

SAMPLE CHAPTER

Agile ALM



Lightweight tools and

agile strategies



Agile ALM

by Michael Hüttermann

Chapter 2

Copyright 2012 Manning Publications

brief contents

PART 1	INTRODUCTION TO AGILE ALM.....	1
	1 ■ Getting started with Agile ALM	3
	2 ■ ALM and Agile strategies	34
PART 2	FUNCTIONAL AGILE ALM	59
	3 ■ Using Scrum for release management	61
	4 ■ Task-based development	92
PART 3	INTEGRATION AND RELEASE MANAGEMENT	117
	5 ■ Integration and release management	119
	6 ■ Creating a productive development environment	170
	7 ■ Advanced CI tools and recipes	191
PART 4	OUTSIDE-IN AND BARRIER-FREE DEVELOPMENT	247
	8 ■ Requirements and test management	249
	9 ■ Collaborative and barrier-free development with Groovy and Scala	297

ALM and Agile strategies

This chapter covers

- Agile strategies within the context of an ALM
- The Agile ALM approach that we'll implement throughout the rest of the book
- A discussion of the process pitfall

Everyone's doing it Agile today. Few will admit that they're not working in an agile way in its classic sense. But what about Agile software development? It comes in many varieties. Agile is a value system that emphasizes important aspects of software development, including communication and open, respectful collaboration, allowing errors and failures to be treated as valuable learning experiences. Agile promotes a safe-to-fail environment, where you can fail quickly and learn from your mistakes. Everybody makes mistakes; most of them are a result of initiatives based on poorly understood facts. Blaming people for mistakes eliminates their motivation to pursue innovation and leads to "management by fear." People who are afraid of expressing ideas won't ask questions and will act defensively.

Having motivated people is essential to further development; without them, your project will be stuck and will likely be a failure. Therefore, management must be careful to lead their teams by example, through coaching, and by improving basic work conditions. Management by numbers, management by objectives, or

management by walking around doesn't work. Instead, prefer management through conversation, socializing, and leadership. Teamwork and leadership are much better when they promote education and improvements, provide a learning process (in which the team members lose nothing by admitting to weakness), and recognize the contributions of all people. "Everyone on the team has some unique contribution."¹ Enable people to have pride of workmanship. When there are issues, leadership shouldn't judge; rather, it should investigate possible causes and offer assistance during the daily work.²

Agile focuses on honest, open communication, including a pragmatic view of requirements that have proven to be effective. Management needs to create an environment that is conducive to productive and efficient work. Management by destructive criticism won't work!

A DEFINITION OF AGILE "You accept input from reality and you respond to it."
—Kent Beck

Clear communication is essential in any project. Agile helps to facilitate clear communication and honest and open assessment of everything, from understanding user requirements to testing software.

Agile is a term applied to different process models. On the one hand, there are pure Agile models, such as Scrum, Extreme Programming, and DSDM (www.dsdm.org).³ On the other hand, there are models that can implement an Agile approach, like the Rational Unified Process. Finally, Agile is also used to describe strategies that incorporate an Agile approach. Figure 2.1 illustrates these three dimensions as features of the Agile ecosystem.

In this book, I don't dictate any specific process model for developing software. You can use rich, full-fledged models like waterfall, spiral, or a hybrid type. The point is that you can enrich your current processes where it makes sense for you by using

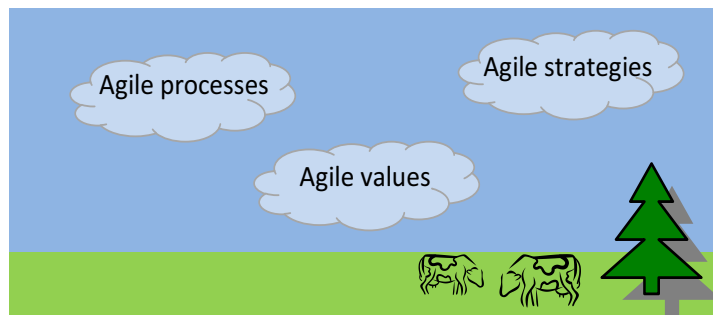


Figure 2.1 The Agile ecosystem: Agile processes, values, and strategies

¹ Gerald M. Weinberg, *Quality Software Management*, vol. 3 (Dorset House, 1994), p. 265.

² In *Out of the Crisis* (MIT Press, 1982), W. Edwards Deming lists 14 points for management that, in his opinion, lead to improved quality, increased productivity, and better team morale.

³ For a detailed discussion of different Agile approaches, see Jim Highsmith, *Agile Software Development Ecosystem* (Addison-Wesley, 2002).

Agile strategies. Furthermore, these strategies aren't dependent on any specific tools (though I will recommend some). Think of Agile as a value system, a set of strategies, and a toolbox.

The waterfall

In 1970, Winston W. Royce published an article in which he described the “waterfall model.” In the scope of software development, people often think about the waterfall model as a sequential chaining of phases, including design, implementation, and maintenance. They contrast it to the Agile approach, claiming they're completely different, but this isn't the case. Royce claimed that this static, sequential phase ordering wouldn't work. He recommended iterations, but this is often ignored when talking about the waterfall model.

What can we learn from that? First, don't believe everything others say. Second, don't think Agile is reinventing the world. And third, please don't think Agile is chaotic programming where we all do what we want without documentation or orderly control of changes.

In this chapter, we'll gain more insight into Agile strategies. Chapter 3 will introduce an implementation guide for Scrum, an Agile management discipline for functional releasing. This management framework is abstract enough that you can adapt it for many different, individual process flavors. We'll implement Scrum and make it concrete. Using the tools discussed later in this book, you'll implement those strategies.

2.1 The Agile and project management

Project management in an Agile environment requires an approach that can handle the frequent changes that exist in iterative development. Agile project management provides transparency and traceability in an environment that thrives on constant change. Everything the team does must add value to the system. The process must be open to changes and be highly organized and disciplined. Continuous cooperation and awareness are indispensable for responsibility, permanent reflection, and synchronization.

The Agile project management process is based on the magic barrel (also known as the magic square; see figure 2.2) and its four pitchers: quality, time, resources/costs, and scope (functionality and number of features). The total volume is limited, which means that features, resources, and time are also limited; in practice, you must save for quality. Agile projects invest much more in quality than traditional projects do.⁴ Their high investment in quality is a fixed constant. Their investments in time (mainly in a time-boxed project where the times are fixed) and resources⁵ are fixed constants as well, but the scope is variable.

⁴ “The typical steps we take to deliver a product in less time result in lower quality.” Tom DeMarco and Timothy Lister, *Peopleware*, 2nd ed. (Dorset House, 1999), p. 137.

⁵ “Adding manpower to a late software project makes it later,” Brooks's law, in Frederick P. Brooks, *The Mythical Man-Month* (Addison-Wesley, 1995), p. 25.

