about in this book has been successful in testing Microsoft products, so the topics in this book are a collection of some of the things we know work.

In the end, I think we succeeded, but as testers, we know it could be better. Sadly, it's time to ship, but we do have a support plan in place! If you are interested in discussing anything from this book with the authors, you can visit our web site, *www.hwtsam.com*. We would all love to hear what you think.

—Alan Page

Who This Book Is For

This book is for anyone who is interested in the role of test at Microsoft or for those who want to know more about how Microsoft approaches testing. This book isn't a replacement for any of the numerous other great texts on software testing. Instead, it describes how Microsoft applies a number of testing techniques and methods of evaluation to improve our software.

Microsoft testers themselves will likely find the book to be interesting as it includes techniques and approaches used across the company. Even nontesters may find it interesting to know about the role of test at Microsoft

What This Book Is About

This book starts by familiarizing the reader with Microsoft products, Microsoft engineers, Microsoft testers, the role of test, and general approaches to engineering software. The second part of the book discusses many of the test approaches and tools commonly used at Microsoft. The third part of the book discusses some of the tools and systems we use in our work. The final section of the book discusses future directions in testing and quality at Microsoft and how we intend to create that future.

Part I, "About Microsoft"

- Chapter 1, "Software Engineering at Microsoft,"
- Chapter 2, "Software Test Engineers at Microsoft"
- Chapter 3, "Engineering Life Cycles"

Part II, "About Testing"

- Chapter 4, "A Practical Approach to Test Case Design"
- Chapter 5, "Functional Testing Techniques"
- Chapter 6, "Structural Testing Techniques"
- Chapter 7, "Analyzing Risk with Code Complexity"
- Chapter 8, "Model-Based Testing"

Part III, "Test Tools and Systems"

- Chapter 9, "Managing Bugs and Test Cases"
- Chapter 10, "Test Automation"
- Chapter 11, "Non-Functional Testing"
- Chapter 12, "Other Tools"
- Chapter 13, "Customer Feedback Systems"
- Chapter 14, "Testing Software Plus Services"

Find Additional Content Online

As new or updated material becomes available that complements this book, it will be posted online on the Microsoft Press Online Developer Tools Web site. The type of material you might find includes updates to book content, articles, links to companion content, errata, sample chapters, and more. This Web site is available at *www.microsoft.com/learning/books/online/developer*, and is updated periodically.

More stories and tidbits about testing at Microsoft will be posted on *www.hwtsam.com*.

Support for This Book

If you have comments, questions, or ideas regarding the book, or questions that are not answered by visiting the sites above, please send them to Microsoft Press via e-mail to

mspinput@microsoft.com.

Or via postal mail to

Microsoft Press
Attn: *How We Test Software at Microsoft* Editor
One Microsoft Way
Redmond, WA 98052-6399.

Please note that Microsoft software product support is not offered through the above addresses.

Chapter 3

Engineering Life Cycles

Alan Page

I love to cook. Something about the entire process of creating a meal, coordinating multiple dishes, and ensuring that they all are complete at the exact same time is fun for me. My approach, learned from my highly cooking-talented mother, includes making up a lot of it as I go along. In short, I like to "wing it." I've cooked enough that I'm comfortable browsing through the cupboard to see which ingredients seem appropriate. I use recipes as a guideline—as something to give me the general idea of what kinds of ingredients to use, how long to cook things, or to give me new inspiration. There is a ton of flexibility in my approach, but there is also some amount of risk. I might make a poor choice in substitution (for example, I recommend that you never replace cow's milk with soymilk when making strata).

My approach to cooking, like testing, depends on the situation. For example, if guests are coming for dinner, I might measure a bit more than normal or substitute less than I do when cooking just for my family. I want to reduce the risk of a "defect" in the taste of my risotto, so I put a little more formality into the way I make it. I can only imagine the chef who is in charge of preparing a banquet for a hundred people. When cooking for such a large number of people, measurements and proportions become much more important. In addition, with such a wide variety of taste buds to please, the chef's challenge is to come up with a combination of flavors that is palatable to all of the guests. Finally, of course, the entire meal needs to be prepared and all elements of the meal need to be freshly hot and on the table exactly on time. In this case, the "ship date" is unchangeable!

Making software has many similarities with cooking. There are benefits to following a strict plan and other benefits that can come from a more flexible approach, and additional challenges can occur when creating anything for a massive number of users. This chapter describes a variety of methods used to create software at Microsoft.

