Chapter 7

The Essence of Agile

With Ryan Martens[1]

Conceptually, agile is simple. Most everything is different.

What Are We Changing with Agile?

We've now reviewed a variety of agile methods and an iterative and incremental one that can be applied in a substantially agile fashion. As we begin to analyze them for commonality, we will find many common practices among them, and these common practices, plus a few extensions, form the basis for Parts II and III of this book.

And yet, when we compare the methods in aggregate to our former plan-based, stage-gated, and waterfall-like processes, we find far more differences than similarities. Indeed, it would not be too extreme to say that when it comes to software development and software project management in general, *agile changes everything*, as Figure 7–1 shows.

These changing paradigms provide both the power and the consternation of agile, because addressing change on such a wholesale basis in an enterprise is not a trivial thing. And yet, team by team, organizations are making these changes over time, allowing them to approach the full benefits of agile development. Let's examine each of these new paradigms to see what additional clues we can find as to what is *so* different about agile.

1. Ryan Martens, agile thought leader and founder and president of Rally Software Development Corp., contributed much of the conceptual content of this chapter.

Process	Waterfall Development	Iterative and Incremental	Agile Development
Measure of Success	Conformance to plan	→	Response to change, working code
Management Culture	Command and control	→	Leadership/ collaborative
Requirements and Design	Big and up front	→	Continuous/emergent/ just-in-time
Coding and Implementation	Code all features in parallel/test later	→	Code and unit test, deliver serially
Test and Quality Assurance	Big, planned/ test late	→	Continuous/concurrent/ test early
Planning and Scheduling	PERT/detailed/fix scope, estimate time and resource	→	Two-level plan/fix date, estimate scope

Figure 7–1 Changing paradigms in agile

New Measures of Success

The basic measures of success are different in agile. Teams and organizations evolve from *conformance to plan*[2] to the *ability to respond to change.*

Measure of Success	Conformance to plan	Response to change, working code
	Work breakdown structure	Feature breakdown
	Single, detailed, complete plan	Two-level plan
	Serial functions	Parallel functions
	Follow the plan	Adapt to changing facts
	Procedural stage gates	Time boxes, inspections
	Documents models, reviews	Working code

2. Agile thought leader, Jim Highsmith, notes that he has seen so many good teams trying to understand why they were not on plan, and faulting themselves, that he eventually came to the conclusion that the *plan*, and not the *team*, was the problem.

This transition involves moving from traditional work breakdown structures to a "value-delivery focus" by implementing stories and requirements on a prioritized basis. Procedural and documentation stage gates are replaced with success measures based on working, tested, and demonstrated code. The *plan* is fluid and flexible; the *actual* is the best that can be achieved under the facts present at the time. More importantly, the *actual* is potentially *shippable*.

Different Management Culture

In many ways, agile turns the traditional approach to software management upside down.

Management Culture	Command and control	Leadership/ collaborative
	Management defines dates and scope	Teams bid stories
	Management dictates implementation	Team selects approach
	Culture of sign-offs	Shared learning
	Protect the scope	Protect the date
	Demonstrate at end	Demonstrate always
	Weekly status meetings	Daily stand-up meeting

Traditionally, management fixed scope, dates, and resources and set the technical direction for the team. Management was also responsible for the team's performance. In agile, the table is turned. Management sets direction; the teams bid the work and figure out how to accomplish as much of the work as possible in the time frame given. The team self-organizes as necessary to meet the objectives. The team makes the technical decisions and corrects them on the fly as necessary.

Management's job is to eliminate impediments within the organization and trust the team to meet the objectives (this trust is reinforced daily with visibility of progress and the presence of working, integrated code). In turn, the team is fully accountable for the deliverable and is responsible for meeting the dates and delivering the requisite quality. Team empowerment and team accountability are two sides of the same agile coin.

Different Approach to Requirements, Architecture, and Design

Our strategies for how to approach requirements, architecture, and design evolve as well.

