HARVARD | BUSINESS | SCHOOL

9-607-084
REV: SEPTEMBER 10, 2008

ROBERT D. AUSTIN

CMM versus Agile: Methodology Wars in Software Development

Jennifer Fenton, CIO of Hardcore Financial Bank (HFB), settled back into a big, comfortable chair in the corner of her office, sipped her coffee and got ready to read a report that a trusted member of the IT staff had given her. The report, written by information technology (IT) thought-leader Ken Orr, had an intriguing title: "CMM versus Agile: Religious Wars and Software Development." The post-it stuck to the report's cover said it was as impartial a comparison of these two ways of developing software as she would find anywhere.

Some advocates of each approach fiercely contested the legitimacy of the other. Fenton had inadvertently touched off a firestorm when she announced her intention to explore whether HFB should adopt the "Capability Maturity Model" (CMM). She had gotten the idea at an industry conference where a competitor had spoken about his company's experience with it. The matter had not seemed controversial at the conference, and some people in her IT department seemed to like the idea. Then she discovered that the angriest opposition to the CMM came from some of her very best developers. Fenton heard harsh language from each side, which made it even more difficult for her to form an opinion. After she had expressed frustration that no one was able to give her the straight story, one of her team had suggested this report.

With HFB on a rapid growth trajectory and moving into new businesses all the time, Fenton had to get this decision right. Most of the company's various businesses depended heavily on IT, and corporate strategy called for developing best-in-class IT services so that clients could manage their own banking affairs. The IT group worked with many outside partners but developed plenty of software in-house, especially for customer interfaces. Her most talented developers knew a lot about software, naturally, but they also knew a lot about banking. Senior management considered the company's software systems a competitive advantage.

Nevertheless, members of the senior team felt that HFB's development processes *needed* to mature; they were still more ad hoc than suited a company with annual revenues now reaching into the hundreds of millions of dollars. Projects too often failed or ran over budget. So Fenton began to consider the CMM, a "maturity model" that marketed itself as best practice and had notable advocates in all sorts of businesses.

Professor Robert D. Austin prepared this case. HBS cases are developed solely as the basis for class discussion. The company mentioned in the case is fictional. Cases are not intended to serve as endorsements, sources of primary data, or illustrations of effective or ineffective management.

Two of her developers had said that they would quit if HFB adopted the CMM. According to them, the CMM was the old way of doing things and that the emerging best practice was "Agile." Fenton had heard of both, knew something about each, but she was far from expert in either. HFB had been so busy growing, and the IT department had been so busy building software, that control of development policy and methodology had remained informal and on the back burner.

Fenton thought she remembered taking classes in college that taught "Warnier-Orr" diagramming, and she was pretty sure that this report was by the same Orr. Ken Orr, a Fellow of the Cutter Consortium, a highly regarded information and analysis service, had written the report for Cutter's Agile Project Management & Software Development practice. The Cutter business model gave it better claim to impartiality than most similar services.[1] Fenton had high hopes that the report could help her figure out whether the CMM was a good idea for her firm.

Here's what she read.

The Cutter Report

NOTE: The contents of the following sections are adapted, with permission of the Cutter Consortium, from its report, "CMM versus Agile Development: Religious Wars and Software Development," by Ken Orr, Executive Report, Cutter Consortium, Vol. 3, No. 7, 2002. Orr's voice has been retained (with the use of "I") in what follows. Cutter holds all copyrights to the original report.

In 2001 proponents of radical software development methodologies met near Salt Lake City, Utah. After intense discussion, they issued "The Agile Manifesto,"[2] setting in motion a software schism between an old guard, supporters of the Software Engineering Institute's[3] "Capability Maturity Model" (CMM), and a newly organized "Agile" development movement. This report explains the relationships between Agile and the CMM, and how each relates to "rigorous," "light," and "heavy" approaches to systems development.

Software Methodology Wars

"Q: What's the difference between a terrorist and a methodologist? A: You can negotiate with a terrorist."
—Variously attributed, source unknown

Every decade or so development methodologies come into conflict. During the 70s, it was structured development vs. traditional development; in the 80s, data modeling driven vs. traditional development; and during the 90s, object-oriented design versus traditional development. In the first decade of the 21st century, supporters of "Agile methods" do battle with supporters of the most recent of the traditional "waterfall" software development approaches, the CMM.

[1] Cutter maintained impartiality by avoiding relationships with IT vendors. Many other analyst firms derived significant portions of their revenues from vendors, creating the possibility of conflict of interest when the analyst firm's advice had implications for vendor products or services. For more information on Cutter and its business model, see www.cutter.com.

[2] http://agilemanifesto.org/.