Software Engineering at Microsoft

There is no "one model" that every product team at Microsoft uses to create software. Each team determines, given the size and scope of the product, market conditions, team size, and prior experiences, the best model for achieving their goals. A new product might be driven by time to market so as to get in the game before there is a category leader. An established product might need to be very innovative to unseat a leading competitor or to stay ahead of the pack. Each situation requires a different approach to scoping, engineering, and shipping the product. Even with the need for variation, many practices and approaches have

become generally adopted, while allowing for significant experimentation and innovation in engineering processes.

For testers, understanding the differences between common engineering models, the model used by their team, and what part of the model their team is working in helps both in planning (knowing what will be happening) and in execution (knowing the goals of the current phase of the model). Understanding the process and their role in the process is essential for success.

Traditional Software Engineering Models

Many models are used to develop software. Some development models have been around for decades, whereas others seem to pop up nearly every month. Some models are extremely formal and structured, whereas others are highly flexible. Of course, there is no single model that will work for every software development team, but following some sort of proven model will usually help an engineering team create a better product. Understanding which parts of development and testing are done during which stages of the product cycle enables teams to anticipate some types of problems and to understand sooner when design or quality issues might affect their ability to release on time.

Waterfall Model

One of the most commonly known (and commonly abused) models for creating software is the waterfall model. Waterfall is an approach to software development where the end of each phase coincides with the beginning of the next phase, as shown in Figure 3-1. The work follows steps through a specified order. The implementation of the work "flows" from one phase to another (like a waterfall flows down a hill).

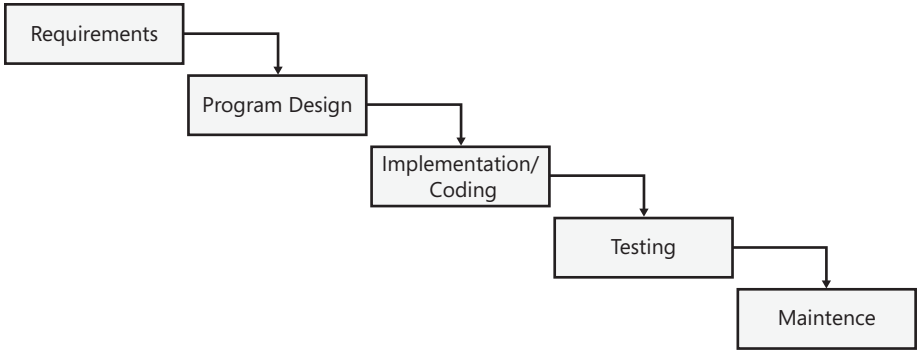

FIGURE 3-1 Waterfall model.

The advantage of this model is that when you begin a phase, everything from the previous phase is complete. Design, for example, will never begin before the requirements are complete. Another potential benefit is that the model forces you to think and design as much as possible before beginning to write code. Taken literally, waterfall is inflexible because it doesn't appear to allow phases to repeat. If testing, for example, finds a bug that leads back to a design flaw, what do you do? The Design phase is "done." This apparent inflexibility has led to many criticisms of waterfall. Each stage has the potential to delay the entire product cycle, and in a long product cycle, there is a good chance that at least some parts of the early design become irrelevant during implementation.

An interesting point about waterfall is that the inventor, Winston Royce, intended for waterfall to be an iterative process. Royce's original paper on the model[1] discusses the need to iterate at least twice and use the information learned during the early iterations to influence later iterations. Waterfall was invented to improve on the stage-based model in use for decades by recognizing feedback loops between stages and providing guidelines to minimize the impact of rework. Nevertheless, waterfall has become somewhat of a ridiculed process among many software engineers—especially among Agile proponents. In many circles of software engineering, *waterfall* is a term used to describe *any* engineering system with strict processes.

Spiral Model

In 1988, Barry Boehm proposed the spiral model of software development.[2] Spiral, as shown in Figure 3-2, is an iterative process containing four main phases: determining objectives, risk evaluation, engineering, and planning for the next iteration.

- **Determining objectives** Identify and set specific objectives for the current phase of the project.
- **Risk evaluation** Identify key risks, and identify risk reduction and contingency plans. Risks might include cost overruns or resource issues.
- **Engineering** In the engineering phase, the work (requirements, design, development, testing, and so forth) occurs.
- **Planning** The project is reviewed, and plans for the next round of the spiral begin.

