

Extracted from:

# Practices of an Agile Developer

---

## Working in the Real World

This PDF file contains pages extracted from Practices of an Agile Developer, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragmaticprogrammer.com>.

**Note:** This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2005 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

*No matter how far down the wrong road  
you've gone, turn back.*

► Turkish proverb

## Chapter 1

# Agile Software Development

---

That Turkish proverb above is both simple and obvious—you'd think it would be a guiding force for software development. But all too often, developers (including your humble authors) continue down the wrong road in the misguided hope that it will be OK somehow. Maybe it's close enough. Maybe this isn't *really* as wrong a road as it feels. We might even get away with it now and then, if creating software were a linear, deterministic process—like the proverbial road. But it's not.

Instead, software development is more like surfing—it's a dynamic, ever-changing environment. The sea itself is unpredictable, risky, and there may be sharks in those waters.

But what makes surfing so challenging is that *every wave is different*. Each wave takes its unique shape and behavior based on its locale—a wave in a sandy beach is a lot different from a wave that breaks over a reef, for instance.

In software development, the requirements and challenges that come up during your project development are your waves—never ceasing and ever-changing. Like the waves, software projects take different shapes and pose different challenges depending on your domain and application. And sharks come in many different guises.

Your software project depends on the skills, training, and competence of all the developers on the team. Like a successful surfer, a successful developer is the one with (technical) fitness, balance, and agility. Agility in both cases means being able to quickly *adapt* to the unfolding situation, whether it's a wave that breaks sooner than expected or a design that breaks sooner than expected.

### 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.

*Copyright 2001, the Agile Manifesto authors*

See [agilemanifesto.org](http://agilemanifesto.org) for more information.

## The Spirit of Agility

So what is agility, exactly, and where did this whole agile software development movement come from?

In February 2001, seventeen interested persons (including Andy) got together in Snowbird, Utah, to discuss an emerging trend of what was loosely being called *lightweight processes*.

We had all seen projects fail because of ponderous, artifact-heavy, and results-light processes. It seemed like there should be a better way to look at methodology—a way to focus on the important stuff and de-emphasize the less important stuff that seemed to take up a lot of valuable time with little benefit.

These seventeen folks coined the term *agile* and published the Agile Manifesto to describe a refocused approach to software development: an approach that emphasizes people, collaboration, responsiveness, and working software (see the sidebar on this page for the introduction to the manifesto).

The agile approach combines responsive, collaborative people with a focus on demonstrable, concrete goals (software that actually works). That's the spirit of agility. The practical emphasis of development shifts

from a plan-based approach, where key events happen in individual, separate episodes, to a more natural, continuous style.

It's assumed that everyone on the team (and working with the team) are professionals who want a positive outcome from the project. They may not necessarily be *experienced* professionals yet, but they possess a professional attitude—everyone wants to do the best job they can.

If you have problems with absenteeism, slackers, or outright saboteurs, this is probably not the approach for you. You'll need something more heavy-handed, slower, and less productive. Otherwise, you can begin developing in the agile style.

That means you don't leave testing to the end of the project. You don't leave integration to the end of the month or stop gathering requirements and feedback as you begin to code.

Instead, you continue to perform all these activities throughout the life cycle of the project. In fact, since software is never really “done” as long as people continue to use it, it's arguable that these aren't even *projects* anymore. Development is continuous. Feedback is continuous. You don't have to wait for months to find out that something is wrong: you find out quickly, while it's still relatively easy to fix. And you fix it, right then and there.

---

Continuous  
development, not  
episodic

---

That's what it's all about.

This idea of continuous, ongoing development is pervasive in agile methods. It includes the development life cycle itself but also technology skills learning, requirements gathering, product deployment, user training, and everything else. It encompasses all activities, at all levels.

Why? Because developing software is such a complex activity, anything substantive that you leave until later won't happen, won't happen well, or will grow worse and fester until it becomes unmanageable. A certain kind of friction increases, and things get harder to fix and harder to change. As with any friction, the only way to fight it effectively is to continually inject a little energy into the system (see “Software Entropy” in *The Pragmatic Programmer* [HTOO]).