[3] The Software Engineering Institute is a research and development center funded by the U.S. government, "conducting software engineering research in acquisition, architecture and product lines, process improvement and performance measurement, security, and system interoperability and dependability." It is part of Carnegie Mellon University, which is located in Pittsburgh, PA.

In software wars, nobody gets killed or maimed. But for participants, the metaphor is real enough. Careers are made or destroyed, organizations thrive or perish in the struggle. The most interested bystanders are CIOs and development managers of large organizations. Just when they decide that "CMM Level 3" represents the current "best practice," someone (usually a young programmer) asserts that the CMM is out and something called "Agile Development" is in. What's a conscientious manager to think?

In addition to comparing Agile and CMM methods to see how they differ, and under what circumstances one approach might be preferable to the other, I'll cast what light I can on the hidden agendas animating the debate. One of the most important of these centers on control: Who has it? The programmer or the manager? The Agile "revolution" could also be labeled "the programmer's revenge." During the 80s and 90s a Computer Aided Software Engineering movement appeared intent upon eliminating, or at least deemphasizing, programmers and programming. Within that movement, programmers occupied a role analogous to assembly line workers in manufacturing: following rote routines laid out by others. Programmers found this condescending, to say the least. When management implemented these ideas, they forced programmers to do all kinds of bothersome, tedious chores. Object-Oriented (OO) techniques put development initiative back into the hands of programmers. Agile goes OO one further by turning even more control of development over to programmers and users, reducing the influence of many traditional management levers.

A related hidden agenda involves process: Should professional development follow a scripted sequence of steps, or should it remain flexible and allow the people doing it to exercise discretion? Many managers prefer the predictability of a scripted process; many programmers prefer a climate of flexibility that allows them to take initiative. The CMM vs. Agile debate is to some degree an argument about how people work best on complex problems.

Yet another hidden agenda has to do with documentation and design: Some programmers consider documentation and abstract design work that doesn't involve "writing code" a waste of their time. On other hand, others, many of them managers, like to see tangible deliverables that they can use to measure the progress emerging from development.

All of the items on this agenda list influence the debate. Partisans on both sides formulate "us" vs. "them" propositions—"We're on the side of the Angels, they're on the other side of . . . the other side." CMM proponents are in favor of "process improvement"; Agile developers are in favor of "increased user involvement" and "rapid development." Who could oppose any of these? But the issues aren't clear cut, and as the debate between the CMM and Agile matures, both sides adjust their positions and improve their methods.

Capability Maturity Model Background

The CMM originated as a means to address the difficulties of large-scale military software development. In 1984, as part of a long-term program to promote improved large-scale software development and software management, the United States Department of Defense (DoD), in conjunction with Carnegie Mellon University, chartered the Software Engineering Institute (SEI) to promote software development best practices and reuse. Over time, SEI became a major force in software research and education; the CMM has become one of the major programs it developed. The SEI has reached outside the defense community to promote software engineering in general and the CMM in particular.

In the 80s, Watts Humphrey, a researcher and manager from IBM, combined software lifecycles with maturity levels from the manufacturing quality movement into an approach to software development. His book became the basis for the development of the CMM.[4] In an article written in 1998, Humphrey explained the CMM's origins:

> . . . the U.S. Air Force asked the Software Engineering Institute (SEI) to devise an improved method to select software vendors. . . . A careful examination of failed projects shows that they often fail for nontechnical reasons. The most common problems concern poor scheduling and planning or uncontrolled requirements. Poorly run projects also often lose control of changes or fail to use even rudimentary quality processes.[5]

Throughout its history, the CMM has been associated with defense and military systems. The DoD is the largest long-term purchaser of software development services in the world; it wields immense influence. The DoD has been developing large-scale systems longer than anyone and, some would say, more successfully than almost any other organization. The DoD has also funded research in software development and software methods longer than any other organization, with the possible exception of the telecommunications industry.[6] DoD systems tend to be both huge and state-of-the-art, requiring vast investments in custom hardware and software. Moreover, the DoD uses outside developers to develop most of their systems. With so much at stake, the DoD has worked mightily to take as much risk as possible out of critical systems projects.

For decades, the DoD has promoted "rigorous systems methodologies" (RSMs). In the 50s and 60s, for example, the DoD promoted the first "Waterfall methodologies" adapted from work on hardware development (see **Exhibit 1**). The Waterfall emphasized the idea of serially phased development: planning, requirements, design, development/testing, and deployment. Each phase produced voluminous documentation to guide the next phase.

The Waterfall allowed careful consideration of problems and made sure that major elements were not missed. The serial approach also made it possible to break up the work so that, for example, one organization might do the planning and requirements while another could do the design, development/test and deployment. This modularity fit well with standard procurement practices on government projects, which called for using different vendors on different activities.