1 Winston Royce, "Managing the Development of Large Software Systems," *Proceedings of IEEE WESCON* 26 (August 1970).

2 Barry Boehm, "A Spiral Model of Software Development," *IEEE* 21, no. 5 (May 1988): 61–72.

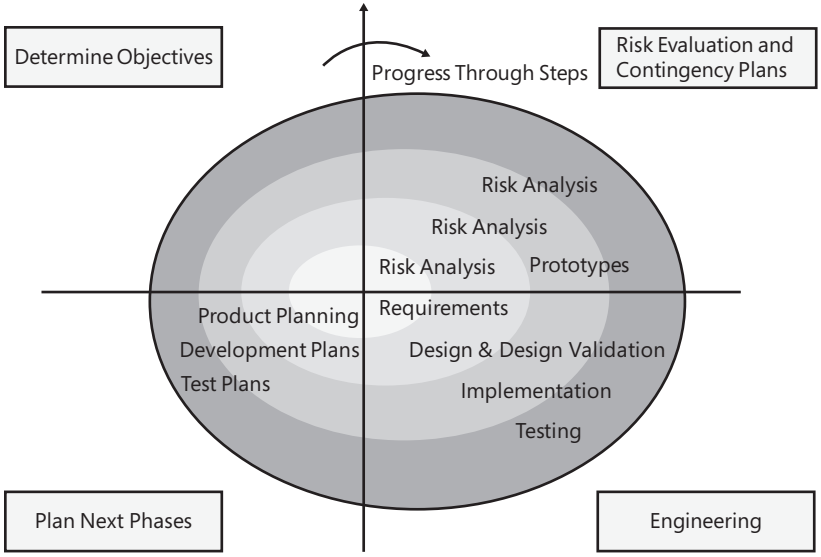

FIGURE 3-2 Simplified spiral model.

Another important concept in the spiral model is the repeated use of prototypes as a means of minimizing risk. An initial prototype is constructed based on preliminary design and approximates the characteristics of the final product. In subsequent iterations, the prototypes help evaluate strengths, weaknesses, and risks.

Software development teams can implement spiral by initially planning, designing, and creating a bare-bones or prototype version of their product. The team then gathers customer feedback on the work completed, and then analyzes the data to evaluate risk and determine what to work on in the next iteration of the spiral. This process continues until either the product is complete or the risk analysis shows that scrapping the project is the better (or less risky) choice.

Agile Methodologies

By using the spiral model, teams can build software iteratively—building on the successes (and failures) of the previous iterations. The planning and risk evaluation aspects of spiral are essential for many large software products but are too process heavy for the needs of many software projects. Somewhat in response to strict models such as waterfall, Agile approaches focus on lightweight and incremental development methods.

Agile methodologies are currently quite popular in the software engineering community. Many distinct approaches fall under the Agile umbrella, but most share the following traits:

- **Multiple, short iterations** Agile teams strive to deliver working software frequently and have a record of accomplishing this.
- **Emphasis on face-to-face communication and collaboration** Agile teams value interaction with each other and their customers.
- **Adaptability to changing requirements** Agile teams are flexible and adept in dealing with changes in customer requirements at any point in the development cycle. Short iterations allow them to prioritize and address changes frequently.
- **Quality ownership throughout the product cycle** Unit testing is prevalent among developers on Agile teams, and many use test-driven development (TDD), a method of unit testing where the developer writes a test before implementing the functionality that will make it pass.

In software development, to be Agile means that teams can quickly change direction when needed. The goal of always having working software by doing just a little work at a time can achieve great results, and engineering teams can almost always know the status of the product. Conversely, I can recall a project where we were "95 percent complete" for at least three months straight. In hindsight, we had no idea how much work we had left to do because we tried to do everything at once and went months without delivering working software. The goal of Agile is to do a little at a time rather than everything at once.

Other Models

Dozens of models of software development exist, and many more models and variations will continue to be popular. There isn't a best model, but understanding the model and creating software within the bounds of whatever model you choose can give you a better chance of creating a quality product.

Milestones

It's unclear if it was intentional, but most of the Microsoft products I have been involved in used the spiral model or variations.[3] When I joined the Windows 95 team at Microsoft, they were in the early stages of "Milestone 8" (or M8 as we called it). M8, like one of its predecessors, M6, ended up being a public beta. Each milestone had specific goals for product functionality and quality. Every product I've worked on at Microsoft, and many others I've worked with indirectly, have used a milestone model.

[3] Since I left product development in 2005 to join the Engineering Excellence team, many teams have begun to adopt Agile approaches.

The milestone schedule establishes the time line for the project release and includes key interim project deliverables and midcycle releases (such as beta and partner releases). The milestone schedule helps individual teams understand the overall project expectations and to check the status of the project. An example of the milestone approach is shown in Figure 3-3.

FIGURE 3-3 Milestone model example.

