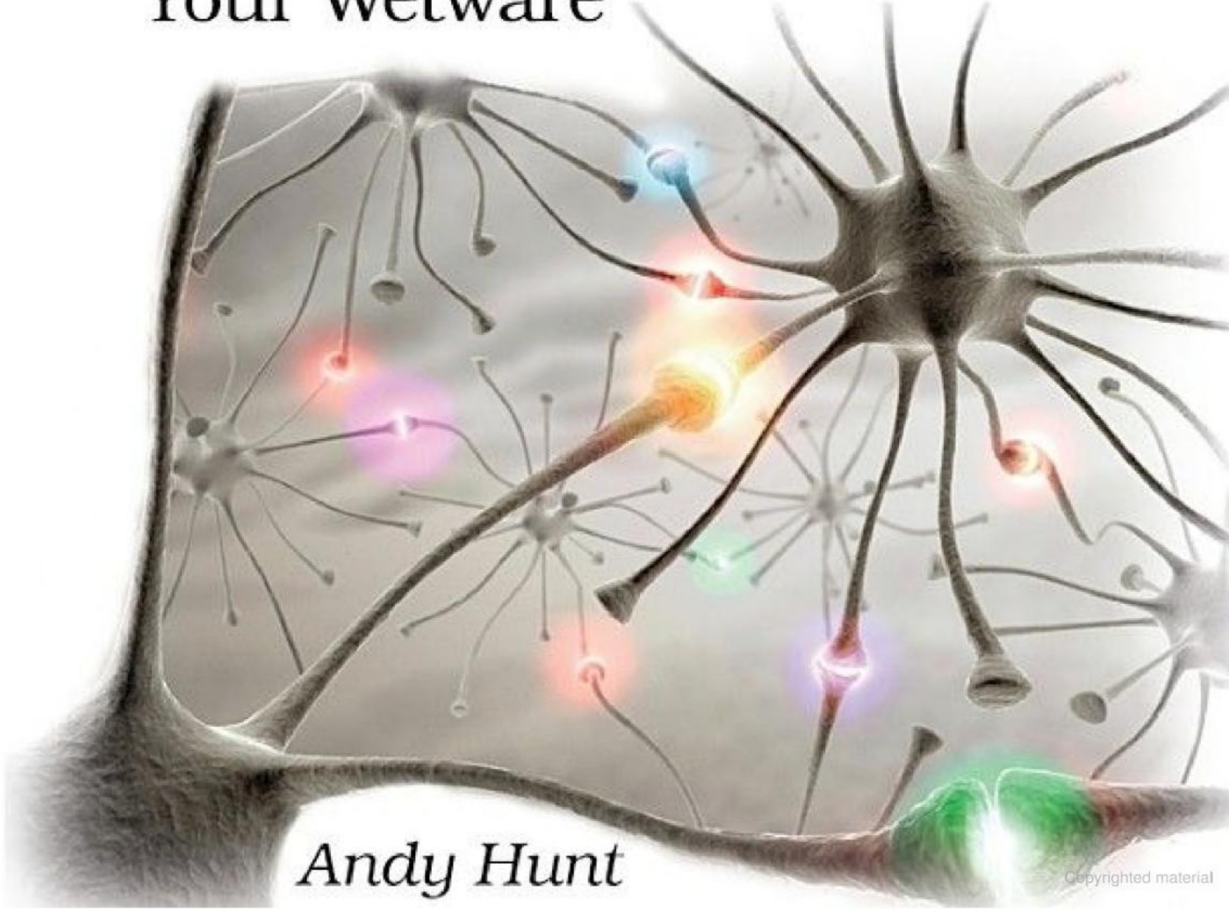
The Pragmatic Programmers

Pragmatic Thinking & Learning

Refactor Your Wetware



Andy Hunt

Table of Contents

What people are saying about Pragmatic Thinking and Learning

This book will be the catalyst for your future.

→ Patrick Elder

Agile Software Developer

By following Andy's concrete steps, you can make your most precious asset—your brain—more efficient and productive. Read this book, and do what Andy tells you to do. You'll think smarter, work better, and learn more than ever before.

→ Bert Bates

Cocreator of Head First, Brain Friendly Books

I've always been looking for something to help me improve my learning skills, but I've never found anything as effective as this book. *Pragmatic Thinking and Learning* represents the best way to help you become an expert learner, improve your skills, and teach you how to improve your work efficiency by learning fast and easily.

→ Oscar Del Ben

Software developer

This is an accessible and insightful book that will be useful to readers in many fields. I enjoyed reading it!

→ Dr. Patricia Benner

Professor and Chair, Department of Social and Behavioral Sciences, University of California, San Francisco

I love books that explain that context matters. This book does—and helps you understand why. From the Dreyfus model (a source of many "aha's" for me) to explaining why experiential training works (the wall climbing story), Andy writes with humor and with tact so you can learn from reading and organize your own thinking and learning.

→ Johanna Rothman

Consultant, author, and speaker

Finished reading the beta last night. I loved this talk at NFJS (and the herding racehorses one), and to have it in book form—spectacular. All of this material has really changed my life!

→ Matt McKnight

Software developer

This has been fun, and I've learned a lot—can't ask for more than that.

→ Linda Rising

International speaker, consultant, and object-oriented expert

Chapter 1

Introduction

Welcome!

Thanks for picking up this book. Together, we're going to journey through bits of cognitive science, neuroscience, and learning and behavioral theory. You'll see surprising aspects of how our brains work and see how you can beat the system to improve your own learning and thinking skills.

We're going to begin to refactor your wetware—redesign and rewire your brain—to make you more effective at your job. Whether you're a programmer, manager, "knowledge worker," technogeek, or deep thinker, or if you just happen to have a human brain you'd like to crank up, this book will help.

I'm a programmer, so my examples and rants will be directed at the world of software development. If you're not a programmer, don't worry; programming really has little to do with writing software in arcane, cryptic languages (although we have a curious attachment to that habit).

Programming is all about problem solving. It requires creativity, ingenuity, and invention. Regardless of your profession, you probably also have to solve problems creatively. However, for programmers, combining rich, flexible human thought with the rigid constraints of a digital computer exposes the power and the deepest flaws of both.

Whether you're a programmer or frustrated user, you may have already suspected that software development must be the most difficult endeavor ever envisioned and practiced by humans. Its complexity strains our best abilities daily, and failures can often be spectacular—and newsworthy. We've smashed spaceships into distant planets, blown up expensive rockets filled with irreplaceable experiments, plagued consumers with automated collection letters for $0.00, and stranded airline travelers on a semiregular basis.

But now the good news (sort of): it's all our fault. We tend to make programming much harder on ourselves than we need. Because of the way the industry has evolved over time, it seems we've lost track of some of the most fundamental, most important skills needed by a software developer.

The good news is that we can fix that right here and right now. This book will help show you how.

The number of bugs programmers introduce into programs has remained constant for the past forty years. Despite advances in programming languages, techniques, project methodologies, and so on, the defect density has remained fairly constant.[1]

Maybe that's because we've been focusing on the wrong things. Despite all these obvious changes in technology, one thing has remained constant: us. Developers. People.

Software isn't designed in an IDE or other tool. It's imagined and created in our heads.

Ideas and concepts are shared and communicated among a team, including the folks who are paying our organization to develop this software. We've spent the time investing in basic technology—in languages, tools, methodologies. That was time well spent, but now it's time to move on.

> *Software is created in your head.*

Now we need to look at the really hard problems of social interaction in and between teams and even at the harder issues of just plain old thinking. No project is an island; software can't be built or perform in isolation.

Frederick Brooks, in his landmark paper *No Silver Bullet—Essence and Accident in Software Engineering* [Bro86], claimed that "the software product is embedded in a cultural matrix of applications, users, laws, and machine vehicles.[2] These all change continually, and their changes inexorably force change upon the software product."

Brooks' observation puts us squarely at the center of the maelstrom of society itself. Because of this complex interaction of many interested parties and forces and the constant evolution of change, it seems to me that the two most important modern skills are these:

- Communication skills
- Learning and thinking skills

Some improvement to communication skills is being addressed by our industry. Agile methods (see the sidebar *What Are Agile Methods?*), in particular, emphasize improved communications between team members and between the ultimate customer and the development team. Mass-media books such as *Presentation Zen: Simple Ideas on Presentation Design and Delivery* [Rey08] are suddenly best-sellers as more and more people realize the importance of simple, effective communication. It's a good start.

But then there's learning and thinking, which is a much harder nut to crack.

Programmers have to learn constantly—not just the stereotypical new technologies but also the problem domain of the application, the whims of the user community, the quirks of their teammates, the shifting sands of the industry, and the evolving characteristics of the project itself as it is built. We have to learn—and relearn—constantly. Then we have to apply this learning to the daily barrage of both old and new problems.

It sounds easy enough in principle perhaps, but learning, critical thinking, creativity, and invention—all those mind-expanding skills—are all up to you. You don't get taught; you have to learn. We tend to look at the teacher/learner relationship the wrong way around: it's not that the teacher *teaches*; it's that the student *learns*. The learning is always up to you.

It's my hope that *Pragmatic Thinking and Learning* can help guide you through accelerated and enhanced learning and more pragmatic thinking.

What Are Agile Methods?

The term *agile methods* was coined at a summit meeting in February 2001 by seventeen leaders in software development, including the founders of various development methodologies such as Extreme Programming, Scrum, Crystal, and, of course, our very own pragmatic programming.

Agile methods differ from traditional plan-based methods in a number of significant ways, most notably in eschewing rigid rules and discarding dusty old schedules in favor of adapting to real-time feedback.

I'll talk about agile methods often throughout the book, because many of the agile ideas and practices fit in well with good cognitive habits.

Again with the "Pragmatic"?

From the original *The Pragmatic Programmer: From Journeyman to Master* [HT00] to our Pragmatic Bookshelf publishing imprint, you will notice that we have a certain preoccupation with the word *pragmatic*. The essence of *pragmatism* is to do what works—for you.

So before we begin, please bear in mind that every individual is different. Although many of the studies that I'll reference have been conducted on large populations, some have not. I'm going to draw on a large variety of material ranging from hard scientific fact proven with functional MRI scans of the brain to conceptual theories, as well as material ranging from old wives' tales to "Hey, Fred tried it, and it worked for him."

In many cases—especially when discussing the brain—the underlying scientific reasons are unknown or unknowable. But do not let that worry you: if it works, then it's pragmatic, and I will offer it here for your consideration. I hope many of these ideas will work for you.

But some folks are just plain wired differently; you may be one of them. And that's OK; you shouldn't follow any advice blindly. Even mine. Instead, read with an open mind. Try the suggestions, and decide what works for you.

Only dead fish go with the flow.

As you grow and adapt, you may need to modify your habits and approaches as well. Nothing in life is ever static; only dead fish go with the flow. So, please take this book as just the beginning.

I'll share the pragmatic ideas and techniques I've found in my journey; the rest is up to you.

What Is Wetware?

wet•ware | ˈwetwe(ə)rˌ |: etymology: wet + software

Noun, humorous. Human brain cells or thought processes regarded as analogous to, or in contrast with, computer systems.