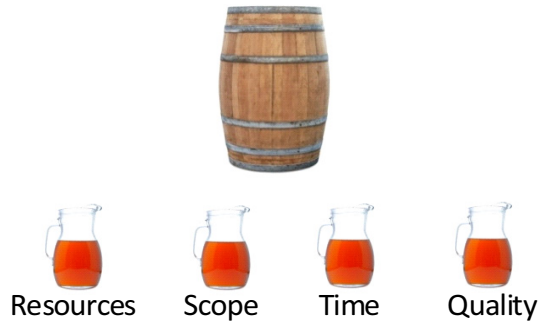


Figure 2.2 The magic barrel. The limited content of the barrel is allocated among four pitchers. If one pitcher is full, then another pitcher will have less liquid. In Agile projects, the quality pitcher is filled to some reasonable amount, with the rest being balanced among the three other pitchers. (Illustrations ©iStockphoto.com/dja65, ©iStockphoto.com/thebroker.)

Software produced in an Agile way is designed, created, tested, and delivered in continuous iterations. An *iteration* is a collection of one to many increments of executable software. An iteration must follow a strict, well-defined process with the required discipline for all participants. The Agile approach is also based upon the fact that customers often don't know all their requirements up front. If customers don't have a complete set of requirements up front, developers often don't have all the information they need to write the code, or even to estimate the amount of time needed to create the application.

Increments and iterations

Increments and iterations are basic concepts in Agile projects:

- An *iteration* is a mini-project that may result in an increment of the software. Iterating starts with an idea of what is wanted, and the code is refined to get the desired result.
- An *increment* is a small unit of functionality. Incrementing allows you to build a better understanding of what you need, assembling the software piece by piece.

Although using both in parallel delivers the best results, this isn't required. You can increment more effectively when requirements are more stable (or you want them to be more stable) or better understood. Iterating allows a better response to changing or unclear requirements.

The Scrum Agile methodology uses both iterations and increments. Each iteration delivers a fully functional increment—a set of shippable features delivered to the end user. Although increments and iterations are highlighted by the Agile ecosystem, many other more traditional process models include these two basic approaches. Both increments and iterations have been around for years. In 1971, one year after Royce talked about iterations in his waterfall model, Harlan Mills from IBM wrote about the concept of incremental development in *Debugging Techniques in Large Systems*.

The Agile approach differs from the traditional approach of developing software. The Agile approach uses Agile values, processes, and strategies, all of which are sometimes viewed from the traditional perspective as an excuse for programmers to abandon

project design and management. Table 2.1 lists some common practices used in Agile projects and briefly describes the Agile approach. The third column identifies common misunderstandings from the traditional viewpoint.

Table 2.1 Common Agile practices and associated misunderstandings

Practice	Agile approach	From the traditional perspective
Software development	Treats software development as an information process.	Software development is a manufacturing process.
Communication	Encourages and requires continuous interaction and feedback; the whole team is collocated.	Project members focus on their individual tasks first and often rely on documents more than on communication.
Courage	Encourages an open atmosphere.	There's a fear of missed deadlines and misunderstandings with customers.
Collective ownership	Specifies that program code and documents are owned and maintained by the team.	People feel responsible for only their piece of work.
Integration	Uses continuous integration to get early feedback and increase quality.	Integrations are rare, late, and felt to be a waste of time.
Test-driven development	Treats testing as of great value for design, code, and quality.	Tests are considered a waste of time. Many tests are done manually.
Customer involvement	Encourages customer participation.	The customer is often seen as the contracted party.
Refactoring	Accepts temporary suboptimal, pragmatic design; design is maintained and improved continuously.	Errors aren't allowed; created artifacts are supposed to run perfectly at once.
No overtime, sustainable pace	Follows regular working schedules that can be sustained over time.	Regular overtime is necessary to deliver on time while planning aggressively.
Iterations	Slices software into handy and convenient iterations.	No iterations are necessary; the work focuses on a single release, mostly a big bang release.
Stand-up meeting	Institutes daily structured exchanges.	Big, long, infrequent project meetings are used. The allocation of people and amount of time are often excessive.
Documentation	Uses documentation only where necessary, and when it adds value.	Documentation is considered an important artifact, written according to standards. In reality, it's seldom read.
Team	Treats the team as important, as a collection of individuals having their own strengths and characteristics. The team should be cross-functional.	The individual expert is in focus. Work is done in isolated islands of knowledge.

Table 2.1 Common Agile practices and associated misunderstandings (continued)

Practice	Agile approach	From the traditional perspective
Standards	Uses standards, where necessary, that are understood and agreed on by the team.	The work involves a strict process, with many heavyweight standards, often for the sake of having standards.
Quality	Is inherent in everything that the team does.	Quality is the first goal to be skipped when time and money get short.
Change	Considers change as a normal part of project work.	Change is more condemned than encouraged.

All projects address these practices in one way or the other.

The “Manifesto for Agile Software Development” (Agile Manifesto, at agilemanifesto.org) lists four value pairs, where one value is more important than the other.

- *Individuals and interactions over processes and tools*

General approach—Projects and software development involve human beings. Software is developed by and for people; IT is a means to an end. The necessity of processes and tools is accented because a methodical process and the use of tools are essential, but individuals and interactions are even more important.

ALM approach—Often people are frustrated when using a comprehensive release or even a lifecycle management approach, because the usage is typically combined with process formalities that are too complex and overloaded. Frequently, tools that provoke a rigid process are used, which isn’t necessary. ALM processes and tools should support the work and not vice versa. Knowing and understanding the process will drive the need for a tool and its usage. The ALM infrastructure ought to be lightweight and support interactions. But while individuals and interactions are the primary consideration, processes and tools are also important.

- *Working software over comprehensive documentation*

General approach—Although documentation of the software is important, a working program is more vital to the customer. Clients can’t run their business by creating documentation; they must be able to run a working software system. Software doesn’t lie; you can document something that is missing in the software, but you can’t simulate executable software functionality just by documenting it. Documentation has to be created in those areas where it adds additional value. The software is self-contained, describing the system and keeping it maintainable for the future (which is often the task of good documentation).

ALM approach—ALM automates the creation of software artifacts (and configurations). It can be based on “executable knowledge” instead of traditional documents. Rather than documenting processes, the processes are converted to an

executable infrastructure. At any time, and starting early, executable software has the priority.

- *Customer collaboration over contract negotiation*

General approach—Relying on contracts in business is standard. But customers and software developers are sitting in the same boat: Both parties want an optimal solution. Intensive collaboration, continuous exchanges, and active customer involvement are essential during the development.

ALM approach—ALM can organize expectations and divert the communication and interactions of all stakeholders into coordinated communication and process channels. A targeted process and effective tooling make the project's status visible at any time. ALM delivers synchronization points and serves as a communication vehicle.

