Herding Cats: A Primer for Programmers Who Lead Programmers

J. HANK RAINWATER

Apress™

Herding Cats: A Primer for Programmers Who Lead Programmers
Copyright © 2002 by J. Hank Rainwater

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-59059-017-1
Printed and bound in the United States of America 12345678910

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Editorial Directors: Dan Appleman, Peter Blackburn, Gary Cornell, Jason Gilmore, Karen Watterson
Technical Reviewer: Dave Christensen
Managing Editor: Grace Wong
Project Manager and Development Editor: Tracy Brown Collins
Copy Editor: Nicole LeClerc
Production Editor: Kari Brooks
Compositor: Diana Van Winkle, Van Winkle Design
Illustrators: Melanie Wells; Cara Brunk, Blue Mud Productions
Indexer: Carol Burbo
Cover Designer: Tom Debolski
Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, e-mail orders@springer-ny.com, or visit http://www.springer-ny.com.
Outside the United States, fax +49 6221 345229, e-mail orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 901 Grayson Street, Suite 204, Berkeley, CA 94710. Phone 510-549-5930, fax: 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at http://www.apress.com in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 1

Adapting to Your Leadership Role

In the beginning of any new job, we all have great hopes and, to some extent, a reasonable amount of fear that we might fail. As a successful programmer, you have, no doubt, had your share of new beginnings on projects and at places of employment. Now that you've been given the reigns to lead a group of programmers, a very new and perhaps daunting task is before you. You must evolve from programmer to leader as quickly as possible to thrive in your new software development role. This will entail adapting to a new social context and adopting new ways of interacting with your work world and the people in it.

Adaptation, a driving force of biological evolution, has been successful in helping our species climb out of the ooze and up the ladder of life to sentience. It took millions of years, but here we are, using language and dealing with abstract concepts such as computer programming. How did we get to this point? You'll have to ask your local biologist that question, but in the pages of this book, you'll rely heavily on your ability to adapt in order to face the challenges of leading programmers.

In this first chapter, you're going to become somewhat of an anthropologist. You'll look at the very human enterprise of writing code—specifically, you'll examine the types of individuals who engage in this wonderful activity. In learning more about the people you manage, you gain insights into how to successfully lead them. Many different ideas about how to lead programmers have currency today. Each generation of leaders starts from their own unique perspective and builds upon what they know and what they find works as a management and leadership style. I'm from the generation that grew up with slide rules and punch cards, and this will, no doubt, color my presentation. However, in years of working with programmers much younger than myself, I've learned that my generation doesn't own the only methods that work. I've had to adapt numerous times to the

changing needs of business, revolutions in technology, and growth and stubbornness in my own character as I faced the challenges of leadership. I'll share these experiences with you, and I believe we'll have a great journey together.

Do Real Leaders Wear Black?

Some do, and some even sport ponytails, depending on their hair situation, to fit in better with some of the younger geeks or (depending on your generational preferences) nerds. You may prefer neither of these terms and see yourself as a modern leader of business, guiding men and women like yourself who find programming a great intellectual thrill. Thus, the affectations I've alluded to, including the one referred to in the title of this section, shouldn't be taken too seriously. Do take seriously, however, your ability to personally relate to and identify with your programmers. Now that you're the leader, you can't fit in like you once did as part of the group, and you shouldn't really try too hard because you're the boss and you'll need to use this advantage from time to time in the course of conducting the business of constructing software. Someone once said, "Give me a lever large enough and I can move the earth." Being the boss can be such a lever.

You may not be convinced that image isn't important. It took me a long time to realize that what I looked like on the outside wasn't necessarily a reflection of my character. I still enjoy the trappings of "nerdom," but I also know that leading my team requires much more than just style. True, a certain image can go a long way toward reassuring your folks that you're one of them. But is it a crucial leadership skill? You may remember from the movie *The Net* how Angela (Sandra Bullock) was accepted by her online pals. They all said she was one of them and thus accepted her into their weird little circle. Of course, in the end it turned out that this wasn't really such a great thing. Learn from this: Image is truly only skin deep. What counts is character. That's why all the management techniques you may have learned and try to practice fail so often: They are techniques you have grafted onto your brain rather than nourished and grown from your heart.