That is, using the model of a computer as an analogy to human thought processes.

Consider the Context

Everything is interconnected: the physical world, social systems, your innermost thoughts, the unrelenting logic of the computer—everything forms one immense, interconnected system of reality. Nothing exists in isolation; everything is part of the system and part of a larger context.

Because of that inconvenient fact of reality, small things can have unexpectedly large effects. That disproportionate effect is the hallmark of nonlinear systems, and in case you hadn't noticed, the real world is decidedly nonlinear.

> When we try to pick out anything by itself, we find it hitched to everything else in the universe.
>
> ➤ *John Muir, 1911, My First Summer in the Sierra*

Throughout this book, you'll find activities or differences that seem to be so subtle or inconsequential that they couldn't *possibly* make a difference. These are activities such as thinking a thought to yourself vs. speaking it out loud or such as writing a sentence on a piece of paper vs. typing it into an editor on the computer. Abstractly, these things should be perfectly equivalent.

But they aren't.

These kinds of activities utilize very different pathways in the brain—pathways that are affected by your very thoughts and how you think them. Your thoughts are not disconnected from the rest of the brain machinery or your body; *it's all connected*. This is just one example (and we'll talk more about the brain later in the book), but it helps illustrate the importance of thinking about interacting systems.

In his seminal book *The Fifth Discipline: The Art & Practice of the Learning Organization* [Sen94], Peter Senge popularized the term *systems thinking* to describe a different approach of viewing the world. In systems thinking, one tries to envision an object as a connection point of several systems, rather than as a discrete object unto itself.

Everything is interconnected.

For instance, you might consider a tree to be a single, discrete object sitting on the visible ground. But in fact, a tree is a connection of at least two major systems: the processing cycle of leaves and air and of roots and earth. It's not static; it's not isolated. And even more interesting, you'll rarely be a simple observer of a system. More likely, you'll be part of it, whether you know it or not.[3]

Recipe 1 Always consider the context.

Put a copy of that up on your wall or your desktop, in your conference room, on your whiteboard, or anywhere you think alone or with others. We'll be returning to it.

Everyone Is Talking About This Stuff

As I was mulling over the idea of writing this book, I started to notice that a *lot* of people in different disciplines were talking about the topics in which I was interested. But these were in very different and diverse areas, including the following:

- MBA and executive-level training
- Cognitive science research
- Learning theory
- Nursing, health care, aviation, and other professions and industries
- Yoga and meditative practices
- Programming, abstraction, and problem solving
- Artificial intelligence research

When you start to find the same set of ideas—the same common threads—showing up in different guises in these very different areas, that's usually a sign. There must be something fundamental and very important lurking under the covers for these similar ideas to be present in so many different contexts.

There's something fundamental here.

Yoga and meditative techniques seem to be enjoying quite a bit of mainstream popularity these days, and not always for obvious reasons. I noticed an article in an in-flight magazine around October 2005 that trumpeted the headline "Companies Now Offering Yoga and Meditation to Help Fight Rising Health-Care Costs."

Large companies have not historically embraced such warm-and-fuzzy activities. But the meteoric rise of health-care costs has forced them to take *any* course of action that might help. Clearly, they believe the studies showing that practitioners of yoga and meditative techniques enjoy greater overall health than the general population. In this book, we're more interested in the areas related to cognition, but greater overall health is a nice side benefit.

I also noticed that a number of MBA and executive-level courses promote various meditative, creative, and intuitive techniques—stuff that fits in perfectly with the available research but that has not yet been passed down to the employees in the trenches, including us knowledge-worker types.

But not to worry, we'll be covering these topics here for you. No MBA required.

Where We're Going

Every good journey begins with a map, and ours appears in the front portion of this book. Despite the linear flow of a book, these topics are entwined and interrelated, as the map shows.

After all, everything is connected to everything else. But it's somewhat difficult to appreciate that idea with a linear read of a book. You can't always get a sense of what's related when faced with countless "see also" references in the text. By presenting the map graphically, I hope you get the opportunity to see what's related to what a little more clearly.

With that in mind, the following is roughly where we are headed, despite a few side trips, tangents, and excursions on the way.

Journey from Novice to Expert

In the first part of the book, we'll look at *why* your brain works as it does, beginning with a popular model of expertise.

The Dreyfus model of skill acquisition provides a powerful way of looking at how you move beyond beginner-level performance and begin the journey to mastery of a skill. We'll take a look at the Dreyfus model and in particular look at the keys to becoming an expert: harnessing and applying your own experience, understanding *context*, and harnessing *intuition*.

This Is Your Brain

The most important tool in software development is, of course, your own brain. We'll take a look at some of the basics of cognitive science and neuroscience as they relate to our interests as software developers, including a model of the brain that looks a lot like a dual-CPU, shared-bus design and how to do your own brain surgery of a sort.

Get in Your Right Mind

Once we have a better understanding of the brain, we will find ways to exploit underutilized facets of thinking to help encourage better creativity and problem solving, as well as harvest and process experiences more effectively.

We'll also take a look at where *intuition* comes from. Intuition, the hallmark of the expert, turns out to be a tricky beast. You need it, you rely on it, but you also probably fight against using it constantly, without knowing why. You may also be actively suspicious of your own and others' intuition, mistakenly thinking that it's "not scientific."

We'll see how to fix that and give your intuition freer reign.

Debug Your Mind

Intuition is a fantastic skill, except when it's wrong. There are a large number of "known bugs" in human thinking. You have built-in biases in your cognition, influences from when you're born and from your cohort (those born about the same time as you), your innate personality, and even hardware wiring problems.

These bugs in the system often mislead you by clouding your judgment and steering you toward bad, even disastrous, decisions.

Knowing these common bugs is the first step to mitigating them.

Learn Deliberately

Now that we've gotten a good look at how the brain works, we'll start taking a more deliberate look at *how* to take advantage of the system, beginning with learning.

Note that I mean *learning* in the broadest sense, covering not only new technologies, programming languages, and the like, but also your learning of the dynamics of the team you're on, the characteristics of the evolving software you're building, and so on. In these times, we have to learn all the time.

But most of us have never been taught how, so we sort of wing it as best we can. I'll show you some specific techniques to help improve your learning ability. We'll look at planning techniques, mind maps, a reading technique known as SQ3R, and the cognitive importance of teaching and writing. Armed with these techniques, you can absorb new information faster and easier, gain more insights, and retain this new knowledge better.

Gain Experience

Gaining experience is key to your learning and growth—we learn best by doing. However, just "doing" alone is no guarantee of success; you have to learn from the doing for it to count, and it turns out that some common obstacles make this hard.

You can't force experience either; trying too hard can be just as bad (if not worse) than slogging through the same old motions. We'll take a look at what you need to create an efficient learning environment using feedback, fun, and failure; see the dangers of deadlines; and see how to gain experience virtually with mental grooving.

Manage Focus

Managing your attention and focus is the next critical step in your journey. I'll share with you some tricks, tips, and pointers to help you manage the flood of knowledge, information, and insights that you need to gain experience and learn. We live in information-rich times, and it's easy to get so swamped under the daily demands of our jobs that we have no chance to advance our careers. Let's try to fix that and increase your attention and focus.

We'll take a look at how to optimize your current context, manage those pesky interruptions better, and see why interruptions are such cognitive train wrecks. We'll look at why you need to defocus in order to focus better in the mental marinade and manage your knowledge in a more deliberate manner.

Beyond Expertise

Finally, we'll take a quick look at why change is harder than it looks, and I'll offer suggestions for what you can do tomorrow morning to get started.

I'll share what I think lies beyond expertise and how to get there.

So, sit back, grab your favorite beverage, and let's take a look at what's under the hood.

⬇ Next Actions

Throughout the book, I'll suggest "next actions" that you can take to help reinforce and make this material real for you. These might include exercises to do, experiments to try, or habits to start. I'll list these using checkboxes so you can check the items you've done, like this:

- Take a hard look at current problems on your project. Can you spot the different systems involved? Where do they interact? Are these interaction points related to the problems you're seeing?
- Find three things you've analyzed out of context that caused you problems later.
- Put up a sign somewhere near your monitor that reads "Consider the context."

About the Figures

You may notice that figures in this book don't look like the typical shiny, mechanically perfect drawings you'd expect from Adobe Illustrator or something similar. That's quite deliberate.

From the electronics books by Forrest M. Mims III to the back-of-the-napkin design documents favored by agile developers, hand-drawn figures have certain unique properties, and we'll see why a bit later in the book.

Grateful Acknowledgments

Very special thanks to Ellie Hunt for introducing me to the Dreyfus model and related nursing research, suffering through my disjointed and rambling prose, keeping me on track, and keeping our domestic enterprises running like a well-oiled machine. A regular editor's job is often difficult and thankless, and mere appreciation in a preface really doesn't do it justice. To be editor, mom, and business manager all at once truly takes skill and patience.

Thanks to my friends on the Pragmatic Wetware mailing list and reviewers, including Bert Bates, Don Gray, Ron Green, Shawn Hartstock, Dierk Koenig, Niclas Nilsson, Paul Oakes, Jared Richardson, Linda Rising, Johanna Rothman, Jeremy Sydik, Steph Thompson, Derek Wade, and everyone else who posted their thoughts, experiences, and readings. Their combined experiences are invaluable.

Special thanks to June Kim for his many contributions throughout the book, including pointers to far-flung research and stories of his own experiences, as well as his feedback throughout the stages of birthing this book.

Special thanks also to Dr. Patricia Benner, who introduced the Dreyfus model of skills acquisition to the nursing profession, for her support and permission to quote from her works and for her enthusiasm for learning.

Thanks go to Dr. Betty Edwards, who pioneered the practical applications of lateral specialization in the brain, for her kind support and permission to quote from her works.

Thanks to Sara Lynn Eastler for the index, to Kim Wimpsett for correcting my many typos and often haphazard grammar, and to Steve Peter for implementing a plethora of typesetting tricks.

And finally, thanks to *you* for purchasing this book and beginning the journey with me.

Let’s move our profession forward in the right direction, harness our experience and intuition, and create new environments where learning matters.

Footnotes

[1] Based on research by Capers Jones via Bob Binder.

[2] That is, platforms.

[3] Suggested by our old buddy Heisenberg and his quantum uncertainty principle, the more general *observer effect* posits that you can’t observe a system without altering it.

We can't solve problems by using the same kind of thinking we used when we created them.

Albert Einstein

Chapter 2

Journey from Novice to Expert

Wouldn't you like to be the expert? To intuitively *know* the right answer? This is the first step of our journey together along that road. In this chapter, we'll look at what it means to be a novice and what it means to be an expert—and all the stages in between. Here's where our story begins.

Once upon a time, two researchers (brothers) wanted to advance the state of the art in artificial intelligence. They wanted to write software that would learn and attain skills in the same manner that humans learn and gain skill (or prove that it couldn't be done). To do that, they first had to study how humans learn.