---

Inject energy

---

Some people raise the concern that agile development is just *crisis management* in disguise. It's not. Crisis management occurs when problems are left to fester until they become so large that you have to drop everything else you're doing to respond to the crisis immediately. This causes secondary crises, so now you have a vicious cycle of never-ending crisis and panic. That's precisely what you want to avoid.

Instead, you want to tackle small problems while they are still small, explore the unknown before you invest too much in it, and be prepared to admit you got it all wrong as soon as you discover the truth. You need to retool your thinking, your coding practices, and your teamwork. It's not hard to do, but it might feel different at first.

## The Practice of Agility

A useful definition of *agility* might be as follows:

Agile development uses feedback to make constant adjustments in a highly collaborative environment.

Here's a quick summary of what that means in practice and what life on an agile team looks like.

It's a team effort. Agile teams tend to be small or broken up into several small (ten or so people) teams. You mostly work very closely together, in the same war room (or bull pen) if possible, sharing the code and the necessary development tasks. You work closely with the client or customer who is paying for this software and show them the latest version of the system as early and as often as possible.

You get constant feedback from the code you're writing and use automation to continuously build and test the project. You'll notice that the code needs to change as you go along: while the functionality remains the same, you'll still need to redesign parts of the code to keep up. That's called *refactoring*, and it's an ongoing part of development—code is never really “done.”

Work progresses in *iterations*: small blocks of time (a week or so) where you identify a set of features and implement them. You demo the iteration to the customer to get feedback (and make sure you're headed in

the right direction) and release full versions to the user community as often as practical.

With this all in mind, we're going to take a closer look at the practices of agility in the following areas:

**Chapter 2: Beginning Agility.** Software development is all in your head. In this chapter, we'll explain what we mean by that and how to begin with an agile mind-set and good personal practices as a firm foundation for the remainder of the book.

**Chapter 3: Feeding Agility.** An agile project doesn't just sit there. It requires ongoing background practices that aren't part of development itself but are vitally important to the health of the team. We'll see what needs to be done to help keep your team and yourself growing and moving forward.

**Chapter 4: Delivering What Users Want.** No matter how well written, software is useless if it doesn't meet the users' needs. We'll take a look at practices and techniques to keep the users involved, learn from their experience with the system, and keep the project aligned with their real needs.

**Chapter 5: Agile Feedback.** Using feedback to correct the software and the development process is what keeps an agile team on course where others might flounder and crash. The best feedback comes from the code itself; this chapter examines how to get that feedback as well as how to get a better handle on the team's progress and performance.

**Chapter 6: Agile Coding.** Keeping code flexible and adaptable to meet an uncertain future is critical to agile success. This chapter outlines some practical, proven techniques to keep code clean and malleable and prevent it from growing into a monster.

**Chapter 7: Agile Debugging.** Debugging errors can chew through a lot of time on a project—time you can't afford to lose. See how to make your debugging more effective and save time on the project.

**Chapter 8: Agile Collaboration.** Finally, an agile developer can be only so effective; beyond that, you need an agile team. We'll show you the most effective practice we've found to help jell a team together, as well as other practices that help the team function on a day-to-day basis and grow into the future.

## An Agile Toolkit

---

Throughout the text, we'll refer to some of the basic tools that are in common use on agile projects. Here's a quick introduction, in case some of these might be new to you. More information on these topics is available from the books listed in the bibliography.

**Wiki.** A Wiki (short for WikiWikiWeb) is a website that allows users to edit the content and create links to new content using just a web browser. Wikis are a great way to encourage collaboration, because everyone on the team can dynamically add and rearrange content as needed. For more on Wikis, see *The Wiki Way* (LC01).

**Version control.** Everything needed to build the project—all source code, documents, icons, build scripts, etc.—needs to be placed in the care of a version control system. Surprisingly, many teams still prefer to plop files on a shared network drive, but that's a pretty amateurish approach. For a detailed guide to setting up and using version control, see *Pragmatic Version Control Using CVS* (TH03) or *Pragmatic Version Control Using Subversion* (Mas05).