How Important Is Being Cool?

So, continuing in this serious tone, should you wear black and embrace the affectations you believe contribute to coolness as a leader of programmers? Take the "Assess Your Level of Cool" test in the sidebar and see how you do. Note that some prefer the term "Ninjitsu"[1] rather than "cool," but I'm from the old school.

---

1. You know what a Ninja is—this word refers to the quality of being one, as in black-belt programming.

Remember, this is a self-help book, so you have to do some work. Pop quizzes aren't just given by stuffy old college professors—they show up in your daily work life all the time.

Assess Your Level of Cool

Select one or more answers to the following questions.

1. A "hacker" is a person who
    a. Makes furniture with an axe
    b. Programs enthusiastically as opposed to just theorizing about it
    c. Enjoys the intellectual challenge of creatively overcoming or circumventing limitations
    d. Maliciously tries to discover sensitive information
    e. Was a character played by Angelina Jolie before she was a tomb raider
2. A "cracker" is
    a. One who breaks security on a system
    b. A Southern white boy like your author
    c. Thinner than a cookie (see question 6)
    d. Considered to be a larval-stage hacker
3. "Phreaking" is
    a. The art and science of cracking the phone network
    b. An old nerd trying to be cool
4. "Ping" is
    a. Packet Internet groper
    b. The sound of a sonar pulse
    c. The other half of pong
    d. A quantum packet of happiness
5. "Worm" refers to
    a. A write-once read-many optical disk drive
    b. A virus program designed to corrupt data in memory or on disk
    c. A bilaterally symmetrical invertebrate
6. A "cookie" is
    a. A token of agreement between cooperating programs
    b. Something Amos made famous
    c. Something used to store and sometimes learn about the browsing habits of users

How do you think you did on the test? I once gave this assessment to a group of nonprogrammer types during a lecture on computer security to illustrate the kinds of people who get involved in hacking as well as protecting computers from threats. They didn't do very well on the test, but I bet you did way above average. All the answers are correct for each question.[2] Well, maybe choice b in question 3 is a bit of fiction, but this test does illustrate how programmers have been traditionally characterized as belonging to a particular subculture. Sometimes it's called, in the nonpejorative sense, the "hacker" culture (see the nice answers for question 1). Today these hacker stereotypes are disappearing. A programmer today more than likely holds an undergraduate degree in computer science and an MBA to boot. Nevertheless, each corporation has a culture, and your team has one as unique as the people who comprise the group. However the culture is defined, it is within this context that you lead and manage your people. Understanding the warp-and-woof (ways of interacting and thinking) of your programmers' culture can help you relate better to them and aid your leadership efforts. So wear a cool black T-shirt with an esoteric message emblazoned on the front if you desire, but there are more effective ways to relate to your people than just adopting or reinforcing a stereotypical image. This is the key theme of this chapter.

Hacker stereotypes are disappearing. A programmer today more than likely holds an undergraduate degree in computer science and an MBA to boot.

Be More Than Cool: Beware

Of course, if you're a dyed-in-the-wool hacker yourself, relating to programmers may be no problem. However, beware: Good programmers, while often promoted into management, don't often make the best managers or leaders of programmers. You have a great desire to work on the coolest projects when you should delegate. You often spend many hours on code review, when an hour would do, trying to get every little comment and indentation just right. There are times when you give up trying to help others understand what you want and you just do it yourself. Don't misunderstand me here, you must be concerned about the details of the code for which you are responsible, but the programmer-cum-manager is often guilty of not seeing the forest for the trees.

2. You can look up most of these terms in *The New Hacker's Dictionary*, Third Edition, by Eric S. Raymond (The MIT Press, 1998).

You must be concerned about the details of the code for which you are responsible but the programmer-cum-manager is often guilty of not seeing the forest for the trees.