They developed the *Dreyfus model* of skill acquisition,[4] which outlines five discrete stages through which one must pass on the journey from novice to expert. We'll take a look at this concept in depth; as it turns out, we're not the first ones to use it effectively.

Event Theories vs. Construct Theories

The Dreyfus model is what's called a *construct* theory. There are two types of theories: *event* theories and *construct* theories.[5] Both are used to explain some phenomenon that you've observed.

Event theories can be measured; these types of theories can be verified and proven. You can judge the accuracy of an event theory.

Construct theories are intangible abstractions; it makes no sense to speak of "proving them." Instead, construct theories are evaluated in terms of their usefulness. You can't judge a construct theory to be accurate or not. That's mixing apples and existentialism. An apple is a thing; existentialism is an abstraction.

For instance, I can prove all sorts of things about your brain using simple electricity or complex medical imaging devices. But I can't even prove you have a mind. *Mind* is an abstraction; there's really no such thing. It's just an idea, a concept. But it's a very useful one.

The Dreyfus model is a construct theory. It's an abstraction, and as we'll see, it's a very useful one.

Back in the early 1980s, the nursing profession in the United States used the lessons of the Dreyfus model to correct their approach and help advance their profession. At the time, the problems faced by nurses mirrored many of the same problems programmers and engineers face today. Their profession has made great progress, and in the meantime we still have some work to do with ours.

Here are some observations that ring true for both nurses and programmers, and probably other professions as well:

- Expert staff members working in the trenches aren't always recognized as experts or paid accordingly.
- Not all expert staff want to end up as managers.
- There's a huge variance in staff members' abilities.
- There's a huge variance in managers' abilities.
- Any given team likely has members at widely different skill levels and

can't be treated as a homogeneous set of replaceable resources.

There's more to skill levels than just being better, smarter, or faster. The Dreyfus model describes how and why our abilities, attitudes, capabilities, and perspectives change according to skill level.

It helps explain why many of the past approaches to software development improvement have failed. It suggests a course of action that we can pursue in order to meaningfully improve the software development profession—both as individual practitioners and for the industry as a whole.

Let's take a look.

Novices vs. Experts

What do you call an expert software developer? A *wizard*. We work with magic numbers, things in hex, zombie processes, and mystical incantations such as `tar -xzvf plugh.tgz` and `sudo gem install --include-dependencies rails`.

Figure 1. A Unix wizard

We can even change our identity to become someone else or transform into the root user—the epitome of supreme power in the Unix world. Wizards make it look effortless. A dash of eye of newt, a little bat-wing dust, some incantations, and poof! The job is done. Despite the mythological overtones, this vision is fairly common when considering an expert in any particular field (ours is just arcane enough to make it a really compelling image).

Making It Look Easy

I once was in a position to interview professional organists. For an audition piece, I chose Charles-Marie Widor's "Toccata" (from Symphony No. 5 in F Minor, Op. 42 No. 1, for those who care about such things), a frenetic piece that sounded suitably difficult to my amateur ears.

One candidate really worked it—both feet flying on the pedals, hands running up and down both ranks of the organ in a blur, a stern look of intense concentration across her brow. She was practically sweating. It was a terrific performance, and I was suitably impressed.

But then came along the true expert. She played this difficult piece a little bit better, a little bit faster, but was smiling and *talking* to us while her hands and feet flew in an octopus-like blur.

She made it look easy, and she got the job.

Consider the expert chef, for instance. Awash in a haze of flour, spices, and a growing pile of soiled pans left for an apprentice to clean, the expert chef may have trouble articulating just *how* this dish is made. "Well, you take a bit of this and a dash of that—not too much—and cook until done."

Chef Claude is not being deliberately obtuse; he knows what "cook until done" means. He knows the subtle difference between just enough and "too much" depending on the humidity, where the meat was purchased, and how fresh the vegetables are.

It's often difficult for experts to explain their actions to a fine level of detail; many of their responses are so well practiced that they become preconscious actions. Their vast experience is mined by nonverbal, preconscious areas of the brain, which makes it hard for us to observe and hard for them to articulate.

It’s hard to articulate expertise.

When experts do their thing, it appears almost magical to the rest of us—strange incantations, insight that seems to appear out of nowhere, and a seemingly uncanny ability to know the right answer when the rest of us aren’t even all that sure about the question.

It’s not magic, of course, but the way that experts perceive the world, how they problem solve, the mental models they use, and so on, are all markedly different from nonexperts.

A novice cook, on the other hand, coming home after a long day at the office is probably not even interested in the subtle nuances of humidity and parsnips. The novice wants to know *exactly* how much saffron to put in the recipe (not just because saffron is ridiculously expensive).

The novice wants to know *exactly* how long to set the timer on the oven given the weight of the meat, and so on. It’s not that the novice is being pedantic or stupid; it’s just that novices need clear, context-free rules by which they can operate, just as the expert would be rendered ineffective if he were constrained to operate under those same rules.

Novices and experts are fundamentally different. They see the world in different ways, and they react in different ways. Let’s look at the details.

The Five Dreyfus Model Stages

In the 1970s, the brothers Dreyfus (Hubert and Stuart) began doing their seminal research on how people attain and master skills.

Dreyfus is applicable per skill.

The Dreyfus brothers looked at highly skilled practitioners, including commercial airline pilots and world-renowned chess masters.[6] Their research showed that quite a bit changes as you move from novice to expert. You don't just "know more" or gain skill. Instead, you experience fundamental differences in how you perceive the world, how you approach problem solving, and the mental models you form and use. How you go about acquiring new skills changes. External factors that help your performance—or hinder it—change as well.

Unlike other models or assessments that rate the whole person, the Dreyfus model is applicable per skill. In other words, it's a situational model and not a trait or talent model.

You are neither "expert" nor "novice" at all things; rather, you are at one of these stages in some particular skill domain. You might be a novice cook but an expert sky diver, or vice versa. Most nondisabled adults are experts at walking—we do so without planning or thinking. It has become instinct. Most of us are novices at tax preparation. We can get through it given a sufficient number of clear rules to follow, but we really don't know what's going on (and wonder why on Earth those rules are so arcane).

The following are the five stages on the journey from novice to expert.

Stage 1: Novices

Novices, by definition, have little or no previous experience in this skill area. By "experience," I mean specifically that performing this skill results in a change of thinking. As a counterexample, consider the case of the developer who claims ten years of experience, but in reality it was one year of experience repeated nine times. That doesn't count as *experience*.

Novices are very concerned about their ability to succeed; with little experience to guide them, they really don't know whether their actions will all turn out OK. Novices don't particularly want to learn; they just want to accomplish an immediate goal. They do not know how to respond to mistakes and so are fairly vulnerable to confusion when things go awry.

They can, however, be somewhat effective if they are given context-free rules to follow, that is, rules of the form "Whenever X happens, do Y." In other words, they need a recipe.

> *Novices need recipes.*

This is why call centers work. You can hire a large number of folks who don't have a lot of experience in the subject matter at hand and let them navigate a decision tree.

A giant computer hardware company might use a script like this:

1. Ask the user whether the computer is plugged in.
2. If yes, ask whether the computer is powered on.
3. If no, ask them to plug it in and wait.
4. *and so on...*

It's tedious, but fixed rules such as these can give novices some measure of capability. Of course, novices face the problem of not knowing *which* rules are most relevant in a given situation. And when something unexpected comes up, they will be completely flummoxed.

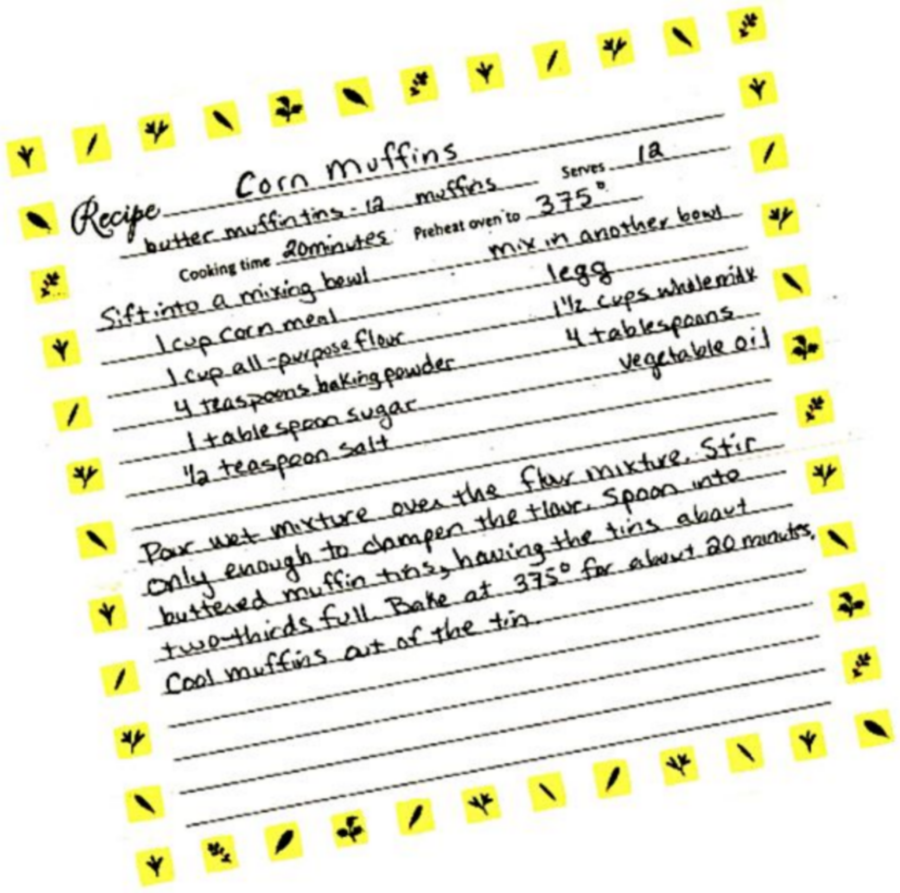
As with most people, I am a novice when it comes to doing my taxes. I have little experience; despite having filed taxes for more than twenty-five years, I haven't learned anything or changed my thinking about it. I don't want to learn; I just want to accomplish the goal—to get them filed this year. I don't know how to respond to mistakes; when the IRS sends me a terse and rather arrogant form letter, I usually have no idea what they're on about or what to do to fix it.[7]

There is a solution, of course. A context-free rule to the rescue! Perhaps it's something such as the following:

- Enter the amount of money you earned last year.
- Send it in to the government.

That's simple and unambiguous.

The problem with recipes—with context-free rules—is that you can never specify everything fully. For instance, in the corn muffin recipe, it says to cook for "about 20 minutes." When do I cook longer? Or shorter? How do I know when it's done? You can set up more rules to explain, and then more rules to explain those, but there's a practical limit to how much you can effectively specify without running into a Clinton-esque "It depends upon what the meaning of the word *is* is." This phenomenon is known as *infinite regression*. At some point, you have to stop defining explicitly.