The powerful part of the milestone model is that it isn't just a date drawn on the calendar. For a milestone to be complete, specific, predefined criteria must be satisfied. The criteria typically include items such as the following:

- **"Code complete" on key functionality** Although not completely tested, the functionality is implemented.
- **Interim test goals accomplished** For example, code coverage goals or tests completed goals are accomplished.
- **Bug goals met** For example, no severity 1 bugs or no crashing bugs are known.
- **Nonfunctional goals met** For example, performance, stress, load testing is complete with no serious issues.

The criteria usually grow stricter with each milestone until the team reaches the goals required for final release. Table 3-1 shows the various milestones used in a sample milestone project.

TABLE 3-1 Example Milestone Exit Criteria (partial list)

Area	Milestone 1	Milestone 2	Milestone 3	Release
Test case execution		All Priority 1 test cases run	All Priority 1 and 2 test cases run	All test cases run
Code coverage	Code coverage measured and reports available	65% code coverage	75% code coverage	80% code coverage
Reliability	Priority 1 stress tests running nightly	Full stress suite running nightly on at least 200 computers	Full stress suite running nightly on at least 500 computers with no uninvestigated issues	Full stress suite running nightly on at least 500 computers with no uninvestigated issues
Reliability		Fix the top 50% of customer-reported crashes from M1	Fix the top 60% of customer-reported crashes from M2	Fix the top 70% of customer-reported crashes from M3

Area	Milestone 1	Milestone 2	Milestone 3	Release
Features		New UI shell in 20% of product	New UI in 50% of product and usability tests complete	New UI in 100% of product and usability feedback implemented
Performance	Performance plan, including scalability goals, complete	Performance baselines established for all primary customer scenarios	Full performance suite in place with progress tracking toward ship goals	All performance tests passing, and performance goals met

Another advantage of the milestone model (or any iterative approach) is that with each milestone, the team gains some experience going through the steps of release. They learn how to deal with surprises, how to ask good questions about unmet criteria points, and how to anticipate and handle the rate of incoming bugs. An additional intent is that each milestone release functions as a complete product that can be used for large-scale testing (even if the milestone release is not an external beta release). Each milestone release is a complete version of the product that the product team and any other team at Microsoft can use to "kick the tires" on (even if the tires are made of cardboard).

The quality milestone

Several years ago, I was on a product team in the midst of a ship cycle. I was part of the daily bug triage, where we reviewed, assigned, and sometimes postponed bugs to the next release. Postponements happen for a variety of reasons and are a necessary part of shipping software. A few months before shipping, we had some time left at the end of the meeting, and I asked if we could take a quick look at the bugs assigned to the next version of our product. The number was astounding. It was so large that we started calling it the "wave." The wave meant that after we shipped, we would be starting work on the next release with a huge backlog of product bugs.

Bug backlog along with incomplete documents and flaky tests we need to fix "someday" are all items that add up to *technical debt*.[4] We constantly have to make tradeoffs when developing software, and many of those tradeoffs result in technical debt. Technical debt is difficult to deal with, but it just doesn't go away if we ignore it, so we have to do something. Often, we try to deal with it while working on other things or in the rare times when we get a bit of a lull in our schedules. This is about as effective as bailing out a leaky boat with a leaky bucket.

Another way many Microsoft teams have been dealing with technical debt is with a *quality milestone*, or MQ. This milestone, which occurs after product release but before

[4] Matthew Heusser writes about technical debt often on his blog (*xndev.blogspot.com*). Matt doesn't work for Microsoft…yet.

getting started on the next wave of product development, provides an opportunity for teams to fix bugs, retool their infrastructure, and fix anything else pushed aside during the previous drive to release. MQ is also an opportunity to implement improvements to any of the engineering systems or to begin developing early prototypes of work and generate new ideas.

Beginning a product cycle with the backlog of bugs eliminated, the test infrastructure in place, improvement policies implemented, and everything else that annoyed you during the previous release resolved is a great way to start work on a new version of a mature product.

Agile at Microsoft

Agile methodologies are popular at Microsoft. An internal e-mail distribution list dedicated to discussion of Agile methodologies has more than 1,500 members. In a survey sent to more than 3,000 testers and developers at Microsoft, approximately one-third of the respondents stated that they used some form of Agile software development.[5]

Feature Crews

Most Agile experts state that a team size of 10 or less collocated team members is optimal. This is a challenge for large-scale teams with thousands or more developers. A solution commonly used at Microsoft to scale Agile practices to large teams is the use of *feature crews*.

A feature crew is a small, cross-functional group, composed of 3 to 10 individuals from different disciplines (usually Dev, Test, and PM), who work autonomously on the end-to-end delivery of a functional piece of the overall system. The team structure is typically a program manager, three to five testers, and three to five developers. They work together in short iterations to design, implement, test, and integrate the feature into the overall product, as shown in Figure 3-4.