At the other extreme, you may manage during the day and write code at night, depending on whether or not you have a life. Perhaps coding is your life and managing is your day job—this can work unless you lose your passion for the work. Maintaining your passion is essential in my understanding of what it takes to lead programmers. You will work through a number of management skills in this chapter and the chapters ahead, and these skills will help you balance your work life and keep your passion strong for your job. One key management skill that allows you to have time to lead is delegation. It is a cornerstone of leadership, and I focus heavily on it in Chapter 8. For now, realize that delegation involves trusting your staff. Trust takes time to build and is essential to successful leadership. Trust also is a reciprocal human activity. In this chapter, you'll learn to trust your instincts about people as you refine those instincts with a bit of anthropological insight into the mind and hearts of programmers.

Leading Weird, Eccentric, Strange, and Regular Folks into Great Work

Now, I don't want to take all the fun out of managing programmers, even though it has often been described as an exercise in "herding cats"—a reference, no doubt, to the independent nature of the creative individuals who choose to write code. The fun part is that these sometimes troublesome, always needed, and usually fascinating employees can be a blast to work with. Getting to know them better will improve your management style.

If you are a true lover of programming, you understand what it means to be close to your code—it may seem like second nature to you. As Ellen Ullman writes in *Close to the Machine*:

> *A project leader I know once said that managing programmers is like trying to herd cats . . . I mean you don't want obedient dogs. You want all that weird strangeness that makes a good programmer. On the other hand, you do have to get them somehow moving in the same direction.*[3]

3. Ellen Ullman, *Close to the Machine* (San Francisco: City Lights Books, 1997), p. 20.

This "same direction" is the goal of programmer management, but since each programmer is different, you have to lead in a unique way for each of your people. You can't lead programmers if you don't understand them. In the following section, I outline various programmer "types" and the traits that define them. You may recognize some of your employees in this list of types, which I'll call "breeds," as this is a book about cats.

Recognizing Programmer Breeds

What's a typical programmer like? Can you stereotype programmers just for the purpose of understanding them? Maybe. Like so many personality assessment tests from the field of psychology, it is helpful to look at programmer traits in isolation and recognize that many of these characteristics can coexist in the same person even if they seem contradictory. I've grouped the breeds into three categories: major, minor, and mongrel. *Major* refers to the most common types you'll find in the workforce. The *minor* breeds are sometimes seen, but not as frequently as the major ones. *Mongrels*, as you might expect, are not very desirable, but they do exist in the workplace, and as a result you need to recognize them. Mongrels can work out fine, as long as you help them build up their skills to overcome the weaknesses they inherently bring to the coding process.

As mentioned previously, any individual can be an amalgam of the characteristics identified with a breed—this makes working with that person a challenge but well worth the effort. Programmers are a wonderfully complex people. Relish the differences and unique styles of each breed. You'll probably recognize many of these traits the next time you look in the mirror.

The Major Breeds

The following are the major breeds and their characteristics.

The Architect.[4] This breed is highly prized by most managers and can be a valuable asset to your team. Architects are mostly concerned with the overall structure of the code. They dream in objects and printable whiteboards are their best friends. They live to solve business problems by abstraction and system analysis and then create concrete solutions in code. This is a necessary component of programming, but it isn't sufficient for the task. An architect may often have great ideas, but his or her

4. I'm using the term "architect" here in the sense of a programmer, not a full-fledged software architect. See Chapter 6 for a discussion about the importance of architecture in the grand scheme of development.

code may be so skeletal or obtuse that no one can pick it up and extend it. The rare architect can create a good system in his head, or preferably in Visio, and then flesh out the code and almost become a one-person show. The downside of this is that sometimes the architect's code may become a one-owner pet: It can't do tricks for anyone else.[5] Some architects are only interested in getting the code started and then handing it over to someone at a "lower" level for completion. You'll sometimes find strange constructs in an architect's code, such as message boxes in error traps when the code is supposed to run as a DLL on a server.

The Constructionist. This programmer just loves the process and result of writing code. Constructionists don't always have a master plan, but they are often fast and their code is usually fairly free of bugs even in the alpha stage. Constructionists' code originates from intuition and thus they appear to code by the seat of their pants. A constructionist also may have really great intuition and the master plan is in his or her head, so the code flows naturally from this source. Ask a constructionist for documentation and he or she will say the code is self-documenting. Tell a constructionist he or she must write documentation and you'll probably get some pretty good stuff. Of course, the code should be self-documenting, but in his or her heart, this programmer loves the act of creation and puts this first in his or her activities. The constructionist does so many daily builds that even Microsoft would be amazed. This can lead to solid code, but sometimes, as the scope creeps (and it always does), the solidity of the code can fracture and the constructionist will find him- or herself hacking away at solutions to preserve a personal sense of completeness and having done the job well. Team a constructionist up with an architect and you'll have a solid team. Find a constructionist in the same body as an architect and you'll solve most of your people problems.