Because the DoD was such a big customer, many large military/aerospace firms adopted Waterfall methods very similar to the DoD's.[7] Those organizations with the most "rigorous" methods were often large, military software vendors. This pattern is being repeated today in the private sector. As more large organizations buy huge chunks of software from outside vendors, they are looking for better ways to manage this important relationship. The DoD has had so much experience in controlling outsourced software projects that commercial firms have taken a clue: they've begun to force the adoption of CMM practices among their own vendors.

One industry segment has embraced the CMM: software development outsourcing vendors, especially in India. For more than a decade now, large Indian software companies have been among the most aggressive in the world in pursuing higher and higher levels of CMM certification.[8] Indian firms feel they need CMM certification to allay fears customers might have about leaving software work to offshore firms. According to my colleague and friend Ed Yourdon, who sits on the Board of

[4] Humphrey, W., *Managing the Software Process*, Reading MA, Addison-Wesley, 1989.

[5] Humphrey, W., "Three Dimensions of Process Improvement (Part I: Process Maturity)," *Crosstalk*, February 1998.

[6] Most technologists will recognize the importance that DoD's Advanced Research Project Agency (ARPA) has had in the development of information technology. ARPA supported research in computers, software, database and, of course, the Internet. Other major technology research funding came from various military and intelligence services.

[7] The DoD has reinforced this by making systems development and systems management methods part of the standards that vendors must comply with to get DoD contracts.

[8] Roughly 40% of all CMM Level 4 and Level 5 certifications have been earned by Indian firms.

Directors of one of India's largest software vendors, these companies have become expert at following CMM best practices and have found ways of applying the CMM to small projects.

Many of the ideas behind the CMM are taken from statistical product quality control work pioneered by people like W. Edwards Deming and Joseph Juran. One of the principal ideas behind the CMM is that organizations can improve quality by gaining (statistical) control of their processes.[9] This idea is widely accepted within the quality movement worldwide; it is one of the guiding ideas behind other popular initiatives such as ISO 9000.[10] The idea of capability levels comes from the quality movement, as do definitions for the five maturity levels (initial, repeatable, defined, managed, and optimized).

In the complex world of software development outsourcing, the CMM has become a way of certifying expertise. In the same way that Oracle DBA and Cisco Internetworking Engineer certifications have become valuable on a resume, "CMM Level 3" certification has become valuable in marketing software development outsourcing services. Becoming Level 4 or 5 places one in rarified company as an elite software engineering firm.

CMM Levels

The CMM (see **Exhibit 2**) describes 5 levels:

- "Initial"
- "Repeatable"
- "Defined"
- "Managed"
- "Optimized"

CMM Level 1—"Initial"

Level 1 processes are ad hoc. There is no specific methodology; each project is a new activity. The "Initial" stage is the default in software management, the state attributed to a company that has achieved no higher level of maturity. "Unfortunately, the initial level processes are the most practiced processes in the software business."[11] Many Level 1 organizations don't admit they are Level 1 organizations. Like an alcoholic who can't be helped until he/she admits the problem, Level 1 organizations can't improve until management recognizes that they are in fact at Level 1. Recognition is sometimes forced when an organization wants to sell services to another organization that insists that its vendors have a CMM process in place.

[9] It is important to note that by processes, the CMM means mostly software management processes, like estimating, scheduling, quality control etc. The CMM has been faulted for its lack of interest in product development process, at which, as we shall see, Agile Development excels.

[10] One of the most controversial ideas behind the CMM is the notion that a software development process can be managed as closely as a manufacturing one. Indeed, a number of major thinkers consider the idea of quality capability maturity models to be fundamentally flawed.

[11] Raynus, J. *Software Process Improvement with CMM* (Norwood, MA: Artech House, 1999), p. 14.

CMM Level 2—"Repeatable"

Level 2 organizations have a basic project management structure in place to track costs, schedules, and some systems functions. Repeatable software processes in place are reinforced by training and by management controls. Because Level 2 organizations follow the same processes time after time, they can begin to work on measuring and improving those processes.

CMM Level 3—"Defined"

At Level 3, a software development organization has a well-defined software engineering approach operating in a project management framework. Well-documented processes are communicated across the organization through standards and training. "If the repeatable level defines what to do and who should do it, the defined level specifies when to do it and how to do it."[12] While Level 2 organizations are mostly concerned with project management processes, Level 3 organizations, theoretically at least, also define and measure their software development processes.

CMM Level 4—"Managed"

At Level 4, software development managers increasingly use statistical methods. They collect and manage process and product quality measures. At Level 4, software products are not only managed to meet time and cost schedules, but to meet specific customer satisfaction and defect levels as well.