- *Responding to change over following a plan*

General approach—It's important to deliver milestones at regular intervals. A fixed, rigid plan can suggest there's a certain security where none exists. There's often a mismatch between an officially described process and the real one used every day. Changing basic conditions and new insights are part of daily business in software development, and the process must be open to change. Change management is implicit in Agile projects.

ALM approach—ALM makes change easier. ALM methods and tools facilitate development in a vital, enduring way. Synchronization points support identifying basic conditions as they change.

Agile values impact the entire application lifecycle and transform the way an organization operates in many important ways. In the next section, I'll give some examples of how to implement Agile strategies.

2.2 *Agile strategies*

Here, I'll describe a set of basic Agile strategies that demonstrate standard approaches to implementing ALM processes. The strategies don't say anything about tooling; they describe concepts and best practices. The strategies will be detailed later in the book and implemented with specific tools.

2.2.1 *Version control and a single coding stream*

First, it's important to store your artifacts in a version-control system (VCS), but which types of artifacts you store there depends on the project context and the requirements:

- Coding artifacts should be stored in the VCS. Although this sounds obvious, it's not always the case. Many projects, for example, patch their applications in production without having those changes under source control. In addition to source code, tests and build scripts need to be versioned.
- I recommend that you externalize (and version) your runtime configuration settings. It's best practice to control all variable configuration values externally

from the application so they can be changed easily without recompiling the application.

- It's wise to place documents, such as use cases (written in Word, for example), into a version-control repository so that you can benefit from versioning and accessing change history.

Although common VCS tools like CVS or Subversion weren't invented to run as file servers, it's possible to store binary artifacts, such as Word documents, in them. Subversion is better equipped for this task than CVS, because it handles binaries more efficiently. This avoids the ugliness of storing documents on a central, shared file structure, which are then replaced randomly, with no history tracking or traceability. Using a VCS for documents is vastly superior to another common mechanism for sharing information: that of sending your documents by email and not having a central place to hold them. Unfortunately, this practice is often the norm.

Additionally, you should set up one central repository to store your assets, to avoid having multiple places where documentation might be located (for example, VCS, a file sharing server, and Notes in parallel).

When you check your artifacts into VCS, you'll have to decide how to arrange the files in the system and decide who works on which stream and why. The rule of thumb here is that you shouldn't open any additional streams for longer than necessary. If you're branching in your VCS, you should close the branch as soon as possible. But consider all developer check-ins and integration processes on one developing stream as synchronization points. This is also valid for distributed version-control systems like Git.

Keeping branches in use for too long is a bad practice.⁶ The longer you wait to incorporate changes from one codeline to another, the more effort it takes to merge those lines and the more error-prone the effort will be. To facilitate progress, it's important to integrate code streams frequently. You should have one stream (for instance, a head or trunk) as the leading stream in your development.

There's no one-size-fits-all solution, though. Using branches is a good way to prepare releases (and to incorporate bug fixes on this stream after releasing). In an Agile context, you may only put a label (a tag) on a special source version (a baseline) while continuing to develop on the main code stream (the head/trunk). Then, if you find a bug that must be fixed in a release already promoted to production, you can create a *bugfix* branch, based on the existing tag or based on a given revision number (with Subversion). You can create these branches when needed. This makes the most sense when your velocity and release frequency are high.

In some environments, branches are often used as feature branches (for developing features or whole products or variants) or developer branches (with developers working on their own streams). In such environments, it can be obligatory to use several streams in parallel.

⁶ See Kent Beck, *Extreme Programming Explained*, 2nd ed. (Addison-Wesley, 2005), p. 67.

2.2.2 Productive workspaces

Although frequent integration is essential to rapid coding, developers need control over how they integrate changes into their workspaces so they can work in the most productive way. Avoiding (or delaying) the integration of changes into a workspace means that the developer can complete a unit of work without having to deal with an unexpected problem such as a surprise compilation error. This is known as *working in isolation*.

Developers should always verify that their changes don't break the integration build by updating their sandbox with the most recent changes (from others) and then performing a private build prior to committing changes back to the VCS. Private workspaces enable developers to test their changes before sharing them with the team. The private build provides a quick and convenient way to see if your latest changes could impact other team members.

These practices lead to highly productive development environments. If the quality of the checked-in code is poor (for example, if there are failed tests or compilation errors), other developers will suffer when they include these changes to their workspaces and then see compilation or runtime errors. Getting broken code from the VCS costs everyone time, because developers have to wait for changes or help a colleague fix the broken build, and then waste more time getting the latest clean code. This also means that all developers should stop checking in code to VCS until the broken build is fixed. Avoiding broken code is key to avoiding poor quality.

Developers test their isolated changes, and then, if they pass the tests, check them into the VCS. But an efficient flow is only possible when the local build and test times are minimal. If the gap between making code changes and getting the new test results is more than 20 to 30 seconds, the flow is interrupted. If the tests aren't run frequently enough, the quality decreases. Decreased quality, in turn, means that broken builds aren't fixed immediately, and this becomes a vicious circle.

You can optimize test roundtrips by categorizing tests. Run smoke tests and unit tests in your private workspace, and then run comprehensive integration tests on a dedicated machine. It's important to automate the build process and provide a quick and easy procedure for building the code as a complete baseline. Do this by creating a build in the local sandbox or calling a dedicated build engine for a private build (perhaps using the same platform designated for the official production build).

BUGS There are always bugs in the system. This is normal. Tests can't guarantee the correctness of an application or ensure that it's bug-free. Tests can only find single bugs, and good testing should come as close as possible to finding all of them.

Having a local environment means the developer has their own local resources, such as a (local) server, or that their own database (or database scheme) enables testing. The local environment doesn't need to be physically on their desktop; it can also be on a central server (like an individual database scheme), but it must be reserved for

the developer's individual use. A one-time investment in database schemes and similar tools increases productivity and application quality.

Furthermore, it's important to aim for a congruent build by comparing the developer and the integration views. Although the developer is working with a private build (which may use dummies and mocks), the build system itself should be identical to the central integration build. This helps avoid the "but it works for me!" syndrome and dramatically improves the start-up time for new peers. Most important, bugs can be identified and fixed quickly. The longer it takes to find them, the more expensive they're to fix when they're found.

This doesn't necessarily mean that the build on the developer's desktop (distinguished from the build system) will be identical to the one on the central build system. Because you want quick feedback, not all tests may run on the desktop—perhaps only the smoke tests. Or you may use mocks to simulate subsystems or components on the desktop but run full tests without mocks on a central integration machine. It's important to have a fast feedback loop on the developer's desktop, and if it isn't fast enough, the developer runs the tests too infrequently (or skips tests altogether) and the quality decreases. That can become a self-defeating process. Chapter 6 describes concepts and tools you can use to set up productive workspaces.

2.2.3 Continuous integration

Continuous integration (CI) includes code integrations that are run at least on a daily basis. The word *continuous*, as used in this context, denotes a repeatable process that occurs regularly and frequently. The word *integration* means that individually developed pieces of code that are stored in a source code repository are checked out as a whole; then they're compiled, packaged, tested, inspected, and deployed with build results integrated into web pages, or sent out as an email, or both.