The Artist. Writing code *is* as much an art as a science—that's why universities often put both departments in the same college and call it the College of Arts and Sciences. Take away the artistic side of programming and you'll lose many who find a great deal of job satisfaction in the craft of coding. The artist is in love with the act of creation: taking business requirements, mapping them to programming constructs, and elegantly making user interface objects present themselves with grace. Some artists, when they work on components that have no visible interface, will create beautiful symmetries of logic. The downside of being an artist is that it often leads to extended coding time as the programmer tries to see

5. This concept is important because one authority estimates that at least 70 percent of software cost is related to maintenance. See William H. Brown et al, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis* (New York: John Wiley & Sons, 1998), p. 121.

how many equal signs he or she can put in one line of code and still get the correct Boolean result. The upside is that code that doesn't reflect artistry is often lacking any real design and craftsmanship on the part of the programmer. Take out these artful qualities and you have a time bomb waiting to go off under the fingers of your users. The artist shares qualities with the constructionist and architect but has a flare for style.

The Engineer. You have to love these boys and girls. They will buy every third-party tool available, write dozens of COM objects, and hook them all together such that they actually work in version 1. It is only when version 1.1 needs to be built that their love of complexity rears its ugly head. Programming is often described as software engineering, and many aspects of our profession can be constrained and guided by this approach. Just don't let the engineer run the whole show for you. Engineered software isn't a bad thing because in the best sense of the term, engineering is the application of scientific principles to software problems. You need programmers who aren't afraid of complexity, but you don't want those who are just in love with creating it needlessly. I don't mean to give a bad rap to engineers—I was one for many years on the hardware side of computers. Nevertheless, it is this hardware dimension that sometimes runs counter to the aspects of software that make it soft (i.e., flexible and easy to reuse). Hardware usually serves one distinct engineered purpose, and you don't always want your software to be like this.

The Scientist. These are men and women after the hearts of Babbage and Turing. They would never write a GoTo in their life. Everything would be according to the fundamentals of computer science, whenever the day that programming is more science than art comes. And this is often the problem: They are overly concerned with purity, while you are concerned with tomorrow's code-complete date and producing good enough software. Scientists are useful and their ideas are often essential in solving particularly tough coding problems. Just watch out that purity doesn't overshadow practicality. Engineers and scientists have some similarities in that they both value the complex—sometimes you might think they worship at the shrine of whatever god represents complexity. (They often do bring their offerings to the temple!) Value the insights of the scientist and use their creations when appropriate, but beware of the legacy of complexity that will be perpetuated if they have free reign in the code.

The Speed Demon. As the name implies, these men and women are fast. No comments, no indentations, and bad variable naming conventions, but they do produce and it often works pretty well until the first untrapped error occurs. Sometimes these coders are just young in the profession and want to impress you because they think speed is the primary behavior you expect as a manager. Haven't we often given that

impression? Perhaps we managers are to blame for speed demons. Our bosses hand down the milestones they gathered from some meeting of the great minds and our job is to make it so. Haven't we often heard how foolish it is to establish a coding deadline before the requirements are gathered? *Get over it.* The real world is like this, and users and the marketplace often demand that we promise before we plan. That's why you are reading this book—you want some help in the fast, cruel, and often unforgiving world of software development.

The Minor Breeds

The following are the minor breeds and their characteristics.

The Magician.[6] You don't know how this programmer does it, but he or she always seems to solve the apparently intractable problems with unique solutions that no one thought of before. The magician also does it on time and sometimes it is understandable and maintainable software. A little magic can go a long way in our craft; too much and you may find yourself a sorcerer's apprentice rather than a clearheaded manager of hardworking people. In other words, if you depend too much on the magician, he or she will eventually let you down: No one can perform magic every time.