Recipe Corn Muffins
Serves 12
butter muffin tins - 12 muffins
Preheat oven to 375°
Cooking time 20minutes

Sift into a mixing bowl
1 cup corn meal
1 cup all-purpose flour
4 teaspoons baking powder
1 tablespoon sugar
½ teaspoon salt

mix in another bowl
1egg
1½ cups whole milk
4 tablespoons
vegetable oil

Pour wet mixture over the flour mixture. Stir only enough to dampen the flour. Spoon into buttered muffin tins, having the tins about two-thirds full. Bake at 375° for about 20 minutes. Cool muffins out of the tin.

Figure 2. Recipe for corn muffins. But how long do you cook it?

Rules can get you started, but they won't carry you further.

Stage 2: Advanced Beginners

Once past the hurdles of the novice, one begins to see the problems from the viewpoint of the *advanced beginner.* Advanced beginners can start to break away from the fixed rule set a little bit. They can try tasks on their own, but they still have difficulty troubleshooting.

They want information fast. For instance, you may feel like this when you're learning a new language or API and you find yourself scanning through the documentation quickly looking for that one method signature or set of arguments. You don't want to be bogged down with lengthy theory at this point or spoon-fed the basics yet again.

Advanced beginners don't want the big picture.

Advanced beginners can start using advice in the correct context, based on similar situations they've experienced in the recent past but just barely. And although they can start formulating some overall principles, there is no "big picture." They have no holistic understanding and really don't want it yet. If you tried to force the larger context on an advanced beginner, they would probably dismiss it as irrelevant.

You might see this sort of reaction when the CEO calls an all-hands meeting and presents charts and figures showing sales projections and such. Many of the less experienced staff will tend to dismiss it as not being relevant to their individual job.

Of course, it is very relevant and can help determine whether you'll still *have* a job with this company next year. But you won't see the connection while you're at the lower skill levels.

Stage 3: Competent

At the third stage, practitioners can now develop conceptual models of the problem domain and work with those models effectively. They can troubleshoot problems on their own and begin to figure out novel problems—ones they haven't faced before. They can begin to seek out and apply advice from experts and use it effectively.

Competents can troubleshoot.

Instead of following the sort of knee-jerk response of the previous levels, the *competent* practitioner will seek out and solve problems; their work is based more on deliberate planning and past experience. Without more experience, they'll still have trouble trying to determine which details to focus on when problem solving.

You might see folks at this level typically described as "having initiative" and being "resourceful." They tend to be in a leadership role in the team (whether it's formal or not).[8] These are great folks to have on your team. They can mentor the novices and don't annoy the experts overly much.

In the field of software development, we're getting there, but even at this level, practitioners can't apply agile methods the way we would like—there isn't yet enough ability for reflection and self-correction. For that, we need to make a breakthrough to the next level: proficient.

Stage 4: Proficient

Proficient practitioners need the big picture. They will seek out and want to understand the larger conceptual framework around this skill. They will be very frustrated by oversimplified information.

For instance, someone at the proficient stage will not react well when they call the tech support hotline and are asked whether it's plugged in. (Personally, I want to reach through the phone and remove the first vital organ that presents itself in these situations.)

Proficient practitioners make a major breakthrough on the Dreyfus model: they can correct previous poor task performance. They can reflect on how they've done and revise their approach to perform better the next time. Up until this stage, that sort of self-improvement is simply not available.

Proficient practitioners can self-correct.

Also, they can learn from the experience of others. As a proficient practitioner, you can read case studies, listen to water-cooler gossip of failed projects, see what others have done, and learn effectively from the story, even though you didn't participate in it firsthand.

Along with the capacity to learn from others comes the ability to understand and apply *maxims*, which are proverbial, fundamental truths that can be applied to the situation at hand.[9] Maxims are not recipes; they have to be applied within a certain context.

For instance, a well-known maxim from the extreme programming methodology tells you to "test everything that can possibly break."

To the novice, this is a recipe. What do I test? All the setter and getter methods? Simple print statements? They'll end up testing irrelevant things.

But the proficient practitioner knows what can possibly break—or more correctly, what is likely to break. They have the experience and the judgment to understand what this maxim means *in context*. And context, as it turns out, is key to becoming an expert.

Pragmatic Tips

When Dave Thomas and I wrote the original *The Pragmatic Programmer*, we were trying to convey some of the advice we thought was most relevant to our profession.

These tips—these maxims—were a reflection of our collective years of expertise. From the mind-expanding practice of learning a new language every year to the hard-won principles of Don't Repeat Yourself (DRY) and No Broken

Windows, maxims such as these are key to transferring expertise.

Proficient practitioners have enough experience that they know—from experience—what's likely to happen next; and when it doesn't work out that way, they know what needs to change. It becomes apparent to them which plans need to be discarded and what needs to be done instead.

Similarly, software Patterns (as espoused in *Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV95], also known as the Gang of Four book) can be effectively applied by proficient-level practitioners (but not necessarily at lower skill levels; see the sidebar *Misapplied Patterns and Fragile Methods*).

Now we're getting somewhere. Proficient practitioners can take full advantage of the reflection and feedback that is core to agile methods. This is a big leap from the earlier stages; someone at the proficient stage is much more like a junior expert than a really advanced competent.

Misapplied Patterns and Fragile Methods

As you may realize by now, some of the most exciting new movements in the software development community are targeted at proficient and expert developers.

Agile development relies on feedback; in fact, my definition of *agile development* from *Practices of an Agile Developer* [SH06] says this: "Agile development uses feedback to make constant adjustments in a highly collaborative environment." But being able to self-correct based on previous performance is possible only at the higher skill levels.

Advanced beginners and competent practitioners often confuse software design patterns with recipes, sometimes with disastrous results. For instance, I once knew a developer on a project who had just been exposed to the Gang of Four (GoF) book. In his enthusiasm, he wanted to start using design patterns. All of them. At once. In a small piece of report-writing code.

He managed to jam in about seventeen of the twenty-three GoF patterns into this hapless piece of code before someone noticed.

Stage 5: Expert

Finally, at the fifth stage, we come to the end of the line: the expert.

Experts are the primary sources of knowledge and information in any field. They are the ones who continually look for better methods and better ways of doing things. They have a vast body of experience that they can tap into and apply in just the right context. These are the folks who write the books, write the articles, and do the lecture circuit. These are the modern wizards.

Statistically, there aren't very many experts—probably something on the order of 1 to 5 percent of the population.[10]

Experts work from *intuition*, not from reason. This has some very interesting ramifications and raises some key questions—what is intuition, anyway? (We'll delve more into the details of intuition throughout the book.)

Experts work from intuition.

Although experts can be amazingly intuitive—to the point that it looks like magic to the rest of us—they may be completely inarticulate as to how they arrived at a conclusion. They genuinely don't know; it just "felt right."

For instance, suppose a physician looks in at a patient. At a glance, the doctor says, "I think this patient has Blosen-Platt syndrome; better run these tests." The staff runs the tests, and indeed, the doctor is correct. How did she know? Well, you could ask, but the doctor may well reply with "He didn't look right."

Indeed, the patient just didn't look "right." Somehow, in the vast array of experiences, distilled judgment, memories, and all the rest of the mental effluvia in the doctor's brain, a particular combination of subtle clues in the patient came together and suggested a diagnosis. Maybe it was the skin pallor or the way the patient was slumped over—who knows?

The expert does. The expert knows the difference between irrelevant details and the very important details, perhaps not on a conscious level, but the expert knows which details to focus on and which details can be safely ignored. The expert is very good at targeted, focused pattern matching.

Unskilled and Unaware of It

When you are not very skilled in some area, you are more likely to think you're actually pretty expert at it.

In the paper *Unskilled and Unaware of It: How Difficulties in Recognizing One's Own Incompetence Lead to Inflated Self-Assessments* [KD99], psychologists Kruger and Dunning relate the unfortunate story of a would-be thief who robbed a bank in broad daylight. He was incredulous at his prompt arrest, because he was under the impression that wearing lemon juice on your face would make you invisible to security cameras.

The "lemon juice man" never suspected that his hypothesis was, er, suspect. This lack of accurate self-assessment is referred to as *second-order incompetence*, that is, the condition of being unskilled and unaware of it.

This condition is a huge problem in software development, because many programmers and managers aren't aware that better methods and practices even exist. I've met many younger programmers (one to five years of experience) who *never* have been on a successful project. They have already succumbed to the notion that a normal project should be painful and should fail.

Charles Darwin pegged it when he said, "Ignorance more frequently begets confidence than does knowledge."

The converse seems to be true as well; once you truly become an expert, you become painfully aware of just how little you really know.

Dreyfus at Work: Herding Racehorses and Racing Sheep

Now that we've looked at the Dreyfus model in detail, let's see how to apply the Dreyfus lessons at work. In software development at least, it turns out that we tend to apply them pretty poorly.

Experts aren't perfect. They can make mistakes just like anyone else, they are subject to the same cognitive and other biases that we'll look at later (in Chapter 5, *Debug Your Mind*), and they will also likely disagree with one another on topics within their field.

But worse than that, by misunderstanding the Dreyfus model, we can rob them of their expertise. It's actually easy to derail an expert and ruin their performance. All you have to do is force them to follow the rules.

In one of the Dreyfus studies, the researchers did exactly that. They took seasoned airline pilots and had them draw up a set of rules for the novices, representing their best practices. They did, and the novices were able to improve their performance based on those rules.

Rules ruin experts.

But then they made the experts follow their own rules.

It degraded their measured performance significantly.[11]

This has ramifications for teamwork as well. Consider any development methodology or corporate culture that dictates iron-clad rules. What impact will that have on the experts in the team? It will drag their performance down to the level of the novice. You lose all competitive advantage of their expertise.

But the software industry as a whole tries to "ruin" experts in this fashion all the time. You might say that we're trying to herd racehorses. That's not how you get a good return on investment in a racehorse; you need to let them run.[12]

Intuition is the tool of the expert in all fields, but organizations tend to discount it because they mistakenly feel that intuition "isn't scientific" or "isn't repeatable." So, we tend to throw out the baby with the bathwater and don't listen to the experts to whom we pay so much.

Conversely, we also tend to take novices and throw them in the deep end of the development pool—far over their heads. You might say we're trying to race sheep, in this case. Again, it's not an effective way to use novices. They need to be "herded," that is, given unambiguous direction, quick successes, and so on. Agile development is a very effective tool, but it won't work on a team composed solely of novices and advanced beginners.

But forces in the industry conspire against us in both directions. A misguided sense of political correctness dictates that we treat all developers the same, regardless of ability. This does a disservice to both novices and experts (and ignores the reality that there is anywhere from a 20:1 to 40:1 difference in productivity among developers, depending on whose study you believe).[13]

Work to Rule

In industries or situations where one is not allowed a full-blown strike, a work slowdown is often used as a means of demonstration.

Often this is called *work to rule* or *malicious obedience*, and the idea is that the employees do *exactly* what their job description calls for—no more, no less—

and follow the rule book to the letter.

The result is massive delays and confusion—and an effective labor demonstration. No one with expertise in the real world follows the rules to the letter; doing so is demonstrably inefficient.

According to Benner (in *From Novice to Expert: Excellence and Power in Clinical Nursing Practice* [Ben01]), “Practices can never be completely objectified or formalized because they must ever be worked out anew in particular relationships and in real time.”