**Unit testing.** Using code to exercise code is a major source of developer feedback; we'll talk much more about that later in the book, but be aware that readily available frameworks handle most of the housekeeping details for you. To get started with unit testing, there's *Pragmatic Unit Testing in Java* (HT03) and *Pragmatic Unit Testing in C#* (HT04), and you'll find helpful recipes in *JUnit Recipes* (Rai04).

**Build automation.** Local builds on your own machine, as well as centrally run builds for the whole team, are completely automated and reproducible. Since these builds run all the time, this is also known as *continuous integration*. As with unit testing, there are plenty of free, open-source and commercial products that will take care of the details for you. All the tips and tricks to build automation (including using lava lamps) are covered in *Pragmatic Project Automation* (Cla04).

Finally, you can find a good reference to tie these basic environmental practices together in *Ship It!* (RG05).

## The Devil and Those Pesky Details

If you've flipped through the book, you may have noticed that the introduction section of the tips features a small woodcut of the devil himself, tempting you into bad and careless habits. They look like this:

*“Go ahead, take that shortcut. It will save you time, really. No one will ever know, and you can be done with this task and move on quickly. That’s what it’s all about.”*



Some of his taunts may seem absurd, like something out of Scott Adams's *Dilbert* cartoons and his archetypical “pointy-haired boss.” But remember Mr. Adams takes a lot of input from his loyal readers.

Some may seem more outlandish than others, but they are all legitimate lines of thought that your authors have heard, seen in practice, or secretly thought. These are the temptations we face, the costly shortcut we try anyway, in the vain hope of saving time on the project.

To counter those temptations, there's another section at the end of each practice where we'll give you your own guardian angel, dispensing key advice that we think you should follow:



***Start with the hardest.*** Always tackle the most difficult problems first, and leave the simple one towards the end.

And since the real world is rarely that black-and-white, we've included sections that describe what a particular practice should feel like and tips on how to implement it successfully and keep it in balance. They look like this:

## What It Feels Like

This section describes what a particular practice *should* feel like. If you aren't experiencing it this way, you may need to revise how you're following a particular practice.

## Keeping Your Balance

- It's quite possible to overdo or underdo a practice, and in these sections we'll try to give you advice to keep a practice in balance, as well as general tips to help make it work for you.

After all, too much of a good thing, or a good thing misapplied, can become very dangerous (all too often we've seen a so-called agile project fail because the team didn't keep a particular practice in balance). We want to make sure you get real benefits from these practices.

By following these practices and applying them effectively in the real world—with balance—you'll begin to see a positive change on your projects and in your team.

You'll be following the practices of an agile developer, and what's more, you'll understand the principles that drive them.

## Acknowledgments

Every book you read is a tremendous undertaking and involves many more people behind the scenes than just your lowly authors.

We'd like to thank all the following people for helping make this book happen.

Thanks to Jim Moore for creating the cover illustration and to Kim Wimpsett for her outstanding copyediting (and any remaining errors are surely the fault of our last-minute edits).

A special thanks to Johannes Brodwall, Chad Fowler, Stephen Jenkins, Bil Kleb, and Wes Reisz for their insight and helpful contributions.

And finally, thanks to all our reviewers who graciously gave their time and talent to help make this a better book: Marcus Ahnve, Eldon Alameda, Sergei Anikin, Matthew Bass, David Bock, A. Lester Buck III, Brandon Campbell, Forrest Chang, Mike Clark, John Cook, Ed Gibbs, Dave Goodlad, Ramamurthy Gopalakrishnan, Marty Haught, Jack Herrington, Ron Jeffries, Matthew Johnson, Jason Hiltz Laforge, Todd Little, Ted Neward, James Newkirk, Jared Richardson, Frédéric Ros, Bill Rushmore, David Lázaro Saz, Nate Schutta, Matt Secoske, Guerry Semones, Brian Sletten, Mike Stok, Stephen Viles, Leif Wickland, and Joe Winter.

*Venkat says:*

I would like to thank Dave Thomas for being such a wonderful mentor. Without his guidance, encouragement, and constructive criticism this book would have stayed a great idea.