Continuous integration

"Continuous integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily—leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly."—Martin Fowler^a

"The value of continuous integration is to reduce risks, reduce repetitive manual processes, generate deployable software at any time and at any place, enable better project visibility and establish greater confidence in the software product from the development team."—Paul M. Duvall et al.^b

a. Martin Fowler, "Continuous Integration," <http://martinfowler.com/articles/continuousIntegration.html>.

b. Paul M. Duvall et al., *Continuous Integration* (Addison-Wesley, 2007), pg. 29.

There are numerous reasons why CI is effective. The more often the code is integrated, the faster it's to figure out exactly what caused a bug. That's because there are far fewer sources of errors to examine, debug, and resolve. For example, it's quicker (and much easier) to find the cause of a bug if there are only two changes to examine since the last integration rather than 50. In addition, it's easier to eliminate bugs (by first identifying the root cause) while the change is still fresh in the developer's mind, and they can remember exactly what they did, why they did it, and how they chose to implement it. Another reason CI is effective is that when multiple changes are integrated together, their combined impact can result in unpredictable bugs or, even worse, serious bugs that then get delivered to the customer. CI not only offers the preceding advantages, it's also "essential for scaling lean and agile development," as discussed by Craig Larman and Bas Vodde.⁷

The relationship between the effort of integration and the amount of time between integrations is exponential: As the number of days between integrations increases, the effort required to fix bugs skyrockets exponentially. Waiting another day to perform an integration build doesn't translate to merely one more day of effort, but often to several more (see figure 2.3).

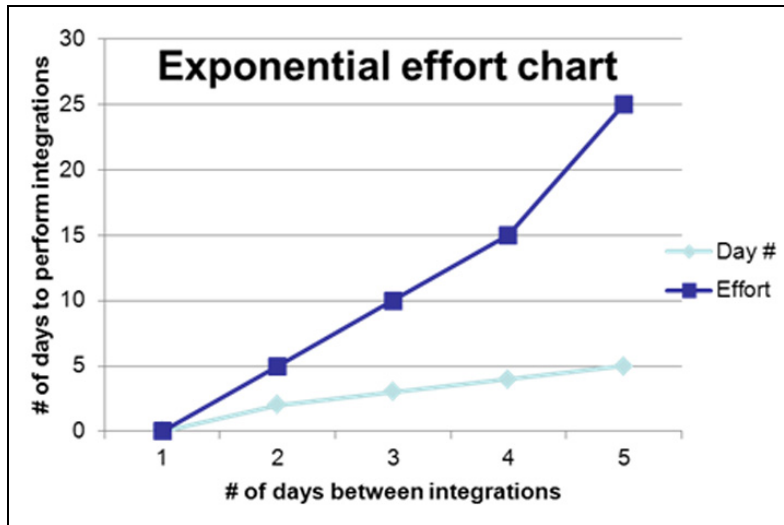


Figure 2.3 The integration effort spent on repairing errors increases exponentially.⁸

⁷ Craig Larman and Bas Vodde, *Practices for Scaling Lean & Agile Development* (Addison Wesley, 2010), chapter 10.

⁸ This illustration isn't based on an empirical study. Data is based on experience of the author and others.

CONTINUOUS INTEGRATION IN A NUTSHELL

Although project setups may differ slightly among teams or projects, the basic process of CI usually stays the same: A CI server captures all information needed to run a build and provides a general build environment, including the Java Virtual Machine (JVM), related runtime properties, and so on. The CI server executes a job definition to trigger project builds, which are reproducible based on a project's build management system. The build scripts should also be versioned and are separated from any IDEs; the CI server will continuously execute the build scripts, such as every time code is committed to the VCS.

Building and integrating software as soon as a developer checks in their changes is called *continuous build*. An *integration build* is the build that is created by a central build server. An integration build can be more complex than local builds that are triggered in developers' workspaces because it integrates more systems and runs more tests. In some environments, all that is needed is a nightly build. CI ensures that a given revision of code in development will build as intended or fail (break the build) if errors occur. The CI build acts as a "single point of truth," so builds can be used with confidence for testing or as a production candidate. An integration build may consume artifacts from a component repository and create artifacts and publish them to the component repository. CI should include all artifact types (configuration items), including coding artifacts, build scripts, database scripts, test cases, and so on.

Whether the build fails or succeeds, the CI makes its results available to the team. Reporting (dashboarding) ensures that the developers are notified about broken builds. The developer may receive the information by email, RSS notification, instant messaging (IM), IDE integration, or any other suitable notification mechanism. A build history provides an archive of former builds and their results. Reporting also contains generated documentation. With each build, documentation can also be created (such as API documentation, project site, and release notes) that reflects the recent state. Manually created documentation may be necessary (for instance, a user guide) but it should be minimized. Continuously updating manual documentation is error-prone and time-consuming. Worse, if the documentation doesn't correspond to the software, it quickly becomes useless.

You can implement "build-staging" according to your corporate standards. For example, validating special quality requirements on higher build stages prevents code from being promoted before it's ready. The initial staging area is the developers' workspaces.

After one experience with a development process incorporating CI, many developers tend to agree that it can potentially add value to their processes. Various positive effects can be triggered by incorporating CI into the development process. Each revision of the project source code that passes the CI build can be considered a release candidate, or at least a demonstrable product state, because the integration has proven to be successful. The team members may have reduced turnaround times, because they aren't required to do full builds and testing all the time. They can concentrate on verifying (unit) tests for the specific pieces of work that they're implementing in a

task-based way, while the CI systems take the long-running builds and tests for the whole project. Most modern IDEs provide features that can help the developer with partial builds and isolated in-place tests.

Reduced turnaround times and the certainty of completing a working task increase the developers' sense of accomplishment as well as their productivity and motivation. This helps to keep the whole team focused on the feature tasks.

Before CI, most software projects had a dedicated integration phase. Integration was a painful part of the project, where developers did nothing but make their code work with that of their colleagues. Can you see what's wrong with this picture? First, the developers might think of code they wrote as their own. But the real owner of the code is the person who pays for it. Second, the developers could spend months doing nothing but merging many code branches together, delivering no value to the customer. That's staggering: stopping a project to make all of your developers' work compile. Continuous integration can eradicate this problem. By considering a project to be green if the code compiles and passes tests, or red if either or both fail, it's easy to make "keeping the build green" the norm.

Humans focus on improving their craft and not merely on improving their tools. Tools come in handy, but they can also come at the cost of maintaining one's discipline. It's all too easy to install a CI tool and then say that you're "doing" continuous integration. You're doing continuous integration if the team makes it a high priority to fix the build frequently and if your developers want to add their builds to your CI server. You are probably not doing continuous integration without some form of testing strategy (although you may start rolling out basic CI by automatically compiling and packaging software on a central machine). Proving that your code compiles isn't enough of a safety net for your developers.