Recipe 2 **Use rules for novices, intuition for experts.**

The journey from novice to expert involves more than just rules and intuition, of course. Many characteristics change as you move up the skill levels. But the three most important changes along the way are the following:[14]

- Moving away from reliance on rules to intuition
- A change in perception, where a problem is no longer a collection of equally relevant bits but a complete and unique whole where only certain bits are relevant
- Finally, a change from being a detached observer of the problem to an involved part of the system itself

This is the progression from novice to expert, away from detached and absolute rules and into intuition and (remember systems thinking?) eventually part of the system itself (see the following figure).

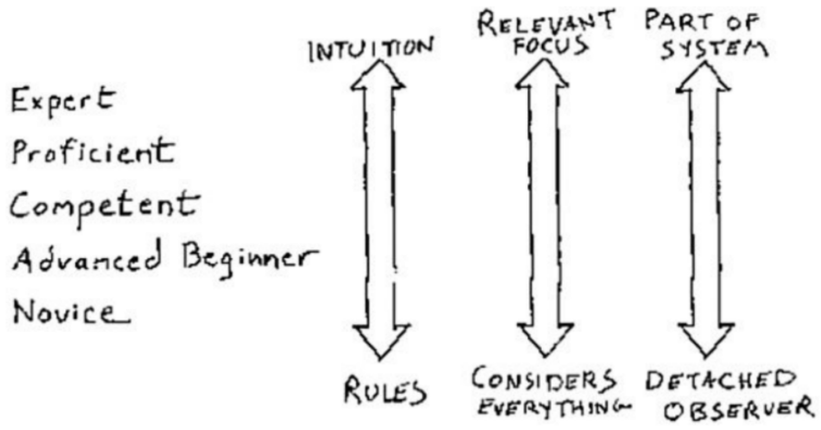


Figure 3. Dreyfus model of skill acquisition

The Sad Fact of Skill Distribution

Now at this point you're probably thinking that the great bulk of people fall smack in the middle—that the Dreyfus model follows a standard distribution, which is a typical bell curve.

It does not.

Sadly, studies seem to indicate that most people, for most skills, for most of their lives, never get any higher than the second stage, advanced beginner, "performing the tasks they need and learning new tasks as the need arises but never acquiring a more broad-based, conceptual understanding of the task environment."[15] A more accurate distribution is shown in the figure here:

Most people are advanced beginners.

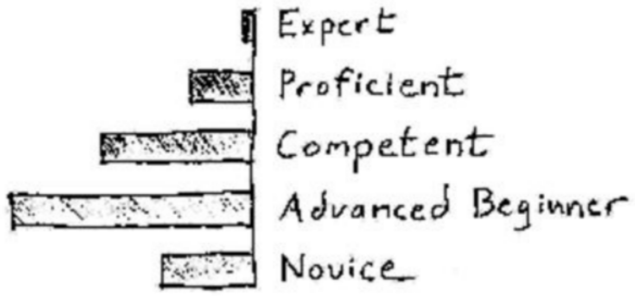


Figure 4. Skill distribution

Anecdotal evidence for the phenomenon abounds, from the rise of copy-and-paste coding (now using Google as part of the IDE) to the widespread misapplication of software design patterns.

Also, *metacognitive* abilities, or the ability of being self-aware, tends to be possible only at the higher skill levels. Unfortunately, this means practitioners at the lower skill levels have a marked tendency to overestimate their own abilities—by as much as 50 percent, as it turns out. According to a study in *Unskilled and Unaware of It: How Difficulties in Recognizing One's Own Incompetence Lead to Inflated Self-Assessments* [KD99], the only path to a more correct self-assessment is to improve the individual's skill level, which in turn increases metacognitive ability.

You may see this referred to as *second-order incompetence*, not knowing just how much it is that you don't know. The beginner is confident despite the odds; the expert will be far more cautious when the going gets weird. Experts will show much more self-doubt.

Recipe 3 Know what you don't know.

Unfortunately, we'll always have more advanced beginners than experts. But even though it is weighted at the bottom, it's still a distribution. If you're lucky enough to have an expert on your team, you need to accommodate them. Similarly, you need to accommodate the few novices, the many advanced beginners, and the small but powerful number of competent and proficient practitioners.

Expert != Teacher

Experts aren't always the best teachers. Teaching is an expertise in its own right; just because you are expert in some subject is no guarantee that you can teach it to others.

Also, given the phenomenon that experts are often unable to articulate why they reached a particular decision, you may find that someone at a competent level might be in a better position to teach a novice than an expert would be. When pairing or mentoring within the team, you might try using mentors who are closer in skill level to the trainee.

The hallmark of the expert is their use of intuition and the ability to recognize patterns in context. That's not to say that novices have zero intuition or that competents can't recognize patterns at all but that the expert's intuition and pattern recognition now take the place of explicit knowledge.

Intuition and pattern matching replace explicit knowledge.

This transition from the novice's context-free rules to the expert's context-dependent intuition is one of the most interesting parts of the Dreyfus model; so our goal, for most of the rest of this book, is to see how we might better harness intuition and get better at recognizing and applying patterns.[16]

Ten Years to Expertise?

So, you want to be an expert? You need to budget about ten years of effort, regardless of the subject area. Researchers[17] have studied chess playing, music composition, painting, piano playing, swimming, tennis, and other skills and disciplines. In virtually every case, from Mozart to the Beatles, you find evidence of a minimum of a decade of hard work before world-class expertise shows up.

The Beatles, for instance, took the world by storm in 1964 with a landmark appearance on the *Ed Sullivan Show*. Their first critically successful album, *Sgt. Pepper's Lonely Hearts Club Band*, was released shortly after, in 1967. But the band didn't just magically form for a tour in 1964—they had been playing in clubs since 1957. Ten years before *Sgt. Pepper's*.

And hard work it is—merely working at a subject for ten years isn't enough. You need to *practice*. Deliberate practice, according to noted cognitive scientist Dr. K. Anderson Ericsson, requires four conditions:

- You need a well-defined task.
- The task needs to be appropriately difficult—challenging but doable.
- The environment needs to supply informative feedback that you can act on.
- It should also provide opportunities for repetition and correction of errors.

Do that sort of practice, steadily, for ten years, and you've got it made. As we noted in *The Pragmatic Programmer: From Journeyman to Master* [HT00], even Chaucer complained that "the lyfe so short, the craft so long to lerne."

However, there is some good news. Once you become an expert in one field, it becomes much easier to gain expertise in another. At least you already have the acquisition skills and model-building abilities in place.

Thanks to June Kim for the reference to Dr. Ericsson.

Using the Dreyfus Model Effectively

By the late 1970s or so, the nursing profession was in dire straits. In a nutshell, these were their problems, which I've drawn from several case studies and narratives:[18]

- Nurses themselves were often disregarded as a mere commodity; they just carried out the highly trained doctor's orders and weren't expected to have any input on patient care.

- Because of pay-scale inequities, expert nurses were leaving direct patient care in droves. There was more money to be made in management, teaching, or the lecture circuit.

- Nursing education began to falter; many thought that formal models of practice were the best way to teach. An overreliance on formal methods and tools eroded real experience in practice.

- Finally, they had lost sight of the real goal—patient outcomes. Despite whatever process and methodology you followed, despite who worked on this patient, what was the outcome? Did the patient live and thrive? Or not?

If you read that list carefully, you may have noticed that these problems sound eerily familiar. Allow me to slightly edit this bullet list to reflect software development:

- Coders themselves were often disregarded as a mere commodity; they just carried out the highly trained analysts' orders and weren't expected to have any input on the design and architecture of the project.

- Because of pay-scale inequities, expert programmers were leaving hands-on coding in droves. There was more money to be made in management, teaching, or the lecture circuit.

- Software engineering education began to falter; many thought that formal models of practice were the best way to teach. An overreliance on formal methods and tools eroded real experience in practice.

- Finally, they had lost sight of the real goal—project outcomes. Despite whatever process and methodology you followed, despite who worked on this project, what was the outcome? Did the project succeed and thrive? Or not?

Huh. It sounds a little more familiar that way; indeed, these are serious problems that our industry now faces.

Back in the early 1980s, nursing professionals began to apply the lessons of the Dreyfus model to their industry with remarkable results. Dr. Benner's landmark book exposed and explained the Dreyfus model so that all involved parties had a better understanding of their own skills and roles and those of their co-workers. It laid out specific guidelines to try to improve the profession as a whole.

Over the course of the next twenty-five years or so, Benner and subsequent authors and researchers turned their profession around.

So in the best spirit of R&D (which stands for "Rip off and Duplicate"), we can borrow many lessons from their work and apply them to software development. Let's take a closer look at how they did it and what we can do in our own profession.

Accepting Responsibility

Twenty-five years ago, nurses were expected to follow orders without question, even vehemently—and proudly—maintaining that they "never veer from doctor's orders," despite obvious changes in patients' needs or conditions.

This attitude was enculturated in part by the doctors, who weren't in a position to see the constant, low-level changes in patients' conditions, and in part by the nurses themselves, who willingly abdicated responsibility for decision making in the course of practice to the authority of the doctors. It was professionally safer for them that way, and indeed there is some psychological basis for their position.

In one experiment,[19] a researcher calls a hospital ward posing as a doctor and orders the nurse to give a particular medication to a given patient. The order was rigged to trigger several alarm bells:

- The prescription was given over the phone, not in writing.
- The particular medication was not on the ward's usual approved list.
- According to labels on the medication itself, the prescribed dosage was double the maximum amount.
- The "doctor" on the phone was a stranger, not known to the nurse or staff.

But despite these clear warning signs, 95 percent of the nurses fell for it and went straight to the medicine cabinet, en route to the patient's room to dose 'em up.

Fortunately, they were stopped by an accomplice who explained the experiment—and stopped them from carrying out the bogus order.[20]

We see very much the same problems with programmers and their project managers or project architects. Feedback from coders to those who define architecture, requirements, and even business process has traditionally been either lacking entirely, brutally rejected, or simply lost in the noise of the project. Programmers often implement something they *know* is wrong, ignoring the obvious warning signs much as the nurses did in this example. Agile methods help promote feedback from all members of the team and utilize it effectively, but that's only half the battle.

Individual nurses had to accept responsibility in order to make in-the-field decisions according to the unfolding dynamics of a particular situation; individual programmers must accept the same responsibility. The Nuremberg-style defense "I was only following orders" did not work in WWII, it did not work for the nursing profession, and it does not work in software development.

"I was just following orders!" doesn't work.

But in order to accomplish this change in attitude, we *do* need to raise the bar. Advanced beginners aren't capable of making these sorts of decision by themselves. We must take the advanced beginners we have and help them raise their skill levels to competent.

A major way to help achieve that is to have good exemplars in the environment; people are natural mimics (see *Learn About the Inner Game*). We learn best by example. In fact, if you have children, you may have noticed that they rarely do as you say but will always copy what you do.