I'm blessed to have Andy Hunt as my coauthor; I've learned a great deal from him. He is not only technically savvy (a fact that any pragmatic programmer out there already knows) but has incredible expressive power and exceptional attitude. I have admired the Pragmatic Programmers in every step of making of this book—they've truly figured and mastered the right set of tools, techniques, and, above all, attitude that goes into publishing.

I thank Marc Garbey for his encouragement. The world can use more people with his sense of humor and agility—he's a great friend. My special thanks to the geeks (err, friends) I had the pleasure to hang out with on the road—Ben Galbraith, Brian Sletten, Bruce Tate, Dave Thomas, David Geary, Dion Almaer, Eitan Suez, Erik Hatcher, Glenn Vanderburg, Howard Lewis Ship, Jason Hunter, Justin Gehtland, Mark Richards, Neal Ford, Ramnivas Laddad, Scott Davis, Stu Halloway, and Ted Neward—you guys are awesome! I thank Jay Zimmerman (a.k.a. agile driver), director of NFJS, for his encouragement and providing an opportunity to express my ideas on agility to his clients.

I thank my dad for teaching me the right set of values, and to you, Mom, for you're my true inspiration. None of this would have been possible but for the patience and encouragement of my wife, Kavitha, and my sons, Karthik and Krupakar; thank you and love you.

*Andy says:*

Well, I think just about everyone has been thanked already, but I'd like to thank Venkat especially for inviting me to contribute to this book. I wouldn't have accepted that offer from just anyone, but Venkat has been there and done that. He *knows* how this stuff works.

I'd like to thank all the good agile folks from the Snowbird get-together. None of us invented agility, but everyone's combined efforts have certainly made it a growing and powerful force in the modern world of software development.

And of course, I'd like to thank my family for their support and understanding. It has been a long ride from the original *The Pragmatic Programmer* book, but it has been a fun one.

And now, on with the show.

*He who chooses the beginning of a road  
chooses the place it leads to.*

► Harry Emerson Fosdick

## Chapter 2

# Beginning Agility

---

Traditional books on software development methodology might start with the Roles you'll need on a project, followed by the many Artifacts you need to produce (documents, checklists, Gantt charts, and so on). After that you'll see the Rules, usually expressed in a somewhat "Thou Shalt..." format.<sup>1</sup> Well, we're not going to do any of that here. Welcome to agility, where we do things a bit differently.

For instance, one popular software methodology suggests you need to fulfill some thirty-five distinct roles on a project, ranging from architect to designer to coder to librarian. Agile methods take a different tack. You perform just one role: software developer. That's you. You do what's needed on the team, working closely with the customer to build software. Instead of relying on Gantt charts and stone tablets, agility relies on people.

Software development doesn't happen in a chart, an IDE, or a design tool; it happens in your head. But it's not alone. There's a lot of other stuff happening in there as well: your emotions, office politics, egos, memories, and a whole lot of other baggage. Because it's all mixed in together, things as ephemeral as *attitude* and *mood* can make a big difference.

And that's why it's important to pay attention to attitude: yours and the team's. A professional attitude focuses on positive outcomes for the project and the team, on personal and team growth, and on success. It's easy to fall into pursuing less noble goals, and in this chapter,

---

<sup>1</sup>Or the ever popular, "The System shall..."

we'll look at ways to stay focused on the real goals. Despite common distractions, you want to *Work for Outcome* (see how beginning on the next page).

Software projects seem to attract a lot of time pressure—pressure that encourages you to take that ill-advised shortcut. But as any experienced developer will tell you, *Quick Fixes Become Quicksand* (see how to avoid the problem starting on page 15).

Each one of us has a certain amount of ego. Some of us (not naming names here) have what might be charitably termed a very “healthy” amount of ego; when asked to solve a problem, we take pride in arriving at the solution. But that pride can sometimes blind our objectivity. You've probably seen design discussions turn into arguments about individuals and personalities, rather than sticking to the issues and ideas related to the problem at hand. It's much more effective to *Criticize Ideas, Not People* (it's on page 18).

Feedback is fundamental to agility; you need to make changes as soon as you realize that things are headed in the wrong direction. But it's not always easy to point out problems, especially if there may be political consequences. Sometimes you need courage to *Damn the Torpedoes, Go Ahead* (we'll explain when, starting on page 23).