AUTOMATIC TESTING AND INSPECTIONS

CI installs testing—particularly automatic testing—as part of the development process. Testing isn't a downstream activity; rather, the team's development and testing activities are integrated. In his book *Succeeding with Agile*, Mike Cohn says that testing at the end doesn't work. For him, it's hard to improve the quality of an existing product with testing at the end. Mistakes continue unnoticed, the state of the project is difficult to gauge, feedback opportunities are lost, and testing is more likely to be cut.⁹

Tests can be divided into functional tests and technical tests (unit, module, and component tests). Those test categories should be linked and reported on together. Unit tests make sure that modules, functions, and methods behave as they should. Functional tests ensure you've developed the right feature. Unless you can surround pesky regression bugs with tests, you'll have no confidence that the code is working and the required features are implemented.

Before you fully implement CI, your teams must possess the discipline to maintain tests and guarantee that the project's build is working and up-to-date. Automating

⁹ Mike Cohn, *Succeeding with Agile* (Addison-Wesley, 2010), pp. 308–310.

tests is the prerequisite to entering short development cycles, getting early feedback, and creating high-quality software. Testing must begin on the developers' workstations and lead into technical integration tests and to functional system tests. Pre-check-in tests should be applied before changes are checked into the version control. If a test case fails, that's an opportunity to build a first *quality gate*, which is a defined, special milestone during development, where special quality requirements are met. You can also think about using smoke tests or sanity checks to run basic operations on the application. They can check whether the application can be started or if the main functions are addressable.

If you detect a failure condition, an Agile approach is to write a new test that validates whether these errors do occur. It's interesting to measure the test coverage and to monitor the results. You can also introduce a quality gate that fails automatically when the coverage isn't good enough. It can be helpful to establish a build threshold that tolerates defects to some degree.

Metrics and audits should also be measured continuously. Some people claim that cycle time is the only valuable metric. *Cycle time* is the total time from the beginning to the end of your process; for example, from the definition of the scope of the release to the delivery of the software.

A continuous inspection analyzes the design and code and points to quality defects. This avoids long, manual review sessions. A prerequisite for a continuous inspection process is to have an open atmosphere and a transparent and collaborative project setting.

It's important to understand that quality doesn't come from inspection but from improving the process. Defects cost twofold: Somebody makes them and gets paid, and then another person (or even the same one) is paid to repair the defects. Measuring something doesn't improve the thing you've measured. Merely inspecting something won't build quality: "You cannot inspect quality into a product."¹⁰ If suboptimal quality is the norm, this can only be addressed by improving the process.

Software craftsmanship

Driving an Agile ALM isn't enough for successful software delivery. Agile ALM is the backbone of your software development. The quality of the product does depend heavily on the quality of the software and the job the developers do. Software craftsmanship emphasizes the coding skills of the developers. For details, see *Clean Code: A Handbook of Agile Software Craftsmanship* by Robert C. Martin.

CONTINUOUS DELIVERY AND DEPLOYMENT

CI also includes continuous delivery and deployment. *Continuous delivery*¹¹ is an essential part of release management that addresses the "last mile" to provide the runnable

¹⁰ W. Edwards Deming, *Out of the Crisis* (MIT Press, 1982), p. 29.

¹¹ See J. Humble and D. Farley, *Continuous Delivery* (Addison-Wesley, 2011).

(functional) release to the user. A change *may* result in a release and *may* start the whole release process. But no changes are applied to the production system without a process; most of the changes will be staged along the full staging path. The process and the infrastructure must enable the team to promote every change to production, if you want to do that.

Often there's a mismatch between how technical people perceive what constitutes "available" and the perception of the final customer—the user of the application. Software is only available for use when it's installed and distributed; it's not available if it's only packaged or deployed but can't be used. The software is available for use only if it's deployed on a system, the users have permission to work on the application, the database is available, and so on.

Specifications or versions of software in the developer's workspace can't replace versions of software available on dedicated test environments. Late binding "big bang" integrations are a bad idea—it's always best to release and deliver early and often. This can only be achieved by automating the delivery process. The more frequently the software is built and integrated, the more the process of delivery becomes a routine job (a repeatable process). If building and integrating doesn't become a daily routine, the people involved become worried about quality and delivery becomes more infrequent.

CI is a virtuous cycle. Once you've invested in the initial setup, your CI system will grow with your software hand in hand. Integration becomes a routine job, not a painful dedicated activity that's postponed downstream. The easier your CI is to use and the better optimized the CI process is, the more it will be used, the more it will be optimized, and the more it will be accepted. If you institute only heavyweight downstream integration, you'll experience a downward spiral; project members will probably have reservations against CI and will fail to integrate as much as necessary. Frequent deployments provide direct, timely feedback loops—a requirement for continuous improvement.

It's important that the technical target environment (the system to which you deploy) be similar, or ideally identical, to the final production system. This will help you detect errors quickly. Deployment should always be automated and should be part of the continuous development process. Preferably you'll have one unified deployment script for deploying the application to different target environments. A precondition for this is to decouple deployment and configuration. The deployment scripts should be self-testing to automatically verify their own output. The deployment should be started easily by executing one script (instead of numerous manual steps). Setting up a centralized machine to deploy your application to multiple target environments can further accelerate productivity.

CONWAY'S LAW

Why does CI (and ALM as a whole) sometimes have acceptance issues in companies? One explanation may be found in Conway's law: "Organizations which design systems . . . are constrained to produce designs which are copies of the communication

structures of these organizations.”¹² For example, consider a software application that has three development teams—developers, operations/deployment engineers, and QA. The system will likely have three subcomponents (or subsystems). Using the premise of Conway’s law, the coupling between these three components and the quality of the interfaces between them can be predicted by the quality of the communication between the three teams. What implications does this have for CI? At least in bigger companies or projects, setting up integration and continuous integration requires strong cross-functional communication, synchronization, and uniformed solutions. Integration also means integrating across different organization borders and roles; for instance, development, deployment, and testing.

The software passes through the hands of these three distinct teams, and all these teams have individual problems and concerns. In the worst-case scenario, these concerns can lead to “empire-building,” where a team attempts to acquire resources (more money, more employees) in order to increase its influence outside its areas and expand its size and power. Besides that, teams can have competing objectives. Consider the example of an operation crew that gets a higher bonus if the applications running in production have fewer bugs. An obvious (and counterproductive) maneuver would be to prevent applications from going into production at all. Rejecting new application versions and sending them back to development due to poor quality (even if it’s not that bad) would improve that team’s situation, netting the higher bonus. The operation team profits, but the whole company will suffer, as does the customer.

When you set up a cross-functional process, you must often address worries over losing power, competence, influence, and control. What could be worse for a huge testing department than to have a continuous process make them redundant? Agile aims to address these worries. Agile will overcome these reservations and focus on those invisible facets of software development. A strong management commitment to the chosen approach and shared objectives are needed to roll out a CI process and an ALM in general.