Requirements and Design	Big and up front	Continuous/emergent/ just-in-time
	Marketing requirements up front	Vision and backlog
	Software specification up front	Just-in-time elaboration
	Models and plans	Build in increments
	Big design up front	LRM[3] design decisions
	Architecture is planned	Architecture emerges

Instead of investing months in building detailed software requirements specifications, architectural models, and even prototypes, teams focus on delivering early, value-added stories into an integrated baseline. Early delivery serves to test the requirements and architectural assumptions, and it drives risk out by proving or disproving assumptions about integration of features and components. If it doesn't work, the team refactors the code until it does, allowing for constant user feedback and visibility along the way.

No longer do management and the user community wait breathlessly for months, hoping that the team is building the right thing. At worst, the next checkpoint is only a week or so away, and with a little luck and foresight, users may be able to deploy or at least evaluate even the earliest iterations in their own working environment.

Revised Coding and Implementation Practices

Coding is different too. Instead of the developers working on all the functionality in parallel with a big bang at the end, the whole team "swarms" over the earliest and highest priorities first.

3. Last Responsible Moment

Coding and Implementation	Code all features in parallel/test later	Code and unit test, deliver serially
	Build in parallel	Build serially
	Integrate late	Integrate continuously
	Hand off to test	Partner with test
	Demonstrate at end	Demonstrate always
	Individual code responsibility	Shared code ownership
	Never miss dev. complete date	Never break the build
	Test code later	Code unit test first

Integration is continuous. Testing is not deferred; it is done first (XP or TDD) or concurrently with the development of the code. Pairing is routine. Conversation is continuous. There is only one kind of code that results: tested, working, integrated code. Feedback is immediate and continuous. All team members know where they are every day and what they need to do that day to meet the goals of the iteration.

Changes to Test and Quality Assurance Practices

The testing and QA organizations are in for big changes as well.

Test and Quality Assurance	Big, planned/ test late	Continuous/concurrent/ test fast
	Contract with customer	Partner with customer
	Big test plan sign off	LRM testing decisions
	Testing at the end	Test from the beginning
	QA is responsibility for test	Everyone is responsible
	Testers write all the tests	Everyone writes tests
	Testing squeezed	Low features squeezed
	Big stand-alone test teams	Integrated with dev
	Automate tests later	Automate tests now

The impact on the testing organization is substantial. Often, entire QA and test organizations are refactored (largely disbanded as a separate organization) and instead dispatched to become part of individual component or feature teams. Testing is no longer a lifecycle phase; it is a continuous activity. No longer do testers test large blocks of untested code; rather, they tests *systems* that include new code that has already been unit- and acceptance-tested. Development of

testing automation is the rule rather than the exception. Testing skill levels increase as testers participate in design decisions and test automation development. Programmers' skills increase as they understand how to write code that is straightforward enough to be tested. QA personnel do real QA instead of managing legions of manual testers.

New Ways of Planning and Scheduling

Planning and scheduling change too.

Planning and Scheduling	PERT/detailed/fix scope, estimate time and resource	Two-level plan/fix date, estimate scope
	Detailed planning early	Detailed planning JIT
	Measures on intermediate deliverables	Measures based on code
	Protect the scope	Protect the date
	Demonstrate at end	Demonstrate always
	Weekly status meetings	Daily stand-up meeting

But contrary to rumor, planning does not disappear in agile; indeed, it is quite intense and it reappears at two levels: gross-level plans for releases and fine-grained plans for iterations. Planning doesn't happen just once, and upfront planning happens at every release and every iteration boundary. Planning is no longer lumpy and ad hoc—it is systematic and routine.

Planning is greatly simplified because the dates are always known in advance, and the teams, with the product owner driving, are responsible for determining priorities. Tracking is simpler too, because daily status meetings and frequent demonstrations illustrate progress. No longer is there a separation between plan and actual. Managers don't worry about interdependent events, such as who has a vacation this week—teams do.

The Biggest Change: Scope versus Schedule—Schedule Wins!

As we learned in DSDM, perhaps the biggest change of all is that, in the battle of date versus scope, *the date always wins.* That is, iteration length (or periodic release date at the release level) is determining scope instead of scope determining the length of a development cycle. In plan-driven methods, scope determined time, and two variables (scope and time) varied with every planning cycle and every significant change. Since agile methods fix the time and let that define scope, only one variable remains (the scope of what gets built). This frees the team to organize as necessary and to remain constantly focused

on what can be accomplished by the date. And since the scope is always prioritized, team members can be assured that they will deliver the best possible solution in the time available, as the DSDM pyramid in Figure 7–2 illustrates.