The key elements of the team are the following:

- It is independent enough to define its own approach and methods.
- It can drive a component from definition, development, testing, and integration to a point that shows value to the customer.

Teams in Office and Windows use this approach as a way to enable more ownership, more independence, and still manage the overall ship schedule. For the Office 2007 project, there were more than 3,000 feature crews.

[5] Nachiappan Nagappan and Andrew Begel, "Usage and Perceptions of Agile Software Development in an Industrial Context: An Exploratory Study," 2007, *http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/proceedings/&toc=comp/proceedings/esem/2007/2886/00/2886toc.xml&DOI=10.1109/ESEM.2007.85*.

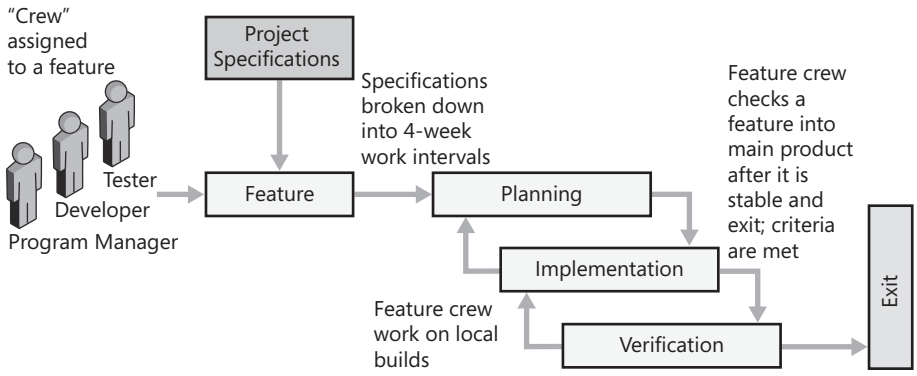

FIGURE 3-4 Feature crew model.

Getting to Done

To deliver high-quality features at the end of each iteration, feature crews concentrate on defining "done" and delivering on that definition. This is most commonly accomplished by defining *quality gates* for the team that ensure that features are complete and that there is little risk of feature integration causing negative issues. Quality gates are similar to milestone exit criteria. They are critical and often require a significant amount of work to satisfy. Table 3-2 lists sample feature crew quality gates.[6]

TABLE 3-2 Sample Feature Crew Quality Gates

Quality gate	Description
Testing	All planned automated tests and manual tests are completed and passing.
Feature Bugs Closed	All known bugs found in the feature are fixed or closed.
Performance	Performance goals for the product are met by the new feature.
Test Plan	A test plan is written that documents all planned automated and manual tests.
Code Review	Any new code is reviewed to ensure that it meets code design guidelines.
Functional Specification	A functional spec has been completed and approved by the crew.
Documentation Plan	A plan is in place for the documentation of the feature.
Security	Threat model for the feature has been written and possible security issues mitigated.
Code Coverage	Unit tests for the new code are in place and ensure 80% code coverage of the new feature.
Localization	The feature is verified to work in multiple languages.

[6] This table is based on Ade Miller and Eric Carter, "Agile and the Inconceivably Large," *IEEE* (2007).

The feature crew writes the necessary code, publishes private releases, tests, and iterates while the issues are fresh. When the team meets the goals of the quality gates, they migrate their code to the main product source branch and move on to the next feature. I. M. Wright's *Hard Code* (Microsoft Press, 2008) contains more discussion on the feature crews at Microsoft.

Iterations and Milestones

Agile iterations don't entirely replace the milestone model prevalent at Microsoft. Agile practices work hand in hand with milestones—on large product teams, milestones are the perfect opportunity to ensure that all teams can integrate their features and come together to create a product. Although the goal on Agile teams is to have a shippable product at all times, most Microsoft teams release to beta users and other early adopters every few months. Beta and other early releases are almost always aligned to product milestones.

Putting It All Together

At the micro level, the smallest unit of output from developers is code. Code grows into functionality, and functionality grows into features. (At some point in this process, test becomes part of the picture to deliver *quality* functionality and features.)

In many cases, a large group of features becomes a *project*. A project has a distinct beginning and end as well as checkpoints (milestones) along the way, usage scenarios, personas, and many other items. Finally, at the top level, subsequent releases of related projects can become a product line. For example, Microsoft Windows is a product line, the Windows Vista operating system is a project within that product line, and hundreds of features make up that project.