Recipe 4 **Learn by watching and imitating.**

Trumpeter Clark Terry used to tell students the secret to learning music was to go through three phases:

- Imitate
- Assimilate
- Innovate

That is, first you imitate existing practice and then slowly assimilate the tacit knowledge and experience over time. Finally you'll be in a position to go beyond imitation and innovate. This echoes the cycle of training in the martial arts known as Shu Ha Ri.

In the *Shu* phase, the student copies the techniques as taught—from a single instructor—without modifications. In the *Ha* stage, the student must reflect on the meaning and purpose and come to a deeper understanding. *Ri* means to go beyond or transcend; no longer a student, the practitioner now offers original thought.

So, among other things, we need to look at ways to keep as much existing expertise as we can in the project itself; none of this progression will help if practitioners don't stay in the field.

No Expertise Without Experience

Jazz is an art form that relies heavily on real-world experience. You may learn all the chords and techniques required to play jazz, but you have to *play* it in order to get the "feel." The famous trumpet player and vocalist Louis "Satchmo" Armstrong said of jazz, "Man, if ya gotta ask, you'll never know."

There's no expertise without experience, and there's no substitute for experience—but we can work to make the experience you have more efficient and more effective.

Keeping Expertise in Practice

The nursing profession was losing expertise rapidly; because of the limits of pay scales and career development, nurses with high skill levels would reach a point in their careers where they were forced to move out of direct clinical practice and into areas of management or education or move out of the field entirely.

This largely remains the case in software development as well. Programmers (aka "coders") are paid only so much; salespeople, consultants, upper management, and so on, might be compensated more than twice the amount of the best programmer on a team.

Companies need to take a much closer, much more informed look at the value that these star developers bring to an organization.

For instance, many project teams use a sports metaphor to describe positive aspects of teamwork, a common goal, and so on. But in reality, our idealized view of teamwork doesn't match what really happens on professional sports teams.

Winners don't carry losers.

Two men may both play the position of pitcher for a baseball team, yet one may earn $25 million a year, and the other may earn only $50,000. The question isn't the position they play, or even their years of experience; the question is, what is the value they bring to the organization?

An article by Geoffrey Colvin[21] expands on this idea by noting that real teams have stars: not everyone on the team is a star; some are rookies (novices and advanced beginners), and some are merely competent. Rookies move up the ladder, but winners don't carry losers—losers get cut from the team. Finally, he notes that the top 2 percent isn't considered world-class. The top 0.2 percent is.

And it's not just high-pressure sports teams; even churches recognize difference in talent and try to use it effectively. Recently I was shown a national church's newsletter that offered advice on how to grow and maintain a music program. Their advice sounds very familiar:

- A group is only as good as its weakest link. Put the best performers together to perform for the main service, and create "farm teams" for other services.
- Keep a steady group with the same performers every week. You want the group to jell; rotating players in and out is counterproductive.
- Timing is everything: the drummer (for a band) or accompanist (for choral groups) has to be solid. Better to use a prerecorded accompaniment than a

flaky drummer or organist.

- Make your group a safe place for talented musicians, and watch what happens.

That's exactly the same thing you want on a software team.[22] This idea of providing the right environment for skilled developers is critical.

Given that the highest-skilled developers are orders of magnitude more productive than the least-skilled developers, the current common salary structures for developers is simply inadequate. Like the nursing profession years ago, we continually face the risk of losing a critical mass of expertise to management, competitors, or other fields.

This tendency is made worse by the recent increases in outsourcing and offshoring development to cheaper countries. It's an unfortunate development in that it further cements the idea in people's minds that coding is just a mechanical activity and can be sent away to the lowest bidder. It doesn't quite work that way, of course.

As in the nursing profession, experts at coding must continue to code and find a meaningful and rewarding career there. Setting a pay scale and a career ladder that reflects a top coder's value to the organization is the first step toward making this a reality.

Recipe 5 **Keep practicing in order to remain expert.**

Beware the Tool Trap

There has been much written on the role of tools, formal models, modeling, and so on, in software development. Many people claim that UML and model-driven architecture (MDA) are the future, just as many people claimed that RUP and CMM process models were the salvation of the industry.

But as with all silver-bullet scenarios, people soon found out that it just ain't that easy. Although these tools and models have their place and can be useful in the right environments, *none* of them has become the hoped-for universal panacea. Worse yet, the misapplication of these approaches has probably done far more damage than good.

Interestingly enough, the nursing profession had similar problems with regard to the use of tools and formal models. They had fallen into the same trap many architects and designers fall for: forgetting that the model is a tool, not a mirror.

> *The model is a tool, not a mirror.*

Rules cannot tell you the most relevant activities to perform in a given situation or the correct path to take. They are at best "training wheels"—helpful to get started but limiting and actively detrimental to performance later.

Dr. Deborah Gordon contributed a chapter to Benner's book, in which she outlines some of the dangers of overreliance on formal models for the nursing profession. I've reinterpreted her sentiments with the particulars of our profession, but even the original version will sound pretty familiar to you.

Confusing the model with reality

> A model is not reality, but it's easy to confuse the two. There's the old story of the young project manager, where his senior programmer announced she was pregnant and going to deliver during the project, and he protested that this "wasn't on the project plan."

Devaluing traits that cannot be formalized

Good problem-solving skills are critical to our jobs, but problem solving is a very hard thing to formalize. For instance, how long can you just sit and think about a problem? Ten minutes? A day? A week?

You can't put creativity and invention on a time clock, and you can't prescribe a particular technique or set of techniques. Even though you *want* these traits on your team, you may find that management will stop valuing them simply because they cannot be formalized.

Legislating behavior that contradicts individual autonomy

You don't want a bunch of monkeys banging on typewriters to churn out code. You want thinking, responsible developers. Overreliance on formal models will tend to reward herd behavior and devalue individual creativity.[23]

Alienating experienced practitioners in favor of novices

This is a particularly dangerous side effect. By targeting your methodology to novices, you will create a poor working environment for the experienced team members, and they'll simply leave your team and/or organization.

Spelling out too much detail

Spelling out the particulars in too much detail can be overwhelming. This leads to a problem called *infinite regress*: as soon as you make one set of assumptions explicit, you've exposed the next level of assumptions that you must now address. And so on, and so on.

Oversimplification of complex situations

Early proponents of the Rational Unified Process (and some recent ones) cling to the notion that all you have to do is "just follow the process." Some advocates of Extreme Programming insist all you need to do is "just follow these twelve—no wait, maybe thirteen—practices" and everything will work out. Neither camp is correct. Every project, every situation, is more complex than that. Any time someone starts a sentence with “All you need to do is...” or “Just do this...,” the odds are they are wrong.

Demand for excessive conformity

The same standard may not always apply equally in all situations. What worked great on your last project may be a disaster on this one. If Bob and Alice are hugely productive with Eclipse, it might wreck Carol and Ted. They prefer IntelliJ or TextMate or vi.[24]

Insensitivity to contextual nuances

Formal methods are geared to the typical, not the particular. But when does the “typical” really ever happen? Context is critical to expert performance, and formal methods tend to lose any nuances of context in their formulations (they have to; otherwise, it would take thousands of pages just to describe how to get coffee in the morning).

Confusion between following rules and exercising judgment

When is it OK to break the rules? All the time? Never? Somewhere in between? How do you know?

Mystification

Speech becomes so sloganized that it becomes trivial and eventually loses meaning entirely (for example, “We’re a customer-focused organization!”). Agile methods are fast losing effectiveness because of this very problem.

Formal methods have other advantages and uses but are not helpful in achieving these goals. Although it may be advantageous to establish baseline rules for the lower skill levels, even then rules are no substitute for judgment. As ability to judge increases, reliance on rules must be relaxed—along with any rigid institutional enforcement.

Avoid formal methods if you need creativity, intuition, or inventiveness.

Don’t succumb to the false authority of a tool or a model. There is no substitute for thinking.

Consider the Context, Again

One of the most important lessons from the Dreyfus model is the realization that although the novice needs context-free rules, the expert uses context-dependent intuition.

> The man with his pickled fish has set down one truth and has recorded in his experience many lies. The fish is not that color, that texture, that dead, nor does he smell that way.

➤ *John Steinbeck, The Sea of Cortez*

In *The Log from the Sea of Cortez* [Ste95], Steinbeck muses on the interplay of context and truth. You can describe a Mexican Sierra fish in the laboratory. All you have to do is "open an evil smelling jar, remove a stiff colorless fish from formalin solution, count the spines, and write the truth 'D. XVII-l5-IX.'" That's a scientific truth, but it's devoid of context. It's not the same as the living fish, "its colors pulsing and tail beating in the air." The living fish, in the context of its habitat, is a fundamentally different reality from the preserved fish in the jar in the lab. Context matters.

You may have noticed that the high-priced consultant's favorite answer is "It depends." They're right, of course. Their analysis depends on a great many things—all those critical details that the expert knows to look for, while ignoring the irrelevant details. Context matters.

You might ask the expert to open a locked door. Fair enough, but consider the difference context might make: opening the door to rescue the baby on the other side in a burning house is quite a different exercise than picking the lock and leaving no traces at the Watergate Hotel, for instance. Context matters.[25]

There is an inherent danger in *decontextualized objectivity*, that is, in trying to be objective about something after taking it out of its context. For instance, in the previous Steinbeck quote, a preserved fish—perhaps dissected for study—is quite a different thing from the silvery flashing beast gliding through a cresting wave.

Beware decontextualized objectivity.

For the breaking-and-entering example, "I want to open this locked door" really isn't sufficient. What's the context? Why does the door need to be opened? Is it appropriate to use an axe, a chainsaw, or lock-picking tools, or can we just go around back and use the other door?

In systems thinking, as in object-oriented programming, it's often the relationships between things that are interesting, not the things themselves. These relationships help form the context that makes all the difference.

Context matters, but the lower several stages on the Dreyfus model aren't skilled enough to know it. So once again, we have to look at ways of climbing the Dreyfus ladder.

Day-to-Day Dreyfus

Well, this has been all fun and fascinating, but what good is the Dreyfus model really? Armed with knowledge of it, what can you do with it? How can you use this to your advantage?

First, remember that one size does not fit all, either for yourself or for others. As you can see from the model, your needs will be different depending on what level you are on. What you need for your own learning and personal growth will change over time. And of course, how you listen and react to others on the team needs to take into account their own skill levels as well.

One size does not fit all.

Novices need quick successes and context-free rules. You can't expect them to be able to handle novel situations on their own. Given a problem space, they'll stop to consider everything, whether it's relevant or not. They don't see themselves as part of the system, so they won't be aware of the impact they're having—positive or negative. Give them the support they need, and don't confuse them unnecessarily with the big picture.

At the other end of the spectrum, experts need to have access to the big picture; don't cripple them with restrictive, bureaucratic rules that aim to replace judgment. You *want* the benefit of their expert judgment. Remember they think they're part of the system itself, for better or for worse, and may take things more personally than you would expect.

Ideally you want a mix of skills on a team: having an all-expert team is fraught with its own difficulties; you need some people to worry about the trees while everyone is pondering the forest.