SYNCHRONIZATION AND CONTINUOUS IMPROVEMENT

ALM drives a comprehensive approach to software development. Whenever a developer commits changes, the system builds the software, the tests are run, and the customer reviews the deployed test version—this type of communication is known as synchronization. But you shouldn’t apply CI that’s strictly shaped to technical artifacts (such as sources and tests), although artifacts are good bases for communication, because synchronization isn’t limited to artifacts. A comprehensive ALM approach also includes synchronization on other levels. Besides artifacts, clearly discussing tools and processes is also important. But tools and processes can’t substitute for personal communication and interactions because software is made by and for humans. The technical infrastructure should accelerate communication by condensing the information and transforming information into knowledge.

¹² Mel Conway, “How do Committees Invent?” <http://www.melconway.com/research/committees.html>.

Key characteristics of Agile releasing are continuous reflection (improvement of the process), detection of process defects, and improvement of processes. Adaptation of your process and how you work is only possible if you know where you stand at any given moment. If you don't know where you stand—and without CI it's hard to know—trying to improve your process is like shooting in the dark. Setting up CI helps you to increase quality, detect issues early, and deliver software more frequently.

To build and integrate software continuously, it's best to create a repository to store essential artifacts and facilitate code and component reuse.

2.2.4 **Component repository**

Component repository is a logical expression. Physically, a component repository can be the same as a sources repository (for instance, Subversion). Alternatively, a component repository may be hosted by a VCS, a file system, or a database. In contrast to traditional version control (source repositories, like CVS, Subversion, Git), a *component repository* contains the binary versions that are the build result of sources (see figure 2.4). In Java those binary versions are the standardized deployment units, like JAR, WAR, and EAR.

In an Agile ALM context, it's mandatory to manage sources in a VCS to enable concurrent modifications and provide a reproducible version history. It's common for companies to have essential artifacts hosted in a couple of different repositories. In the best scenario, a single access repository contains the components your project or company uses, but this isn't necessarily the repository that the sources are housed in. You may choose to use more than one repository.

Minimizing media versus using a component repository

Using many different repositories in parallel, such as CVS, Subversion, file servers, and so on, inhibits efficiency. It's important to make the locations transparent so you don't know, and don't need to know, where exactly the assets are stored. On the one hand, it's wise to reduce channels and mediums if possible. On the other hand, it can be wise to use a component repository.

It's often efficient to store deployment units and their versions and dependencies in a medium other than a VCS. It's even more efficient to store derived artifacts beside the original sources (because you don't want to build the software on all environments again). In some situations, you *must* store binaries due to reproducibility reasons: Sources have a reproducible context by applying tagging in a VCS. Binaries also have a context that is resolved at build time, such as version ranges or dynamic properties.

Which approach you choose will depend on your specific requirements. For many projects, it's helpful to host the sources and their deployment units in two different locations.

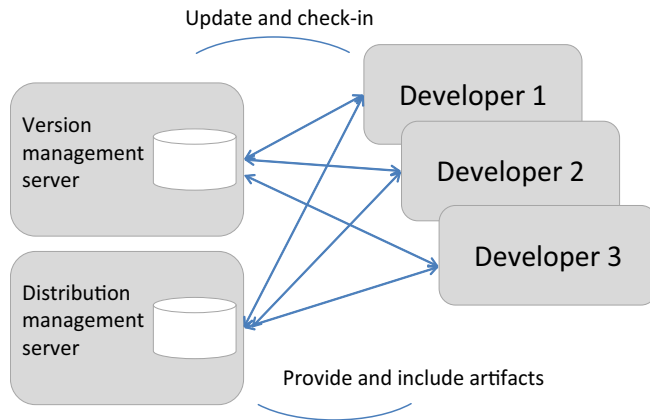


Figure 2.4 Version management servers store sources, and distribution management servers store binary artifacts

Labels should be sticky

Sources that are tagged with a VCS tag are labeled. A *label* is a snapshot that illustrates which state the software is in at the time. This approach is also known as *creating a baseline*. Suppose you use version 1 of a product that has the label 1.0b12. The twelfth build was the last successful one and was chosen to be the one for delivery to the outside world. It's not the twelfth version of 1.0b1.

Don't move or change labels once they're set, keep them. If you change parts of the system after labeling, label them again. You should never work with moving targets, sometimes called *floating* labels.

Often it's necessary to reproduce software that's running on different machines. It can therefore be useful for a central release department to label final versions of the software and put the build artifacts into a build archive. This ensures binary integrity, which means that for each versioned software state, the same deployment units are delivered to each target environment (there's no recompilation for further environments). Such a component repository can also protect company assets and boost reusability, as well as minimize the number and complexity of dependencies. Additionally, the artifacts in the component repository can be included as binary dependencies. Chapter 5 details this concept and implements it with tools.

2.2.5 Quality, standards, and release cycles

Productivity increases as quality improves, because less rework is necessary and waste (parts of the solution not needed to solve the given problem or to implement the requirements) is reduced.¹³ Simpler solutions lead to better quality, and quality improvements lead to lower costs, a better competitive position, and happier people

¹³ See W. Edwards Deming, *Out of the Crisis* (MIT Press, 1982), chapter 1.

on the job. In *Peopleware*, Tom DeMarco, and Timothy Lister state that “we all tend to tie our self-esteem strongly to the quality of the product we produce—not the quantity of product, but the quality” (p. 19), and they list “quality reduction of the product” as one of the negative “teamicide techniques” (p. 133). They also talk about having a “cult of quality” to foster team building (p. 151).¹⁴ Quality begins with the intent, which is fixed by management and is pursued by the whole team. Quality can vary in different contexts. Gerald M. Weinberg states that “quality is conforming to some person’s requirements.”¹⁵

You need an identifiable process to be able to improve on your process. Therefore, having a systematic release process is always a good idea. For fast release cycles, developing and releasing is the top priority; it’s necessary to aim for the best quality and to work according to the highest standards. Features and quality must be balanced. Refactoring to improve the design of the code without changing its functionality can help to improve existing code¹⁶—don’t provide functionality at the expense of technical debt and poor quality!

The quality can be kept high by running tests frequently and automatically checking metrics (audits). Rules for designing and coding shouldn’t only be described on paper, but also they should also be available as executable media. The integration build should run tests and audits to ensure that the quality requirements are fulfilled.

If the build fails, it’s referred to as being *broken*. If a build doesn’t fulfill the requirements, it’s possible to break it automatically. In contrast to measurable tests and audits, nonfunctional requirements (such as usability) can’t be tested automatically.

Lean software development

Lean software development, inspired by the success of the Toyota production system, claims to “stop the line” when defects are detected. If we transfer the picture of an assembly line being completely stopped when a bug is found on the staging ladder (not on the developer’s workspace, because it’s isolated), we have a CI landscape and builds that aren’t passed through if standards are ignored or the quality falls below the requirements.