Scheduling and planning occur at every level of output, but with different context, as shown in Figure 3-5. At the product level, planning is heavily based on long-term strategy and business need. At the feature level, on the other hand, planning is almost purely tactical—getting the work done in an effective and efficient manner is the goal. At the project level, plans are often both tactical and strategic—for example, integration of features into a scenario might be tactical work, whereas determining the length of the milestones and what work happens when is more strategic. Classifying the work into these two buckets isn't important, but it is critical to integrate strategy and execution into large-scale plans.

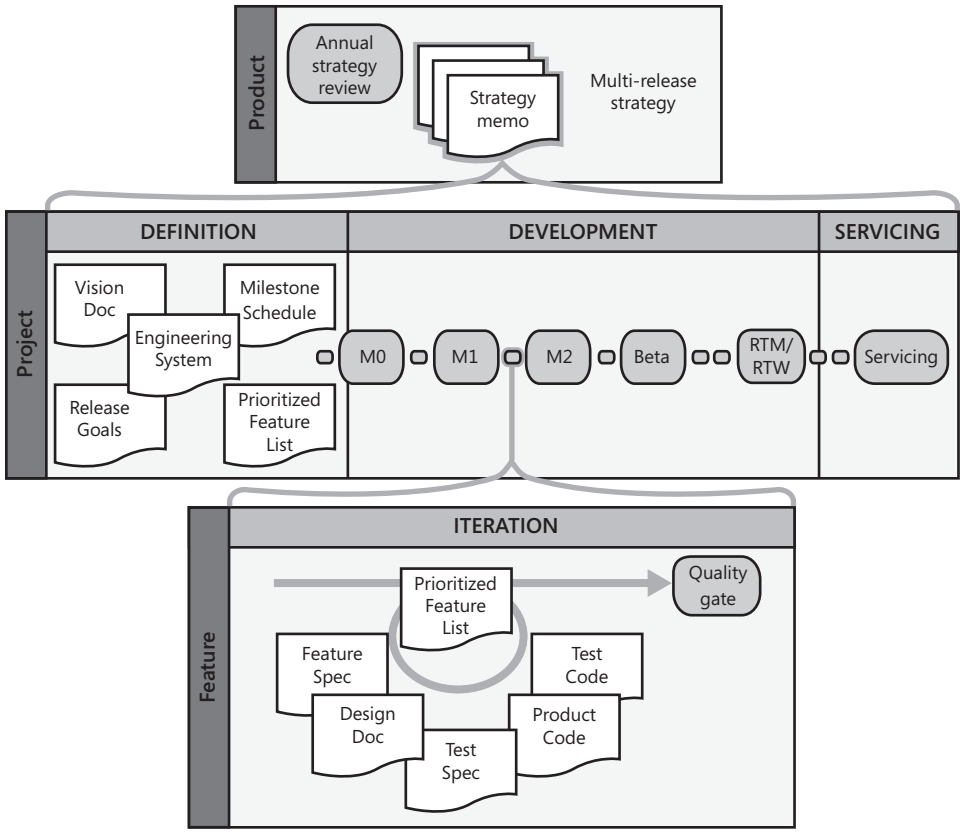

FIGURE 3-5 Software life cycle workflow.

Process Improvement

In just about anything I take seriously, I want to improve continuously. Whether I'm preparing a meal, working on my soccer skills, or practicing a clarinet sonata, I want to get better. Good software teams have the same goal—they reflect often on what they're doing and think of ways to improve.

Dr. W. Edwards Deming is widely acknowledged for his work in quality and process improvement. One of his most well known contributions to quality improvement was the simple *Plan, Do, Check, Act* cycle (sometimes referred to as the Shewhart cycle, or the PDCA cycle). The following phases of the PDCA cycle are shown in Figure 3-6:

- **Plan** Plan ahead, analyze, establish processes, and predict the results.
- **Do** Execute on the plan and processes.

- **Check** Analyze the results (note that Deming later changed the name of this stage to "Study" to be more clear).
- **Act** Review all steps and take action to improve the process.

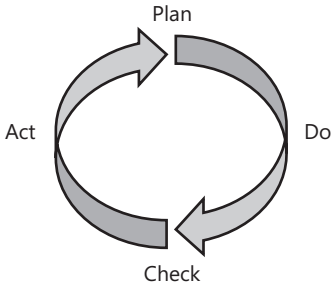

FIGURE 3-6 Deming's PDCA cycle.

For many people, the cycle seems so simple that they see it as not much more than common sense. Regardless, this is a powerful model because of its simplicity. The model is the basis of the Six Sigma DMAIC (Define, Measure, Analyze, Improve, Control) model, the ADDIE (Analyze, Design, Develop, Implement, Evaluate) instructional design model, and many other improvement models from a variety of industries.