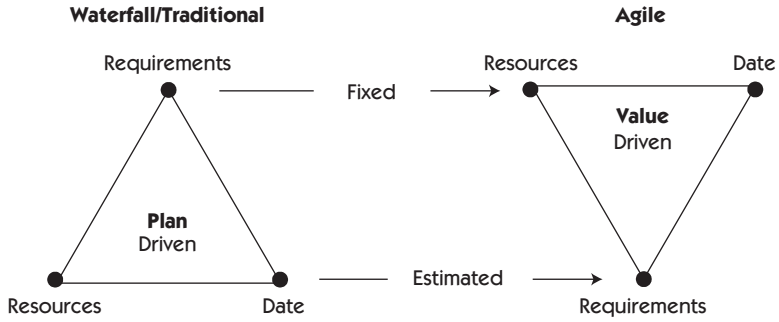

Figure 7–2 Plan-driven (traditional) versus value-driven (agile) methods

If for some reason, the delivered result lacks sufficient functionality to be "above the bar" (which can only be determined by the users when they have a system to evaluate), have no fear because the next iteration is only a week away, and the next release will be available only a month or two thereafter.

THE HEARTBEAT OF AGILE: WORKING CODE IN A SHORT TIME BOX

So much for the differences in plan-based and agile development methods! It's time now to move toward what is *common* among the methods themselves so we can start to distill common practices to apply at scale.

As we do so, we discover that all agile methods have one thing *big* thing in common, and that may well be the quintessential difference between agile methods and the waterfall. That is, *all development proceeds by creating small chunks of working code in a time box* (fixed date.) This new skill is the heartbeat of agile, and when teams master this skill, many other things agile will fall naturally into place. Figure 7–3 illustrates this simple "process model within a process model."

Figure 7–3 illustrates that stories are pulled from a backlog in order of the "next most important thing to do for the customer." Each item is defined/built/tested in a fast, concurrent loop. We use define/build/test as a verb to

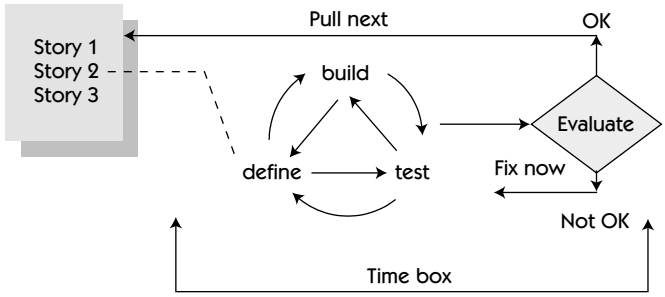

Figure 7–3 The heartbeat of agile, working code in a time box

illustrate that this operation is atomic; one part can't be done without the other.

Within the time box, each item is evaluated for acceptance, and when it passes the test, another story is pulled from the backlog. If it fails, it is reworked, *on the spot*, until it passes the test. Of course, in order to accomplish this, all the resources necessary to define it, build it, and test it must be continuously present on the team (Chapter 9).

Even then, there are two key elements to this simple process statement, each of which represents a paradigm shift for the new agile team.

1. Working in a Time Box

In a well-disciplined agile process, everything is time-boxed.

> *The mechanism for handling flexibility of requirements in DSDM is the time box.*
>
> —DSDM [2006]

> *This time-boxing establishes a rhythm for the development organization that becomes the drum beat that synchronizes the activities of all participants. Like a manufacturing schedule, it is repetitive, predicable, and reliable, and all software production and delivery rotates around this cycle. The most important benefit is that time-boxing introduces a near-term milestone that forces both the team and code lines to converge and actually deliver working software at regular intervals.*
>
> —Leffingwell and Muirhead [2004]

Iterations Are Time-Boxed Iterations are timeboxed, and they may be as short as one week or, more typically, two weeks, but rarely longer. (Learning to accomplish this art is the subject of Chapter 10.)

Releases Are Time-Boxed Releases are time-boxed too. Release dates are known in advance, and scope, not quality or resources or schedule, is adjusted to assure the team meets the commitment of the date.