A second major message of the Lean approach is to create “just in time” releases and to avoid waste. The Lean approach is articulated in seven principles: eliminate waste, amplify learning, decide as late as possible, deliver as fast as possible, empower the team, build integrity in, and see the whole. For further details, see *Lean Software Development*, *Implementing Lean Software Development*, and *Leading Lean Software Development* (by Poppendieck and Poppendieck, Addison-Wesley).

¹⁴ Tom DeMarco and Timothy Lister, *Peopleware* (Dorset House, 1999).

¹⁵ Gerald M. Weinberg, *Quality Software Management*, vol. 1 (Dorset House, 1992), p. 5.

¹⁶ See Martin Fowler, *Refactoring* (Addison-Wesley, 1999).

Results of tests and audits, as well as the verification of standards, should be handled according to fixed rules. Merely watching how tests fail, and then deploying the software anyway (perhaps after removing failed tests from the test suite), doesn't increase quality. Here, quality gates and a zero tolerance approach are the best practices for stopping the release when defects are detected. Otherwise, bugs will result in higher costs and missed deadlines.

In addition, an enterprise Agile ALM process as a whole should be standardized. It has to be carefully architected, with nonnegotiable elements. It should provide a mandatory framework that's capable of weaving together different elements to support each project's unique requirements.

Frequent releasing of software enables fast feedback, better measurability, and a meaningful picture of the software's status: "It forces you to get really good at doing releases and deployments."¹⁷ Only built, integrated, and deployed software gives you an idea of what the software accomplishes and what it doesn't. The content of the release should be fixed before starting it. The dates should also all be fixed and published in a publicly accessible release calendar. Consequently, time, (high) quality, and resources (people) are three cornerstones of the releases that are fixed, constant, and balanced with your individual requirements. It's the number of features that should be variable: At the end of the release, if the implementation of features isn't finished or if tests fail, the features should be postponed for a future release. Short iterations and time-boxing are ingredients of good risk management. In a dynamic environment, individually chosen release lengths are a good way to keep requirements and basic conditions stable. Chapter 7 illustrates auditing with tools.

2.3 The process pitfall, the illusion of control

There's a conflict between having too much lifecycle management process and not having enough. This conflict can be named the *process pitfall*. An Agile approach advances and demands feedback and communication. The release management process should be effective, efficient, and targeted. In practice, though, many projects suffer from not having enough process.

To resolve this conflict, you need to focus on priorities, including the root cause of the process pitfall. Often, more process is introduced in order to address the problem of the illusion of control. Introducing too many rules or wrong process rules could suggest control that doesn't truly exist. In the worst case, you have a described process and a real process in parallel. Or you have a rigid process that dramatically decreases productivity. ALM and its major facet, release management, must be a balanced set of processes and tools aligned with your individual requirements.

¹⁷ Michael T. Nygard, *Release It!* (The Pragmatic Bookshelf, 2007), p. 326.

2.3.1 *Effectiveness and efficiency*

Effectiveness is doing the right thing, and efficiency is doing the right thing correctly. After consulting on a significant number of projects, I'm left wondering why some teams don't grasp this distinction.

If you have issues (or better, challenges), try a root cause analysis to detect the original evil. If you find it, you can think about possible improvements. Mostly, they all have pros and cons, so decide wisely which way to go. Choose only a few pain points, sign up for the actions, and track them over your next development iteration to ensure you complete them successfully. If you dig into challenges deep enough, you'll usually find communication defects inside the team. This is what Agile is all about: Communication and interaction are more important than processes and tools, as the Agile manifesto says. If you can solve the people issues, yet still see room for improvement, proceed to the processes.

Defects in processes are often a problem. For example, it's not possible to configure a workflow system to cover your processes unless you know what the processes are. If they're not described, identify and describe them. Sometimes, processes don't exist at all. Set them up; don't be satisfied if the whole team speaks about the task of "daily business." If you're managing the processes, and you know the requirements, then, and only then, can you think about tooling. There's no point in buying a full-fledged commercial ALM suite or using some of the great tools I'll introduce in this book if you don't know your requirements (and consequently can't determine whether the tools fulfill them).

You can work with prototypes, evaluation versions of tools, or a "release 0.0/zero" for setting up infrastructure. These provide good ways to get early feedback and gain some valuable experience. But always remember that you should stay flexible. It's often better to use a collection of lightweight, integrated tools that are de facto standards on the market and that do the best job in their domain. You can integrate and decouple your infrastructure while remaining quite independent and flexible.

If you want to kick-start your development of new components, you may decide to use a build tool, such as Maven, that provides component and build management and a neat archetype feature. If you want to integrate your system continuously, add a build server to your infrastructure. You may want to add tests and audits later. Little by little, you can extend your infrastructure in a requirement-based, focused way. And if you're not satisfied with one decision, you can replace one tool while still sticking with the other ones.

Managing the identification of configuration items is also important in ensuring your process matches your requirements.

2.3.2 *Agile ALM and configuration items*

ALM deals with the management of tasks and artifacts. Controlling artifacts is only possible if the artifacts are identified: Without determining which artifacts affect the release and the project and without putting the artifacts into the ALM system, it's not

possible to control the artifacts or perform status accounting (to ensure completeness and provide a consistent version) and audits. Additionally, setting up an efficient ALM is only possible when processes and tools are optimally chosen, integrated, and standardized.

Identifying assets, controlling configuration items, and performing status accounting and audits are major tasks of traditional software configuration management (SCM). In an Agile ALM, you'll find the best fit to implement the traditional activities of a SCM—pure Agile projects implement SCM facets in an implicit way. There should be an SCM-aware expert on every Agile team or a traditional build manager or a (technical) release manager can drive the daily SCM needs of the business. This depends on how you slice your roles.

SCM is mainly about access to project artifacts. This includes not only tracking artifact versions over time, but also controlling and managing changes to them. Whereas in traditional SCM scenarios, you track every artifact, in an Agile ALM scope, you'll focus on final deployment units and important artifacts, including documents that influence the project (like requirement documents). For example, the Agile approach tracks EARs, WARs, and JARs independent of their contents (their packages and classes). The sources themselves are stored in the VCS. You won't store artifacts that you can generate out of other artifacts (unless you have good reason) or documents that won't change over time or that are written by multiple users (such as meeting minutes). See figure 2.5.