CMM Level 5—"Optimizing"

The Level 5 organization has reached quality Nirvana. They produce software at Level 4, and never stop improving processes and standards. CMM Level 5 organizations are presumed to be world-class innovators in software development.

The CMM certification process examines organizational practices, determines their current maturity level, and provides management with a roadmap for improving that level. By committing to the CMM process, organizations can start wherever they happen to be and, through a conscious program, begin to improve along a trajectory much like that described by Total Quality Management gurus. It takes nothing more or less than commitment, training, measurement, and correction.

CMM Pluses and Minuses

The CMM appeals to organizations and managers interested in control and certification. Expert consultants certify an organization's CMM level. This is important in organizations in which CIOs and others look for outside creditability.

Surely there's nothing wrong with being "well managed." Software is not such a unique business that sound management practices don't apply. As the best-known, best-documented software management method in use in the real world, the CMM looks much like applied common sense. Tens of thousands of managers and programmers have been trained in its use and find the CMM a *lingua franca* for communicating with others. For managers seeking tangible improvement, those five CMM levels provide a way to direct attention to needs and to set objectives for meeting them.

On the other hand, people have noted that some organizations that train companies in CMM methods also provide certification, an apparent conflict of interest. Others have complained that gaining certification at Levels 2 and 3 seems to require good paperwork, not good management. Still

[12] Ibid., p. 17.

others ask how a CMM certification process can hope to succeed in assessing very complex software development practices. They point to the difficulty audit firms have in preventing poor financial reporting. In addition, critics aver that CMM certification has more to do with an organization's management processes (estimating, scheduling, control) than with the quality of the software produced.[13] There are even those who say that in order to become a Level 2 or 3 organization, you don't need a good (or state-of-the-art) software process in place, you need only blindly follow some process, any process.

One thing stands out: CMM Level 3 or 4 leads to a very strong (even heavy-handed) management style. By and large, these organizations pay more attention to project management than to product output. This can make software developers cranky; they'd rather program than fill out project management documentation. The CMM in the wrong hands may stifle innovation. Poorly implemented, the CMM becomes all the bad things that its opponents say about it

Agile Development Background

With some notable exceptions,[14] the people most responsible for the Agile movement came of age at the beginning of the "object revolution" in the late 80s and early 90s. That revolution was very successful; the most popular development languages and operating environments today all have an OO flavor. But not everything has gone as planned. Most disturbing, OO projects have been late and over budget as often as any other projects. As a consequence, some of the most devoted OO advocates came to recognize by the mid 90s that OO techniques alone would not revolutionize the software world. In order to really create new classes of software in very short periods of time, the software *process* needed a radical overhaul. People experimented with different approaches to OO development. Agile grew out of the convergence of a number of the most successful outcomes of OO development experiments.

The important themes of the early Agile movement included the following:

- Systems are best developed in small increments;
- Users and developers have to work hand-in-hand;
- Each system increment should be designed to handle the minimum requirements;
- When changes in requirements occur, they should be designed in; and,
- There should be little or no documentation beyond the actual code.

The visual image most associated with the CMM is a waterfall; the image of Agile is a rapid prototyping cycle (see **Exhibit 3**).

Most OO programming gurus who took up Agile techniques shared a dislike for paperwork, especially paperwork imposed by "rigorous" software approaches such as the CMM. In the circle of Agile gurus, there is a strong minimalist bias. Software itself is the product; everything else is overhead. "Why worry about documentation or project management forms, etc. if all that people want is the product?" Then: "What if we throw a few good programmers in a small room with the

[13] In response to the criticism about the CMM's lack of objectivity, the SEI has created a measurement program to support CMM certification. This program is called Software Engineering Measurement and Analysis (SEMA).

[14] A very notable exception: Jim Highsmith, the author of several influential books on Agile development, especially *Adaptive Software Development*, is a pioneer in systems methodologies going back to the 70s. Bob Charette, Steve Mellor, and Ken Schwaber are other greybeards that have joined forces with younger Agilists.

user and let them develop the software a little at a time. That could eliminate waste and miscommunication." Far from being imposed by some large organization like the DoD, Agile developed spontaneously in the work of a number of independent consultants.

One of the most influential gurus in the Agile movement is Kent Beck. He's written many books and articles on eXtreme Programming (XP), and has lectured around the world. After receiving his MS in computer programming from the University of Oregon, Beck worked for a number of organizations including Apple and HP. At Chrysler in the mid-90s he began to formulate his ideas for a radical new way to develop systems. Throughout the late 90s these ideas began to catch on. Now, into the 21st century, an increasing number of people have taken up XP or one of the other, similar "fast-track" methodologies.