The Minimalist. This programmer produces sparse code, though it is often very powerful. Every procedure fits on a single screen in the code editor. Objects are nice and tidy and have a single-minded purpose. Sounds good, doesn't it? It can be, as long as the minimalist isn't just trying to get through the job so he or she can move on to the next, more exciting project. Sometimes—and this is a trait sometimes shared by architects—minimalists are easily bored once the problem is solved and they don't want to get down and dirty with the code as problems show up in alpha testing. Some minimalists are rather picky about the applications they want to work on. They are often very bad at code maintenance.

The Analogist. Okay, I may have made up this word and no, it isn't the nurse that puts you to sleep before a surgery—this is the programmer who really isn't very good at abstraction but is excellent at analogy. Analogists drive you mad during the design meeting as you constantly tire of their analogies, but often they do grasp the problem at hand and

6. Some may prefer the term "guru" or "wizard." I like "magic."

can often produce practical, maintainable code. Sometimes they have favorite analogies that they try to apply to every software issue. They like to think of components as moving parts and when things are working well they will say their code is "firing on all cylinders." Their analogies are always tied to some tangible object rather than an abstract one. You get the idea. Mix them with an architect and, if they don't kill each other first, you'll have some righteous software. The only danger with an analogist is that he or she may not do sufficient abstraction to create objects that have a clean interface for hooking up to other layers in the software. Being able to create a sufficiently abstract object interface is one of the great strengths of object-oriented (OO) programming, and sometimes the person who must always think in concrete ways will be unable to get the job done adequately.

The Toy Maker. This programmer overemphasizes the joys of technology. You have a person who loves new toys but you get the same old woes. In all honesty, all of us in this great craft love the toy aspect of technology. I remember my first computer: It was analog, you turned dials that closed switches in a predetermined hardware algorithm. It was sort of a slide rule on steroids and I loved it. I still love the joy that comes from working with neat technological tools. With toy makers, temper their love of toys with the purpose of their employment: to produce business solutions. Just because they managed to fit 30 user interface controls on a screen that is supposed to work at 800×600 doesn't mean they have met users' needs.[7] Toy makers, while showing a good grasp of the technology, fail to consider the end purpose of the software. They often think that their job is to have fun with the tools, rather than consider the aspects of programming that make maintenance possible without massive efforts.

The Mongrels

The following are the mongrels and their characteristics.

The Slob. There isn't much good to say here. Some folks are just sloppy and it shows in their code. They ignore small things such as properly spelled variables in correct Hungarian notation. Perhaps personal problems prevent them from doing good work. Perhaps they need some guidance in how to write effective code. They start out with one style and after one or two procedures they have adopted another. Following their code is painful and sometimes you have to rewrite it late at night just to be sure

7. Don't you hate users? What fun we could have if we only wrote software for programmers.

you meet the deadline. You, their manager, failed—they didn't, they are just slobs who probably should be transferred to beta testing. No, strike that, this would only move the problem down the path a bit and it could come back to bite you. Some slobs can be rehabilitated if they really love writing code and should be given more personal attention and mentoring. Those that can't may just need a metaphorical kick in the seat of the pants or an introduction to a job placement counselor.

The Intimidated. This programmer doesn't know where to start. He or she is constantly looking at (or waiting for) the specification, trying to find a point to begin. Not to worry, being timid can be a good thing when it leads to careful code, even if it is the result of the poor programmer not wanting to create runtime errors. Your job is to give the intimidated some prototype code that illustrates where to begin and a style to emulate. Often, those with only a few years in the profession exhibit some timidity and by nurturing them you can change their nature. You may also find timidity in an experienced programmer who hasn't had such a great track record. Maybe his last performance review was bad and he wants to do better but is afraid of screwing up. Lack of confidence often shows up as timidity, so bear with him and enable him achieve a little success as you hold his hand from time to time. I cover mentoring, which is the best way to nurture a timid programmer, in Chapter 8. You'll discover that being a mentor is one of your primary roles as a leader and it will pay you back handsomely for the efforts you put into it.