From an underlying SCM point of view, an Agile ALM focuses on aggregating and documenting the most important parts of the software necessary for release. For example, if you want to integrate further components or subsystems into your enter-

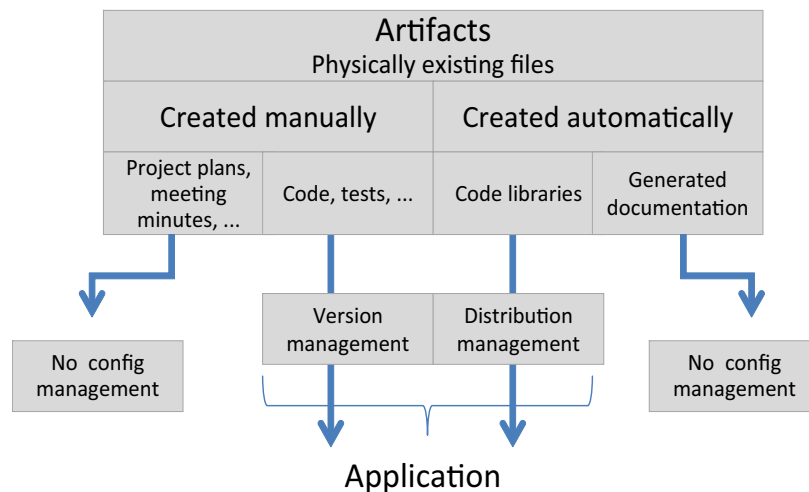


Figure 2.5 Artifacts in configuration management: Artifacts that are updated continuously and are of special interest are put into configuration management. Sources and tests are put into version control; libraries are put into distribution management.

prise SCM, you need basic information about these components, including their deployment units. Table 2.2 collects some of the major components in an SCM checklist. This is a much leaner approach than is promoted by traditional SCM. Depending on your particular situation and requirements, the checklist can be implemented in a smart way. If you use Maven, for instance, some of the checklist items are covered out of the box (such as documenting deployment units). Tools like Maven can help you define your SCM in an executable medium. This means, for instance, you have XML definitions that can be executed reproducibly.

Table 2.2 Approaching SCM in an Agile way: the SCM checklist

Group	Item	Details
Overview	Configuration elements	Complete list of all configuration elements, including scripts, database elements, deployment units, properties.
System	Deployment diagram	Deployment units (and their versions), packaging types, protocols, technical information (like a version of an application server), dependencies between configuration elements, nodes.
System	Infrastructure	Database elements (users, DDL), technical users, permissions, security.
System	Test environments	For all subsystems, mapping to other test environments where needed.
Build	Build system	System must provide its deployment units in a reproducible way (build must be provided by component development team).

Traditional SCM requires listings of configuration items and checklists. This can be necessary in an Agile context, too, but it's usually handled automatically by the ALM tools. But there are many possible usage models.

Release notes should be created automatically. You should also be able to audit the system automatically. For example, the JIRA issue and project tracking tool can be used to derive release notes based on a specific time interval or version number. Mapping sources to requirements to generate a list of what you're looking for is an effective way to create documentation automatically, along with applying active impact management with Mylyn or FishEye. Acceptance tests (for example, creating and running tests with the Fit tool) can also be part of this documentation.

Tests and audits (metrics) can be applied automatically as part of the continuous integration system. You can base these audits on tools like Checkstyle and Cobertura and your testing on Selenium, FEST, or something similar. Track your components in a component repository, where the system puts them continuously. If you already use Maven, you're familiar with this; if this is completely new for you, don't panic. Either way, this book will give you valuable tips.

Agile ALM minimizes overhead while maximizing benefit. It also acts as an enabler for change. We'll discuss this next.

2.3.3 Agile ALM as change enabler

All systems try to achieve stable states (*panta rhei*¹⁸). Being flexible in software development doesn't mean chaotic drifting, but rather, being able to change and transform from one stable state to another. Any substantial improvements must come from an action on the system.¹⁹ This is management's responsibility of management and the ALM system can improve management's ability to know what's happening (meta-measurement²⁰) and can improve insight into the best decisions.

The importance of lifecycle management will continue to grow. In this time of distributed, heterogeneous system landscapes, legacy systems that must be integrated, systems and components in many different versions, and (transitive) dependencies on different platforms, following a systematic release management approach has become a precondition to providing high-quality software in constant, short intervals. Agile ALM is the catalyst that enables the daily work of all project stakeholders. It also helps track and control the artifacts that were created during the project activities.

Agile ALM acts as a change-enabler. During the development of complex systems, change is a constant companion of the development process. Instead of being exceptional, changes are more and more the norm. A high percentage of projects miss their project goals because they don't grant enough space for changes in the process. Modern software development understands that changes are a major part of the project. They're part of the process of aligning the current activities with the valid requirements and basic conditions at any time (a process of continuous adaptation).

In the extreme approach, defects (bugs) and all kinds of functional and nonfunctional requirements are handled like a coordinated set of changes to the system. Following this paradigm, software development is the process of identifying and processing changes. ALM is evolving to be the hub of reproducibility and is the change enabler.

2.4 Summary

In this chapter, you learned about Agile and what Agile means in the context of ALM. We discussed continuous integration in detail and considered many aspects of the Agile ALM, concluding with Agile ALM being a change enabler. In the rest of the book, we'll implement an Agile ALM with lightweight tools and apply those Agile strategies. In the next chapter, we'll use Scrum to bridge functional releasing to technical releasing.

¹⁸ Meaning "everything flows," here: it flows from one stable state to another.

¹⁹ See W. Edwards Deming, *Out of the Crisis* (MIT Press, 1982), chapter 11.

²⁰ Gerald M. Weinberg, *Quality Software Management*, vol. 2 (Dorset House, 1993), chapter 12.

Agile ALM

Michael Hüttermann



Agile Application Lifecycle Management combines flexible processes with lightweight tools in a comprehensive and practical approach to building, testing, integrating, and deploying software. Taking an agile approach to ALM improves product quality, reduces time to market, and makes for happier developers.

Agile ALM is a guide for Java developers, testers, and release engineers. By following dozens of experience-driven examples, you'll learn to see the whole application lifecycle as a set of defined tasks, and then master the tools and practices you need to accomplish those tasks effectively. The book introduces state-of-the-art, lightweight tools that can radically improve the speed and fluidity of development and shows you how to integrate them into your processes.

What's Inside

- A thorough introduction to Agile ALM
- Build an integrated Java-based Agile ALM toolchain
- Use Scrum for release management
- Reviewed by a team of 20 Agile ALM experts

The tools and examples are Java-based, but the Agile ALM principles apply to all development platforms.

Michael Hüttermann is a Java Champion, a member of the JCP and Agile Alliance, and founder of the Cologne Java User Group. He led the Tools for Agility track at Agile 2009. The Technical Editor on this book was **Robert Aiello**.

For access to the book's forum and a free ebook for owners of this book, go to manning.com/AgileALM

“Agile answers for managers, leads, and engineers.”

—Ben Ogden, General Dynamics
Advanced Information Systems

•
“Power up your Agile ALM tool chest with this book!”

—Tariq Ahmed
Amcom Technology

•
“A practical guide to improving the software delivery process.”

—Darren Neimke
HomeStart Finance

•
“Covers a wide variety of tools that make up the Agile ALM.”

—Craig Smith, Suncorp

•
“A great book for streamlining and improving your projects!”

—Robert Wenner
Port25 Solutions, Inc.