Software developers and managers had a tough time sorting out the differences and similarities among fast-track methodologies. Finally, in early 2001, proponents of the new ideas met in Utah to see if they could come up with a set of common principles. Out of that meeting came the now famous *Agile Manifesto*. The Manifesto proposed a simple and straightforward set of principles (see **Exhibit 4**). These threw down a gauntlet, an unmistakable challenge. The answer? Open warfare between the newly united rebel band and the establishment—at least the part supporting the CMM.

The Agile Approach

In contrast to the CMM, which is a set of loosely connected software management best practices, Agile is a set of highly integrated development and management practices.[15] In his book, *Extreme Programming Explained*,[16] Kent Beck proposed 12 ideas that together constituted an Agile approach[17]:

1. The Planning Game

- Small Releases
- Metaphor
- Simple Design
- Testing
- Refactoring
- Pair Programming
- Collective Ownership
- Continuous Integration
- 40 Hour-Week

[15] I've used eXtreme Programming here as a representative of Agile because it is the best known and best documented. There are a number of other Agile methodologies that have signed on to the Agile Manifesto. Some of the better known are SCRUM, DSDM, FDD, and Crystal Light. All of these methods share most of the basic tenets with XP, especially the high level of user involvement and the importance of rapid prototyping.

[16] Beck, Kent, *Extreme Programming Explained: Embrace Change*, Second Edition (Boston: Addison-Wesley, 2004).

[17] Throughout this section I use XP and Agile Development interchangeably since XP is the best known and best documented of all the Agile approaches.

- On-site Customer
- Coding Standards

The Planning Game

For Beck, "software development is always an evolving dialog between the possible and the desirable."[18] Neither customers (business users) nor technical staff should have complete control. Users should decide about the scope, priorities, timing, and functionality of product releases. Technical people should decide about estimates, technical consequences, the development process, and detailed scheduling. In successful XP projects, the planning game becomes a give-and-take process of true collaboration.

Small Releases

"Every release should be as small as possible, containing the most valuable business requirements."[19] As we say in my organization, the Ken Orr Institute, a "doable project" is one that is small enough to be done (quickly) and big enough to be interesting (to the business, customer, developer, etc.).

Metaphor

Every project should have a single "overarching" metaphor. Metaphor replaces the term "architecture" within a project; a good one unifies the project, keeps team members focused and greatly improves the work. "As development proceeds, and the metaphor matures, the whole team can find new inspiration by re-examining the metaphor."[20] Of all the ideas that make up XP, metaphor may be the hardest one to grasp. Some have described it as the "essence" or the "irreducible core" of whatever it is your project is building.

Simple Design

From Beck's standpoint, the right design for software at any given time is one that:

1. Successfully completes all the tests.
2. Has no duplicated logic.
3. States every intention important to programmers.
4. Has the fewest possible classes and methods [i.e., technical structures that manifest functionality].[21]

Agile developers strive to achieve this minimalist design standard.

Testing

"Top XP teams practice 'test-driven development,' working in very short cycles: add a test, make it work. Almost effortlessly, teams produce code with nearly 100 percent test coverage . . . a great step

[18] Beck, K. *Extreme Programming Explained* (Reading, MA: Addison-Wesley2000), p. 55.
[19] Ibid., p. 56.
[20] Ibid., p. 56.
[21] Ibid., p. 56.

forward in most shops."[22] Test-driven development means that Agile developers define tests before they develop code. It's easy to overlook the importance of this idea. By doing this, programmers work out the inputs and outputs they want to achieve; then they build the code that produces the correct results. Like many Agile techniques, these tests replace heavy-handed management controls; they produce good programming discipline and good code.

Refactoring

Refactoring involves the constant redesign of the entire OO class hierarchy and methods as new requirements emerge from the work process. Agile programs evolve through iteration. At each release, new requirements make themselves known. Each new set of requirements requires that the programming team go back and design them into the existing code. Refactoring goes hand in hand with the idea of minimal requirements. One of the basic tenants of XP and Agile: implement only those requirements known for sure and essential to the current release.

After a number of iterations programs can become increasingly difficult to maintain. Adding change after change to a program makes it harder and harder to understand. Refactoring prompts a revisit to the basic design at each iteration. This means that changes must be integrated into the design, making the system's evolution smoother. Refactoring means iterative design/redesign.

Early OO designers spent enormous amounts of time designing class hierarchies and methods before they could build an application using them. But this overloaded early OO on the front end. Designers spent weeks or even months working on the initial classes, only to discover that they always had to redo them as the project progressed and they learned more about the business problem they were trying to solve. In OO, this became known as "class thrashing."

Agile features minimal design up front, but repeats design at each iteration. Done right, this doesn't degrade the program, but improves it over time. There are some difficulties with this approach. For one thing, manually refactoring OO code isn't simple or easy; for another, some kinds of refactoring (data structure, for instance) are much more difficult than others.