The Amateur. Amateurs are programmer wannabes. They come into the ranks of hackers as power users of some macro-writing tool. They left their cozy role in support or testing because they think programmers are way cool. Of course, we *are* cool but this is just a by-product of what we do. These folks need education and you must carefully assess their progress up the learning curve before you let them handle mission-critical application creation. These wannabes often become disillusioned with the job once they learn how hard programming can be and how much attention to detail is required. They often fail to see that object-oriented methods are superior to the procedural paradigm because they just haven't had the right epiphany. In defense of amateurs, remember the following saying: "Amateurs built the ark, professionals built the Titanic." Sometimes the fresh viewpoint of an amateur can be helpful to us old, sour techno-grouches.

The Ignoramus. This programmer is also known as Mr./Ms. Stupid, or worse, he or she is dumb and doesn't know it. Watch out for these people. You may have inherited them—please don't hire them. Now, I'm not being prejudiced toward the mentally challenged, but they just don't have a useful role in a profession that requires constant learning and

self-discipline. Ignorance can be tolerated as long as it isn't willful. Maybe move this person to the testing department, because sometimes users who are stupid find the bugs.[8] Another word about stupidity: We all suffer from the constant problem that occurs between the keyboard and the chair. If writing code didn't require some smarts, everyone would do it, right? Just be sure you don't mistake ignorance for stupidity. Ignorance can be cured, stupidity should be shunned. If you came on board a department that wasn't assembled by a professional programmer, you may find some of these breeds in your midst. Some nontechnical business leaders may have put the group together from folks who had sold themselves as programmers but don't have the gift.

The Salad Chef. A love of cooking up software reigns here. This breed consists of a bit of the engineer, the slob, and a not-so-gifted artist combined, but the ingredients are out of proportion. The result is a smorgasbord of coding styles and add-on components, and a general disorderliness of code. It might look appealing, but one bite and you know you're going to die. Send this programmer to a cooking class and be sure you don't have a full-blown slob lurking underneath what appears on the surface to be talent. This mongrel breed might appear seldom as a pure form, but I mention it because the trait shows up in a number of programmers' coding styles. If they can't conform to your corporate standards, you'll have a full-time job on your hands just trying to figure out what they have done and how to maintain their code. Your role as a code reviewer (see Chapter 6) will be crucial to rehabilitating the salad chef.

Working with the Breeds

Programmers are people first, so all the traits in the previous section may exist to a greater or lesser degree in the same person. Some of these traits are in opposition to each other, but don't worry about this. Every human walking the planet tonight is a bit of a contradiction, and you and your programmers are no different. What is important for you to do as the manager of these wonders of nature is respond with understanding, appropriate motivation and, above all, wisdom that's best gained by experience on the job learning to know your folks. Try to identify your programmers by the facets of their characters that shine the brightest under the sunshine of new endeavors and the lightening flashes of those projects near their deadlines.

---

8. I prefer the term "program anomaly" or "undocumented feature offering (UFO)" over "bug."

Try to identify your programmers by the facets of their characters that shine the brightest under the sunshine of new endeavors and the lightening flashes of those projects near their deadlines.

What breeds would make up a good team of programmers, assuming you could create your department from scratch? My first choice would be to have a good balance between architects and constructionists. These two breeds bring the best of needed skills to software creation: One has a high-level view, and the other is good at the details. An artist might be your second choice to mix in from time to time. Alas, you probably will not be able to assemble your group from the ideal candidates. You'll need to work with what you have, and thus your ability to deal with the blended nature of the traits I've stereotyped must be shaped by your insight, patience, and mentoring—leadership requires all three working in concert.

Another personality type you should be on the lookout for is the cowboy programmer. This type doesn't fit well into the other breeds listed previously because it is best described as an overall attitude. This attitude describes the programmer who may be really good at what he or she does but practically impossible to manage. Cowboys have the idea that they can pick and choose the programming work they want to do, and they can do it on their terms, according to their schedule, and by whatever means they see fit to employ. You could call this type a lone wolf (or a stray cat, in keeping with the context of this book). These cowboys can work miracles or wreak havoc, depending on your needs and ability to tolerate personal eccentricity. Be careful with cowboys—they will never be part of your team. Use them only as a last resort or if the project is truly a stovepipe or silo job that you don't expect your team to maintain.