Since the Dreyfus model is probably new to you from reading about it here, you're probably still a novice at understanding and using it. Understanding the

Dreyfus model and skills acquisition is a skill itself; learning to learn is subject to the Dreyfus model.

Recipe 7 Learn the skill of learning.

Going Forward

We will use these lessons of the Dreyfus model to guide the rest of the book. To embark on this path to expertise, we'll need to do the following:

- Cultivate more intuition
- Realize the increasing importance of context and of observing situational patterns
- Better harness our own experience

To see how to accomplish these goals, we'll start the next chapter by taking a closer look at how the brain works.

Next Actions

- Rate yourself. Where do you see yourself in the Dreyfus model for the primary skills you use at work? List the ways your current skill level impacts you.
- Identify other skills where you are a novice, advanced beginner, and so on. Be aware of the possibility of second-order incompetence when making these evaluations.
- For each of these skills, decide what you need to advance to the next level. Keep these examples in mind as you read the remainder of this book.

- Think back to problems you've experienced on a project team. Could any of them have been avoided if the team had been aware of the Dreyfus model? What can you do differently going forward?

- Think of your teammates: Where are they on their journey? How can that be helpful to you?

Footnotes

[4] Described in *Mind Over Machine: The Power of Human Intuition and Expertise in the Era of the Compute* [DD88].

[5] See *Tools of Critical Thinking: Metathoughts for Psychology* [Lev97].

[6] Cited in *From Novice to Expert: Excellence and Power in Clinical Nursing Practice* [Ben01].

[7] I forward it with my compliments and a large check to my accountant, who is expert in these matters. I hope.

[8] See *Teaching and Learning Generic Skills for the Workplace* [SMLR90].

[9] See *Personal Knowledge* [Pol58].

[10] See *Standards for Online Communication* [HS97].

[11] Cited in *The Scope, Limits, and Training Implications of Three Models of Aircraft Pilot Emergency Response Behavior* [DD79].

[12] Like thoroughbreds, not mustangs.

[13] In 1968, a difference of 10:1 in productivity among programmers was noted in *Exploratory experimental studies comparing online and offline programming performance* [SEG68]. The gulf seems to have widened since then.

[14] Identified in *From Novice to Expert: Excellence and Power in Clinical Nursing Practice* [Ben01]; more on this landmark book in just a bit.

[15] Described in *Standards for Online Communication* [HS97].

[16] That's *patterns* in the usual English sense, not software design patterns.

[17] See *The Complete Problem Solver* [Hay81] and *Developing Talent in Young People* [BS85].

[18] Described in *From Novice to Expert: Excellence and Power in Clinical Nursing Practice* [Ben01].

[19] Described in *Influence: Science and Practice* [Cia01].

[20] This was an older study; don't go calling the hospital up now with bogus orders, or the feds may well come a knockin'.

[21] *Fortune Magazine*, March 18, 2002, p.50.

[22] The drummer analogy is stretching it a bit, but I do talk more about the rhythm of development projects in *Practices of an Agile Developer* [SH06].

[23] Of course, there's a balance here—you do not want a "cowboy coder" who ignores the team and common sense to strike out on his own.

[24] OK, I have to confess that over the course of time, I wrote this book using vi, XEmacs in vi mode, and TextMate.

[25] For more on expertise in lock picking, see *How to Open Locks with Improvised Tools* [Con01].

The human brain starts working the moment you are born and never stops until you stand up to speak in public.

Sir George Jessel

Chapter 3

This Is Your Brain

Your brain is the most powerful computer in existence. But it's not at all like the computers we're familiar with, and in fact it has some really odd peculiarities that can either trip you up or propel you to greatness. So in this chapter, we're going to take a quick look at how your brain works.

We'll see where intuition comes from, begin to look at harnessing it better to become more expert, and learn why a lot of things that perhaps you think "don't matter" turn out to be absolutely critical to your success.

Since we're pretty familiar with computers, it seems useful to talk about the brain and its cognitive processes as if they were designed as a computer system.

But that's *just a metaphor*. The brain is not a mechanical device; it's not a computer. You aren't programmable. Unlike a computer, you can't even perform the same action exactly the same way twice.

That's not just a hardware problem; it has nothing to do with muscles. It's a software problem. The brain actually plans out your motion slightly differently each and every time, much to the chagrin of golfers, pitchers, and bowlers.[26]

The brain is a horrifically complicated squishy lump of stuff. It's so complicated that it has a very hard time analyzing and studying itself. So, please remember that this is *just an analogy*—but I hope a helpful one.

With that said: Your brain is configured as a dual-CPU, single-master bus design, as shown in Figure 5, *This is your brain*.

As we'll see in this chapter and the next, this dual design presents some problems, but it also presents some terrific opportunities that you might not be aware of.

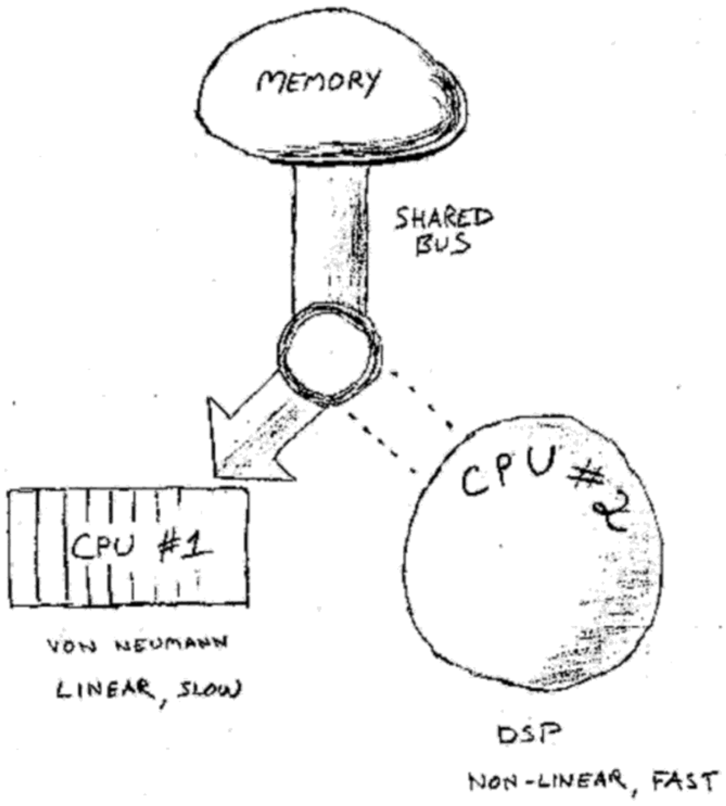


Figure 5. This is your brain

Your Dual-CPU Modes

CPU #1 is probably the most familiar to you: it is chiefly responsible for linear, logical thought, and language processing. It's like a traditional von Neumann--style CPU that processes instructions step-by-step, in order. CPU #1 is relatively slow and uses a relatively small amount of overall brain real estate.

It's programmed with an "idle loop" routine as well. If CPU #1 is not processing anything else, it will simply generate an internal stream of verbal chatter. It's that little voice in your head.[27]

CPU #2, however, is very different. Instead of the linear, step-by-step approach of CPU #1, CPU #2 is more like a magic digital signal processor. It's your brain's answer to Google: think of it like a super regular-expression search engine, responsible for searching and pattern matching. As such, it might grab matching patterns that aren't obviously related. It can go off searching while you are "thinking" of something else and return a result set asynchronously—and possibly days later. Since CPU #2 doesn't do any verbal processing, that means its results aren't verbal, either.

Notice that both CPUs share the bus to the memory core; only one CPU can access the memory banks at a time. That means if CPU #1 is hogging the bus, CPU #2 can't get at memory to perform searches. Similarly, if CPU #2 is cranking away on a high-priority search, CPU #1 cannot get at memory either. They interfere with each other.

Two CPUs provide R-mode and ***L****-mode.*

These two CPUs correspond to two different kinds of processing in your brain. We'll call the linear processing style of CPU #1 *linear mode*, or just **L**-mode. We'll refer to the asynchronous, holistic style of CPU #2 as *rich mode*, or *R*-mode for short.

Holographic Memory

Memory is stored holographically, in the sense that your memory has certain properties of a hologram.[28]

In a real hologram (made using a laser), every piece of the film contains the entire image. That is, if you cut the film in half, each half will still have the entire image—but with lower fidelity or resolution. You can continue to cut the film in half indefinitely, and smaller and smaller pieces will continue to contain a representation of the whole image. That's because the whole image is stored scattered across the whole film; each small part contains a representation of the whole.

Scientists have studied this phenomenon in mice. Researchers start by training a bunch of mice in a maze. Then they scoop out half of their brains with a melon baller (what better to do on a lonely Saturday night in the lab?).

The mice can still navigate the whole maze (although I imagine somewhat spastically), but with less and less precision as the researchers scoop out more and more.[29]

You need both: *R*-mode is critical for intuition, problem solving, and creativity. **L**-mode gives you the power to work through the details and make it happen. Each mode contributes to your mental engine, and for best performance, you need these two modes to work together. Let's start looking at the details of each of these vital cognitive modes.

Memory Must Be Refreshed

Remember the movie *Total Recall*? Well, if you can't, maybe your memories were suppressed by a secret spy agency as well. It turns out that this sort of mental manipulation isn't science fiction after all. Memories can be erased by simply repressing a specific enzyme.[30]

An enzyme located in the synapses called PKMzeta acts as a miniature memory engine that keeps memory up and running by changing some facets of the structure of synaptic contacts. If the PKMzeta process in an area of the brain stops for some reason, you lose that memory—no matter what it is.

It had long been thought that memory was somewhat like flash RAM; memory was somehow recorded by neuron configuration with a physical persistence. Instead, it is actively maintained by an executing loop.

Even with volatile static RAM, data sticks around as long as power is applied. It turns out your brain doesn't have static RAM, but instead it has dynamic RAM that needs constant refreshing or it fades. That means even riding a bicycle isn't something you can take for granted. It means you can unlearn anything. It means no matter how horrible or wonderful some experience is, you *can* lose it.

So, your brain is not like software. Software never ages and never degrades. But wetware must be refreshed, must be used, or it is lost.

If your brain stops running, it forgets everything.

Thanks to Shawn Harstock for this tidbit and write-up.

Memory and Bus Contention

R-mode is very important to your day-to-day work: it acts as your search and retrieval engine for long-term memory and ideas that are "in process." But as I mentioned, *R*-mode *doesn't* do any verbal processing. It can retrieve and recognize verbal elements, but it can't do anything with them by itself because of that memory bus contention between **L**-mode and *R*-mode.

For instance, have you ever had the experience of trying to describe a dream when you first wake up? Many times it seems that a crystal-clear, vivid dream evaporates from your memory as soon as you try to describe it in words. That's because the images, feelings, and overall experience are *R*-mode things: your dream was generated in *R*-mode. As you try to put your dream into words, you experience a sort of bus contention. **L**-mode takes over the bus, and now you can't get at those memories anymore. In effect, they aren't verbalizable.[31]

You have amazing perceptual powers, many of which *can't* be effectively put into words. For instance, you can instantly recognize the faces of a large number of familiar people. It doesn't matter whether they've changed their hairstyle, changed their manner of dress, or put on ten pounds or twenty years.