Pair Programming

In XP, pairs of programmers work so closely together that one can pick up or modify the work of the other at any time. Developers who work this way maintain that pair programmers produce much more over the long haul than individual programmers working independently. They maintain that programmers working in tandem must make each part of the code clear and comprehensible. Moreover, pairs working together learn better habits and produce more code than they could on their own. Some managers, to whom it seems obvious that individual programming would have double the productivity of pair programming, can't accept this counterintuitive claim.

Pair programming institutionalizes code reviews. Each day, each pair programmer must understand all of the pair's code no matter who wrote it. Agile gurus maintain that such pairs produce better, tighter, more understandable code. Used in conjunction with "test first," pair programming speeds a development process by forcing both programmers to use a common design strategy. Pair programming creates an environment in which people know more because they are involved in more. It can be seen as a form of knowledge management. By making sure that for every function there are two people who completely understand what is going on, managers get free of depending on some key individual who might suddenly become unavailable.

[22] Jeffries, R. "What is Extreme Programming?" XProgramming.com.

Collective Ownership

"In XP, everybody takes responsibility for the whole system."[23] In Agile, *the group* owns the central product of the organization, the software. Instead of individual programs being the responsibility of one individual, from concept through maintenance, every one in the group takes responsibility for all the code. Anyone has access to any piece of code; anyone can, within the limits implied by pair programming, change anything.

Continuous Integration

Because of its emphasis on speed, Agile strives for constant integration. There are no long, deliberately planned cycles between building system prototypes. Instead, the code base evolves, generating new prototypes frequently. The code base is maintained in a buildable state at all times. If something fails during a prototype build, the team that has made the most recent changes tracks down and fixes the problem.

The 40-hour Week

Agile gurus preach that software development is more marathon than sprint. In knowledge-based work, team members must be as fresh as possible. While there is not much that software development managers can do to help team members avoid stress and burnout at home, they can do plenty on the job. The rule, then: People on the Agile development cycle work no more than 40 hours a week. While there may be occasions for overtime, they should be few and far between.

The On-site Customer

"A real customer must sit with the team, available to answer questions, resolve disputes, and set small-scale priorities. By 'real customer' I mean someone who will really use the system when it is in production."[24] In Agile development, people who will use the system take responsibility for how that system will work; they commit to the period of the entire project. Total user commitment characterizes an Agile project. Users are as responsible for project definition, user interface design, review of prototypes, and test case presentation as anyone else on the team. They are, in fact, full-fledged members of the team. This greatly reduces finger pointing when a product is delivered.

Coding Standards

Pair programming, collective ownership, and minimal documentation, make it critical that code follows very strict guidelines. In this environment, "you simply can't afford to have different coding practices. With a little practice, it should be impossible to say who on the team wrote that code."[25]

Comments on Agile

Where the CMM is often referred to as a "heavy" or "rigorous" method, in fact individual contributions in Agile are much more closely scrutinized. Agile doesn't rely on managers to insist on proscribed behavior. The proscriptions are part of its methodology, which thus requires a high order of competence among the team members.

[23] Ibid., p. 59.
[24] Ibid., p. 60.
[25] Ibid., p. 61.

The Agile approach derives from certain beliefs about group dynamics and human psychology. People work better in groups, so we program in teams; code review is good, so we always have pair programming; output-oriented design is good, so we incorporate "test-first development"; people work better with enough rest, so we institute the 40-hour work rule.

In the typical CMM environment, organizations can organize any way they wish, use any method they like, as long as they commit to doing the same things the same ways, measuring what they do, and continually improving. Agile is a much more tightly integrated package. Agile gurus argue that you can't decide to implement items 1, 4, 6, and 9 of the list and leave out the rest. You need to commit to the entire program if you're going to be successful—each of the 12 items is there for a reason and ignoring any one item out will jeopardize the entire program.

In CMM development, managers assume most of the responsibilities. In Agile, the team takes responsibility. Teams set priorities, set schedules, supervise design, testing, and delivery. The product rolls out in small increments, and individuals are not accountable except as part of a team.

Agile Pluses and Minuses

"Deliver quickly. Change quickly. Change often. These three driving forces compel us to rethink traditional software engineering practices."

—Jim Highsmith

To some experienced managers and developers, Agile looks like a sophisticated justification for unstructured hacking. They find particularly problematic the Agile inclination to get quickly started writing software, incorporating only the requirements that are known at the time. They'd rather spend more time on understanding customer requirements before beginning to write code. In the opinion of some, Agile avoids proper discipline in problem definition and design.