Numerous examples of applications of this model can be found in software. For example, consider a team who noticed that many of the bugs found by testers during the last milestone could have been found during code review.

1. First, the team plans a process around code reviews—perhaps requiring peer code review for all code changes. They also might perform some deeper analysis on the bugs and come up with an accurate measure of how many of the bugs found during the previous milestone could potentially have been found through code review.
2. The group then performs code reviews during the next milestone.
3. Over the course of the next milestone, the group monitors the relevant bug metrics.
4. Finally, they review the entire process, metrics, and results and determine whether they need to make any changes to improve the overall process.

Formal Process Improvement Systems at Microsoft

Process improvement programs are prevalent in the software industry. ISO 9000, Six Sigma, Capability Maturity Model Integrated (CMMI), Lean, and many other initiatives all exist to help organizations improve and meet new goals and objectives. The different programs all focus on process improvement, but details and implementation vary slightly. Table 3-3 briefly describes some of these programs.

TABLE 3-3 Formal Process Improvement Programs

Process	Concept
ISO 9000	A system focused on achieving customer satisfaction through satisfying quality requirements, monitoring processes, and achieving continuous improvement.
Six Sigma	Developed by Motorola. Uses statistical tools and the DMAIC (Define, Measure, Analyze, Implement, Control) process to measure and improves processes.
CMMI	Five-level maturity model focused on project management, software engineering, and process management practices. CMMI focuses on the organization rather than the project.
Lean	Focuses on eliminating waste (for example, defects, delay, and unnecessary work) from the engineering process.

Although Microsoft hasn't wholeheartedly adopted any of these programs for widespread use, process improvement (either formal or ad hoc) is still commonplace. Microsoft continues to take process improvement programs seriously and often will "test" programs to get a better understanding of how the process would work on Microsoft products. For example, Microsoft has piloted several projects over the past few years using approaches based on Six Sigma and Lean. The strategy in using these approaches to greatest advantage is to understand how best to achieve a balance between the desire for quick results and the rigor of Lean and Six Sigma.

Microsoft and ISO 9000

Companies that are ISO 9000 certified have proved to an auditor that their processes and their adherence to those processes are conformant to the ISO standards. This certification can give customers a sense of protection or confidence in knowing that quality processes were integral in the development of the product.

At Microsoft, we have seen customers ask about our conformance to ISO quality standards because generally they want to know if we uphold quality standards that adhere to the ISO expectations in the development of our products.

Our response to questions such as this is that our development process, the documentation of our steps along the way, the support our management team has for quality processes, and the institutionalization of our development process in documented and repeatable processes (as well as document results) are all elements of the core ISO standards and that, in most cases, we meet or exceed these.

This doesn't mean, of course, that Microsoft doesn't value ISO 9000, and neither does it mean that Microsoft will never have ISO 9000–certified products. What it does mean at the time of this writing is that in most cases we feel our processes and standards fit the needs of our engineers and customers as well as ISO 9000 would. Of course, that could change next week, too.

Shipping Software from the War Room

Whether it's the short product cycle of a Web service or the multiyear product cycle of Windows or Office, at some point, the software needs to ship and be available for customers to use. The decisions that must be made to determine whether a product is ready to release, as well as the decisions and analysis to ensure that the product is on the right track, occur in the *war room* or *ship room*. The war team meets throughout the product cycle and acts as a ship-quality oversight committee. As a name, "war team" has stuck for many years—the term describes what goes on in the meeting: "conflict between opposing forces or principles."

As the group making the day-to-day decisions for the product, the war team needs a holistic view of all components and systems in the entire product. Determining which bugs get fixed, which features get cut, which parts of the team need more resources, or whether to move the release date are all critical decisions with potentially serious repercussions that the war team is responsible for making.

Typically, the war team is made up of one representative (usually a manager) from each area of the product. If the representative is not able to attend, that person nominates someone from his or her team to attend instead so that consistent decision making and stakeholder buy-in can occur, especially for items considered plan-of-record for the project.

The frequency of war team meetings can vary from once a week during the earliest part of the ship cycle to daily, or even two or three times a day in the days leading up to ship day.

War, What Is It Good For?

The war room is the pulse of the product team. If the war team is effective, everyone on the team remains focused on accomplishing the right work and understands why and how decisions are made. If the war team is unorganized or inefficient, the pulse of the team is also weak—causing the myriad of problems that come with lack of direction and poor leadership.