But try to describe the face of even your closest loved one. How do you put that recognition ability into words? Can you make a database describing the faces of the people you know in such a way that you could recognize them based on that description? No. It's a great ability, but it isn't rooted in the verbal, linguistic, **L**-mode.

And to compound problems, the *R*-mode search engine isn't under your direct conscious control. It's a bit like your peripheral vision. Peripheral vision is much more sensitive to light than your central vision. That's why if you see something faint out of the corner of your eye (such as a ship on the horizon or a star), it can disappear if you look at it head-on. *R*-mode is the "peripheral vision" of your mind.

R-mode isn't directly controllable.

Have you ever had the solution to a vexing problem (a bug, a design problem, the name of a long-forgotten band) come to you while you're in the shower? Or sometime the next day, when you aren't thinking about it? That's because *R*-mode is asynchronous. It's running as a background process, churning through old inputs, trying to dig up the information you need. And there's a lot for it to look through.

R-mode is quite diligent at storing input. In fact, it's possible that every experience you have, no matter how mundane, is stored. But it is not necessarily indexed. Your brain saves it (writes it to disk, if you will) but doesn't create a pointer to it or an index for it.[32]

Have you ever driven to work in the morning and realized with a start that you have no memory of actually driving the last ten minutes? Your brain recognizes that this isn't terribly useful data, so it doesn't bother to index it. That makes remembering it a little difficult.

However, when you're trying hard to solve a problem, *R*-mode processes will search *all* your memory for matches that might aid in the solution. Including all this unindexed material (and perhaps that lecture in school that you half-dozed through). That might really come in handy.

We'll see how to take advantage of that and look at particular techniques to help get around some of the other problems with *R*-mode in the next chapter. But first, let's take a look at a hugely valuable but very simple technique to deal with the fact that *R*-mode is asynchronous.

Who's in Charge Here?

You might think that the narrative voice in your head is in control and that the voice is your consciousness, or the real "you." It is not. In fact, by the time the words are formed in your head, the thought behind them is very old. Some considerable time later those words might actually be formed by your mouth.

Not only is there a time delay from the original thought to your awareness of it, but there is no central locus of thought in the brain. Thoughts rise up and compete in clouds, and the winner at any point in time is your *consciousness*. We'll look at this in more depth in *Defocus to Focus*.

Capture Insight 24x7

R-mode is unpredictable at best, and you need to be prepared for that. Answers and insights pop up independently of your conscious activities, and not always at a convenient time. You may well get that million-dollar idea when you are nowhere near your computer (in fact, you're probably much more likely to get that great idea precisely *because* you are away from the computer, but more on that later).

That means you need to be ready to capture any insight or idea twenty-four hours a day, seven days a week, no matter what else you might be involved in. You might want to try these techniques:

Pen and notepad

I carry around a Fisher Space Pen and small notepad. The pen is great; it's the kind that can write even upside down in a boiling toilet, should that need arise.[33] The notepad is a cheap 69-cent affair from the grocery store —skinny, not spiral bound, like an oversize book of matches. I can carry these with me almost everywhere.

Index cards

Some folks prefer having separate cards to make notes on. That way you can more easily toss out the dead ends and stick the very important ones on your desk blotter, corkboard, refrigerator, and so on.

PDA

You can use your Apple iPod or Touch or Palm OS or Pocket PC device along with note-taking software or a wiki (see *Manage Your Knowledge* for more on this idea).

Voice memos

You can use your cell phone, iPod/iPhone, or other device to record voice memos. This technique is especially handy if you have a long commute, where it might be awkward to try to take notes while driving.[34] Some voicemail services now offer voice-to-text (called *visual voicemail*), which can be emailed to you along with the audio file of your message. This means you can just call your voicemail hands-free from wherever you are, leave yourself a message, and then just copy and paste the text from your email into your to-do list, your source code, you blog, or whatever. Pretty slick.

Pocket Mod

The free Flash application available at http://www.pocketmod.com cleverly prints a small booklet using a regular, single-sided piece of paper. You can select ruled pages, tables, to-do lists, music staves, and all sorts of other templates (see Figure 6, *Disposable pocket organizer from pocketmod.com*). A sheet of paper and one of those stubby pencils from miniature golf, and you've got yourself a dirt cheap, disposable PDA.

Notebook

For larger thoughts and wanderings, I carry a Moleskine notebook (see the sidebar *Moleskine Notebooks*). There's something about the heavyweight, cream-colored, unlined pages that invites invention. Because it feels more permanent than the cheap disposable notepad, I noticed a tendency to *not* write in it until a thought had gelled for a while, so I wouldn't fill it up prematurely. That's bad, so I started making sure I always had a backup Moleskine at the ready. That made a big difference.

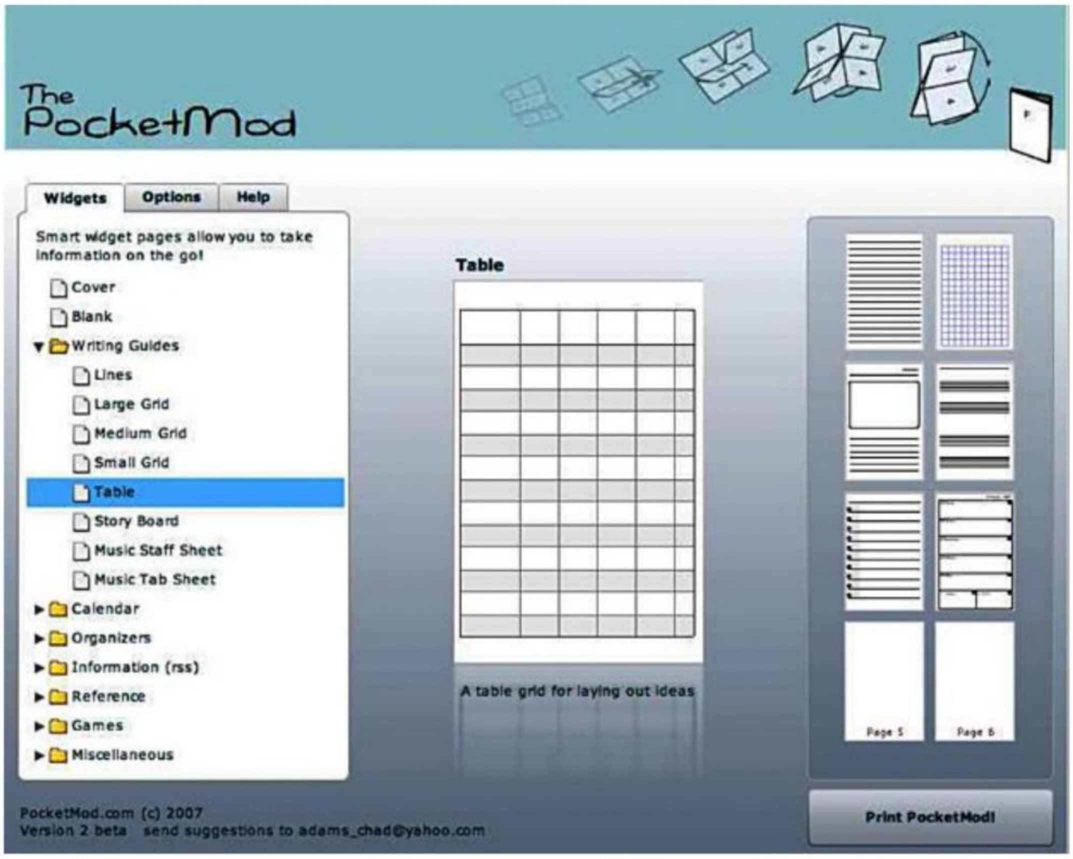


Figure 6. Disposable pocket organizer from pocketmod.com

Recipe 8 Capture all ideas to get more of them.

The important part is to use something that you *always* have with you. Whether it's paper, a cell phone, an MP3 player, or a PDA doesn't matter, as long as you always have it.

Moleskine Notebooks

A very popular style of notebook these days is made by Moleskine (see http://www.moleskine.com). These come in a variety of sizes and styles, ruled or not, thicker or thinner paper. There's a certain mystique to these notebooks, which have been favored by well-known artists and writers for more than 200 years, including van Gogh, Picasso, Hemingway, and even your humble author.

The makers of Moleskine call it "a reservoir of ideas and feelings, a battery that stores discoveries and perceptions, and whose energy can be tapped over time."

I like to think of it as my *exocortex*—cheap external mental storage for stuff that doesn't fit in my brain. Not bad for ten bucks.

If you don't keep track of great ideas, you will stop noticing you have them.

The corollary is also true—once you start keeping track of ideas, *you'll get more of them.* Your brain will stop supplying you with stuff if you aren't using it. But it will happily churn out more of what you want if you start using it.

Everyone—no matter their education, economic status, day job, or age—has good ideas. But of this large number of people with good ideas, far fewer bother to keep track of them. Of those, even fewer ever bother to act on those ideas. Fewer still then have the resources to make a good idea a success.[35] To make it into the top of the pyramid shown in Figure 7, *Everyone has good ideas; fewer go further*, you have to at least keep track of your good ideas.

Everyone has good ideas.

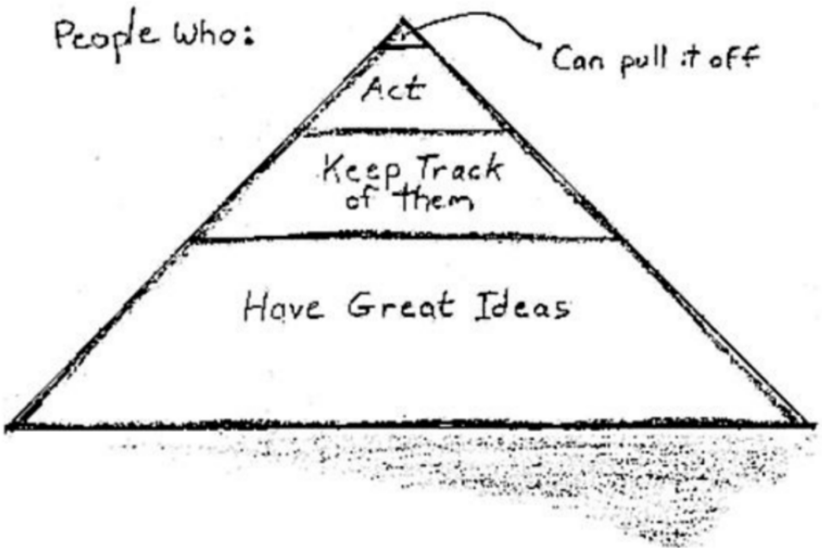


Figure 7. Everyone has good ideas; fewer go further

But that's not enough, of course. Just capturing ideas is only the first step; you then need to work with the idea, and there are some special ways we can go about doing that to be more effective. We'll talk about this in depth a bit later (see *Manage Your Knowledge*).

STOP Get something to take notes on, and keep it with you.