The Agile defense is that there is no easy place to stop if you worry about future (unknown) requirements. You're guessing about those future requirements and you might guess wrong. Better to focus on what you know and know how to change as you learn more.

Build things in, don't add things on: that's how to make a clean, elegant design in an Agile environment. In practice this grows more difficult as the program and pressures to finish grow.

On an Agile project, the end product is always the clear focus—not the requirements, not the design, but actual working code. Agile institutionalizes incremental product release, which lets users respond to something they actually experience. This leads to less abstract discussions than traditional requirements definition involved. Agile integrates many conventional best practices into a closely woven fabric of basic principles that all work together. Agile, because of this integration in the system's design, is greater than the sum its parts.

Nevertheless, Agile presents hurdles for some organizations that would like to adopt it. Most of the Agile gurus they might consult naturally assume that OO is the basic practice, so to learn Agile they may have to learn OO first. Some organizations don't know what to do when the team, not the individual, is the center; hard-line managers insist on individual accountability. Then there is the issue of full-time participation by the end user. Some organizations don't get the essential participation: the client can't or won't provide it. Agile gurus may shrug and say, "Well that's tough; no full-time user, no product or system," but the issue of user involvement can be a deal breaker if you just can't get it. A related concern: Most large systems have many users, not just one. If you have "customers" (rather than "a customer") they are likely to have different viewpoints, different needs. Working with only one may not take you in the right direction to suit others.

Developing the Agile way is not even every programmer's favorite cup of tea. Many young people go into programming because they like to work alone, just me and the machine. Agile calls for a degree of sociability.

For most organizations, Agile presents more of a stretch than the CMM. While the CMM may seem like overkill at times, at least the band will recognize the music and instruments. With Agile, we're talking about new music, new instruments, and no conductor—a jazz ensemble instead of a concert band.

Concluding Thoughts

There's a lot of propaganda in software methodology wars. For Agile enthusiasts, Agile is a lean-and-mean, minimalist, fast-track approach and anything else, especially the CMM, is heavy handed, bureaucratic, and doomed to failure. CMM proponents point to the fact that some of the best software organizations in the world use the CMM on their most important projects. To them, Agile is for small (inconsequential) projects; if you want to do important (big, mission-critical) projects, you had best adopt the CMM.

There is probably truth in both points of view. The CMM produces vast quantities of paper. Implementing it is very expensive, but probably less costly than repeated failure on high stakes projects. Small boats are, as a rule, faster and more maneuverable than big ships, but you can't transport millions of gallons of oil or thousands of passengers on a small boat. The approaches that work on small projects may not scale up.

Agile advocates suggest that theirs is a management movement as much as it a development strategy. If you're careful as you plan your incremental releases, they maintain, you can do big things with fewer people in less time. The keys to Agile are collaboration and commitment.

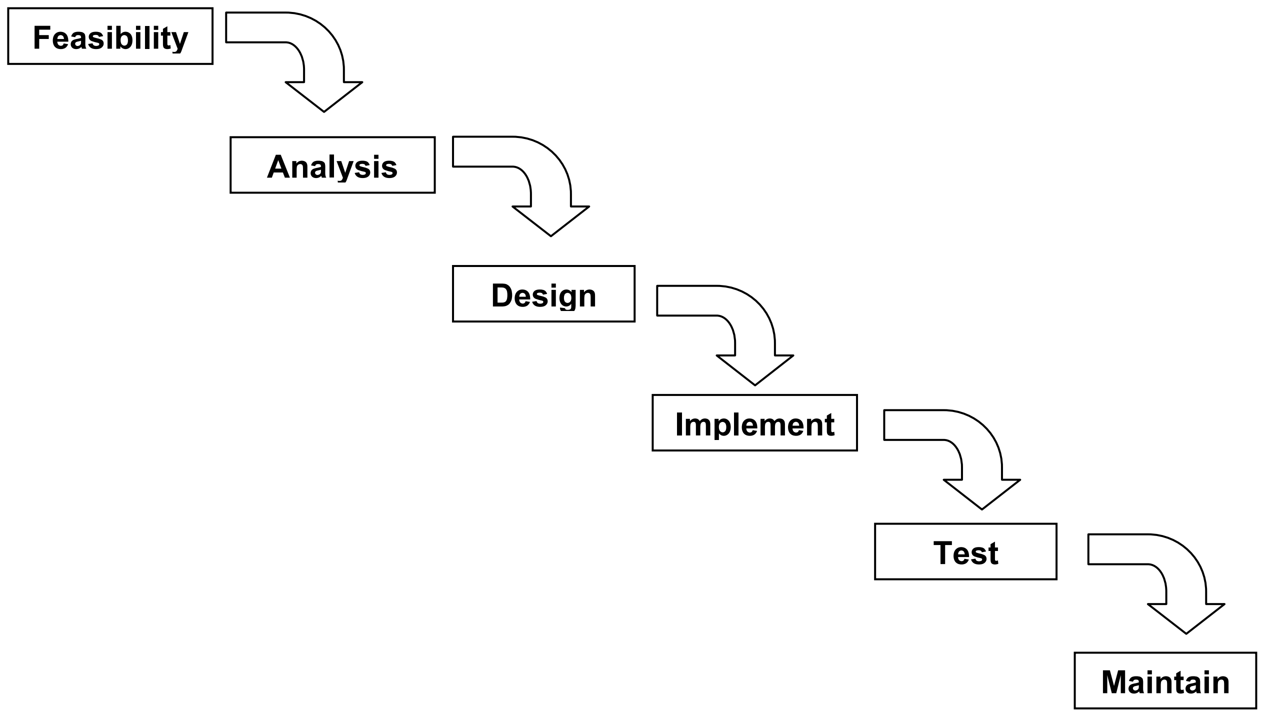
So who are you to believe? Well, as with most things, the answer lies somewhere in the middle. Depending on your context and business, you might have more luck with one or the other, or with each one applied in different contexts. The war between the CMM and Agile gives us a chance to make both approaches better.