Agility works only when you adopt a professional attitude toward your project, your job, and your career. Without the right attitude, these practices won't help all that much. But with the right attitude, you can reap the full benefits of this approach. Here are the practices and advice we think will help.

## Work for Outcome

*“The first and most important step in addressing a problem is to determine who caused it. Find that moron! Once you’ve established fault, then you can make sure the problem doesn’t happen again. Ever.”*



Sometimes that old devil sounds so plausible. Certainly you want to make finding the culprit your top priority, don’t you? The bold answer is no. Fixing the problem is the top priority.

You may not believe this, but not everyone always has the outcome of the project as their top priority. Not even you. Consider your first, “default” reaction when a problem arises.

You might inadvertently fuel the problem by saying things that will complicate things further, by casting blame, or by making people feel defensive. Instead, take the high road, and ask, “What can I do to solve this or make it better?” In an agile team, the focus is on outcomes. You want to focus on fixing the problem, instead of affixing the blame.

---

### Blame doesn’t fix bugs

---

The worst kind of job you can have (other than cleaning up after the elephants at the circus) is to work with a bunch of highly reactive people. They don’t seem interested in solving problems; instead, they take pleasure in talking about each other behind their backs. They spend all their energy pointing fingers and discussing who they can blame. Productivity tends to be pretty low in such teams. If you find yourself on such a team, don’t walk away from it—run. At a minimum, redirect the conversation away from the negative blame game toward something more neutral, like sports or the weather (“So, how about those Yankees?”).

On an agile team, the situation is different. If you go to an agile team member with a complaint, you’ll hear, “OK, what can I do to help you with this?” Instead of brooding over the problem, they’ll direct their efforts toward solving it. Their motive is clear; it’s the outcome that’s important, not the credit, the blame, or the ongoing intellectual superiority contest.

You can start this yourself. When a developer comes to you with a complaint or a problem, ask about the specifics and how you can help. Just that simple act makes it clear that you intend to be part of the

### **Compliance Isn't Outcome**

Many standardization and process efforts focus on measuring and rating *compliance to process* on the rationale that if the process works and it can be proved that you followed it exactly, then all is right with the world.

But the real world doesn't work that way. You can be ISO-9001 certified and produce perfect, lead-lined life jackets. You followed the documented process perfectly; too bad all the users drowned.

Measuring compliance to process doesn't measure outcome. Agile teams value *outcome* over process.

solution, not the problem; this takes the wind out of negativism. You're here to help. People will then start to realize that when they approach you, you'll genuinely try to help solve problems. They can come to you to get things fixed and go elsewhere if they're still interested in whining.

If you approach someone for help and get a less than professional response, you can try to salvage the conversation. Explain exactly what you want, and make it clear that your goal is the solution, not the blame/credit contest.



***Blame doesn't fix bugs.*** *Instead of pointing fingers, point to possible solutions. It's the positive outcome that counts.*

### **What It Feels Like**

It feels safe to admit that you don't have the answer. A big mistake feels like a learning opportunity, not a witch hunt. It feels like the team is working together, not blaming each other.

## Keeping Your Balance

- “It’s not my fault” is rarely true. “It’s all your fault” is usually equally incorrect.
- If you aren’t making *any* mistakes, you’re probably not trying hard enough.
- It’s not helpful to have QA argue with developers whether a problem is a defect or an enhancement. It’s often quicker to fix it than argue about it.
- If one team member misunderstood a requirement, an API call, or the decisions reached in the last meeting, then it’s very likely other team members may have misunderstood as well. Make sure the whole team is up to speed on the issue.
- If a team member is repeatedly harming the team by their actions, then they are not acting in a professional manner. They aren’t helping move the team toward a solution. In that case, they need to be removed from this team.<sup>2</sup>
- If the majority of the team (and especially the lead developers) don’t act in a professional manner and aren’t interested in moving in that direction, then you should remove yourself from the team and seek success elsewhere (which is a far better idea than being dragged into a “Death March” project [You99]).

---

<sup>2</sup>They don’t need to be fired, but they don’t need to be on this team. But be aware that moving and removing people is dangerous to the team’s overall balance as well.