Some considerations that lead to a successful war team and war room meetings are the following:

- Ensure that the right people are in the room. Missing representation is bad, but too many people can be just as bad.
- Don't try to solve every problem in the meeting. If an issue comes up that needs more investigation, assign it to someone for follow-up and move on.
- Clearly identify action items, owners, and due dates.
- Have clear issue tracking—and address issues consistently. Over time, people will anticipate the flow and be more prepared.

- Be clear about what you want. Most ship rooms are focused and crisp. Some want to be more collaborative. Make sure everyone is on the same page. If you want it short and sweet, don't let discussions go into design questions, and if it's more informal, don't try to cut people off.
- Focus on the facts rather than speculation. Words like "I think," "It might," "It could" are red flags. Status is like pregnancy—you either are or you aren't; there's no in between.
- Everyone's voice is important. A phrase heard in many war rooms is "Don't listen to the HiPPO"—where HiPPO is an acronym for highest-paid person's opinion.
- Set up exit criteria in advance at the beginning of the milestone, and hold to them. Set the expectation that quality goals are to be adhered to.
- One person runs the meeting and keeps it moving in an orderly manner.
- It's OK to have fun.

Defining the Release—Microspeak

Much of the terminology used in the room might confuse an observer in a ship team meeting. Random phrases and three-letter acronyms (TLAs) flow throughout the conversation. Some of the most commonly used terms include the following:

- **LKG** "Last Known Good" release that meets a specific quality bar. Typically, this is similar to self-host.
- **Self-host** A self-host build is one that is of sufficient quality to be used for day-to-day work. The Windows team, for example, uses internal prerelease versions of Windows throughout the product cycle.
- **Self-toast** This is a build that completely ruins, or "toasts," your ability to do day-to-day work. Also known as *self-hosed*.
- **Self-test** A build of the product that works well enough for most testing but has one or more blocking issues keeping it from reaching self-host status.
- **Visual freeze** Point or milestone in product development cycle when visual/UI changes are locked and will not change before release.
- **Debug/checked build** A build with a number of features that facilitate debugging and testing enabled.
- **Release/free build** A build optimized for release.
- **Alpha release** A very early release of a product to get preliminary feedback about the feature set and usability.
- **Beta release** A prerelease version of a product that is sent to customers and partners for evaluation and feedback.

Mandatory Practices

Microsoft executive management doesn't dictate how divisions, groups, or teams develop and test software. Teams are free to experiment, use tried-and-true techniques, or a combination of both. They are also free to create their own mandatory practices on the team or division level as the context dictates. Office, for example, has several criteria that every part of Office must satisfy to ship, but those same criteria might not make sense in a small team shipping a Web service. The freedom in development processes enables teams to innovate in product development and make their own choices. There are, however, a select few required practices and policies that every team at Microsoft must follow.

These mandatory requirements have little to do with the details of shipping software. The policies are about making sure that several critical steps are complete prior to shipping a product.

There are few mandatory engineering policies, but products that fail to adhere to these policies are not allowed to ship. Some examples of areas included in mandatory policies include planning for privacy issues, licenses for third-party components, geopolitical review, virus scanning, and security review.

Expected vs. Mandatory

Mandatory practices, if not done in a consistent and systematic way, create unacceptable risk to customers and Microsoft.

Expected practices are effective practices that every product group should use (unless there is a technical limitation). The biggest example of this is the use of static analysis tools. (See Chapter 11, "Non-functional Testing.") When we first developed C#, for example, we did not have static code analysis tools for that language. It wasn't long after the language shipped, however, before teams developed static analysis tools for C#.

One-Stop Shopping

Usually, one person on a product team is responsible for release management. Included in that person's duties is the task of making sure all of the mandatory obligations have been met. To ensure that everyone understands mandatory policies and applies them consistently, every policy, along with associated tools and detailed explanations, is located on a single internal Web portal so that Microsoft can keep the number of mandatory policies as low as possible and supply a consistent toolset for teams to satisfy the requirements with as little pain as possible.

Summary: Completing the Meal

Like creating a meal, there is much to consider when creating software—especially as the meal (or software) grows in size and complexity. Add to that the possibility of creating a menu for an entire week—or multiple releases of a software program—and the list of factors to consider can quickly grow enormous.

Considering *how* software is made can give great insights into what, where, and when the "ingredients" of software need to be added to the application soup that software engineering teams put together. A plan, recipe, or menu can help in many situations, but as Eisenhower said, "In preparing for battle I have always found that plans are useless, but planning is indispensable." The point to remember is that putting some effort into thinking through everything from the implementation details to the vision of the product can help achieve results. There isn't a *best* way to make software, but there are several *good* ways. The good teams I've worked with don't worry nearly as much about the actual process as they do about successfully executing whatever process they are using.

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z