The Decision

Fenton closed the report and took another sip of coffee. The report had helped, but Ken Orr could not choose for her. Many of the systems developed by HFB these days were large. Failure was expensive. But the company also needed to react to rapidly changing customer requirements and to pounce on opportunities as they appeared.

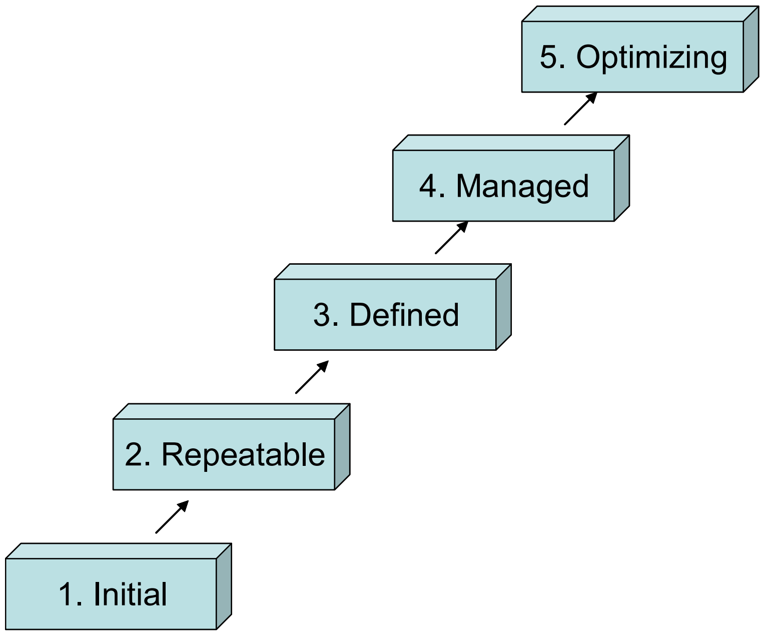
She resolved to learn more by talking to her peers and developers. The casual way she had introduced the idea of adopting the CMM had clearly been a mistake. She would have to make up for that on the way to an eventual decision, whatever she decided to do . . .

Exhibit 1 The Software Development "Waterfall"



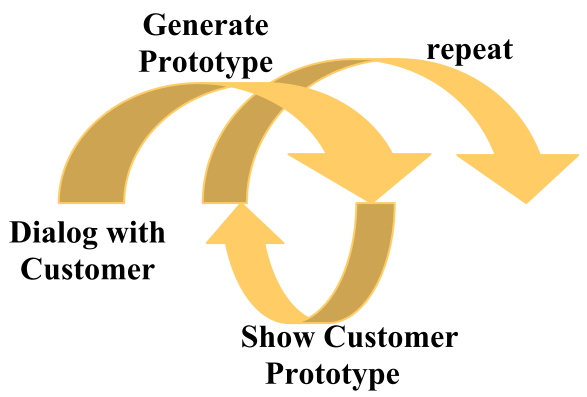
Source: Authors.

Exhibit 2 The CMM Levels



Source: Cutter Consortium.

Exhibit 3 Agile's Iterative Development Cycle



Source: Authors.

Exhibit 4 The Agile Manifesto

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over *processes and tools*
Working software over *comprehensive documentation*
Customer collaboration over *contract negotiation*
Responding to change over *following a plan*

That is, while there is value in the items on
the right, we value the items on the left more.

Source: Adapted from http://agilemanifesto.org/.