Meetings Are Time-Boxed Meetings are also time-boxed. Release planning, iteration planning, iteration retrospectives, iteration demos, and daily standups are all time-boxed, thereby instilling discipline in the team and constantly communicating that *time matters*. This discipline also helps every team member meet his or her commitments by knowing in advance how much time can be committed to any specific overhead activity.

> *When we started with agile, I was concerned it might be a less disciplined method for development. In reality, it's more disciplined and provides more accountability.*
>
> —Paul Beavers, Director, BMC R&D

2. Developing in Small, Bite-Sized Chunks

> *The feature (story) should be small enough that it can be done in a few days.*
>
> —Poppendieck and Poppendieck [2003]

> *I like to break stories into tasks that individuals take responsibility for and estimate. I think ownership of tasks goes a long way towards satisfying the human need for ownership.*
>
> —Beck [2005]

Agile, in general, and XP, in particular, takes incrementalism to the extreme. Work is divided into stories and tasks. Stories represent "pliable" requirements (or at least the need for a discussion about a requirement); tasks are added to define the specific work that each team member must do to implement the story. The granularity of stories typically represents no more than a few days' work. The granularity of tasks is on the order of less than a day. For those applying use cases, significant use cases may be implemented as granularly as one scenario at a time or by picking any other logical and cohesive piece (such as a single step) of the use case that can be done in an iteration. There are a variety of reasons for doing this, and each represents a key element of agile.

Small Chunks Can Be Estimated and Tracked The size of the chunks of work that are scheduled into the iteration has a dramatic effect on the visibility of their status. If one large chunk of work is scheduled into an iteration, then by implication, the entire iteration will be devoted to working toward finishing that one chunk, and the chunk of work can't possibly be completed until the end. Gauging progress is a matter of guessing the "percentage complete" of the monolithic chunk. It is also difficult to get the testing group involved early in the iteration when all of the work is delivered in a very large chunk at the end.

If we instead break the monolithic chunk of work into smaller pieces, we can then consider the status of the smaller pieces individually and we no longer have to wait until the end of the iteration to see a chunk of work transition from a state of "in progress" to "complete." Instead, all along the way, smaller chunks of work are transitioning from "planned" to "in progress" to "complete" states as developers take responsibility for, work on, and then deliver each small chunk.

This breakdown gives us a more fine-grained means to track the progress of the iteration. We suddenly can understand whether any particular chunk of work is blocked, ready to be tested, completed ahead of schedule, or perhaps unlikely to be worked on at all. We also have the ability to deliver the smaller pieces to the testing group *during* the iteration.

Small Chunks Foster Ownership and Accountability Working in small chunks fosters ownership and accountability on the part of the team. After, all if chunk A is to be delivered in less than a week, the team had better be very certain of which team members are going to define/build/test it!

Small Chunks Divide Big Jobs into Doable, Smaller Pieces On systems of scale, complexity reigns. Imagine the challenge of coordinating 300 people to build a new system, the likes of which have never been seen on the market. In traditional development models, the problem can appear overwhelming, and no amount of planning can possibly flesh out all the details. In addition, for very large jobs, job satisfaction for the team is often deferred until the end. And as we have seen with our analysis of what *doesn't* work, even the ending can be pretty unpleasant at times. In such a case, job satisfaction can be entirely missing from the employment equation!

With agile, progress and job satisfaction are constant, frequent, and in real time. The next opportunity to show your wares to a customer, peer, or other stakeholder is at most a week or so away.

Small Chunks Uncover Key Risks Early It is not unusual for teams new to agile to achieve only 25 to 35 percent of the scope of work they set out for themselves in the first iteration. The reasons can be varied, as we'll see in later chapters. But whether the goal is missed because, for example, the teams are unable to estimate very well or perhaps because a small story uncovered an unanticipated technical obstacle, the results of the first iteration will immediately expose the risks, wherever they may lay. In addition, *management has its first tangible view of progress at the end of only the first week or two of the project.*

SUMMARY

We've seen in this chapter that, conceptually, agile is indeed very different from our plan-driven methods. However, given this data, we can now also start to imagine how these different paradigms can potentially deliver dramatically different results. That is why we are here, and that is the power of agile.