Why do we find all these interesting personality characteristics in programmers? I believe it is because the nature of the software developer's work attracts a certain breed of human. In the classic book, *The Mythical Man-Month*, Frederick Brooks[9] describes our craft as one that provides five kinds of joys:

1. The joy of making things

2. The joy of making things that are useful to other people

3. The fascination of fashioning puzzle-like objects of interlocking moving parts

---

9. Frederick P. Brooks, *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition* (New York: Addison-Wesley, 1995), p. 230. This is a timeless classic—very few books in our field are reissued after 25 years, and this one is truly worthy.

4. The joy of always learning, of a nonrepeating task

5. The delight of working in a medium so tractable—pure thought-stuff—which nevertheless exists, moves, and works in a way that word-objects do not

These joys attract the kind of people you manage, and understanding what motivates them (and you) can be a tremendous aid to your leadership role.

Cat Fight! A Hissing and Scratching Contest

John and Kevin were constantly at odds during the design meeting. We had begun to discuss how the user would log into the system, and they were arguing about low-level details of construction techniques. The meeting was not progressing and there were many more features to be designed, or at least everyone thought so in spite of the lack of a clear agenda for the meeting. John and Kevin always fought because John, who was a consultant, and Kevin, who was a longtime employee and a very creative programmer, had very different motives and plans for the meeting. They were more interested in proving who was the smartest than designing a system, even though John had been designated as the development leader, a job Kevin desperately wanted. It didn't help that the boss wasn't in the meeting; there was no one to act as a negotiator.

A day went by, and the other programmers grew more silent as John and Kevin fought over each piece of the design. By the end of the second day, very few whiteboard printouts had been produced and what had been done came at the cost of long hours of tedious battling between the two scrapping cats, John and Kevin. The rest of the team became so discouraged that they began to doubt whether the system would ever be built. To compound matters, the team had been tasked to get this new system created as soon as possible because the legacy system was hurting the company in the marketplace.

This short story illustrates some of the difficulties with consultants and employees working on the same team, especially when the boss isn't in the room. You might question the choice of a consultant as the development leader, too. You also can see that a design meeting without a careful plan, an agenda, and a method for resolving differences can waste time and place the whole idea of "design before coding" in jeopardy. The dynamics of interpersonal relationships on the development team also illustrate the need to know your people before you create team leadership roles.

The end result of this particular story was that the project was canceled and a competing team in another division designed and built the software. The absent manager also had some dreary and difficult days explaining to his boss why they didn't step up to the plate and hit a home run after many promises and reassurances were given prior to project commencement.

Glory, Honor, and Greenbacks

Everyone with a job wants to be appreciated and to feel his or her contributions are meaningful. Even incompetent workers want to feel appreciated, in spite of the fact that they make negative contributions to the company. Sure, we all say coding is its own reward, but take away our paycheck and see how long we pound the keys. Pay us well and we will still crave recognition among our peers and an occasional pat on the back from the boss. Real cats may preen alone in a corner thinking no one is watching, but programmer cats really do like to preen in public. The practice of creating Easter eggs, though somewhat out of style at present, was always a sure sign that programmers wanted and needed an audience. You, their leader, are sitting in the first row and they are looking for your applause. Of course, some don't deserve applause but are truly in need of a serious word of prayer.[10]

There is a time to praise and a time to pause and consider if you're getting your money's worth out of your people. This is your challenge as a leader and manager. If you praise them, make sure it is in public. If you criticize, do it in private. These guidelines are not just given because they conform to rules of polite society—they are necessary because of the affect your actions toward one team member have on the whole group. Public humiliation never makes a group function as a productive unit. Public praise, when done genuinely and because the recipient deserves it, can work wonders. Don't be flippant when you praise, shouting out "You guys did a great job!" as your team walks out of a meeting. Take the time to make it stick. Give thought to the reason for the praise and let your team know your thinking.

There is a time to praise and a time to pause and consider if you're getting your money's worth out of your people.

Another word about praise. You may feel left out yourself, depending on how your boss treats you, when it comes time for compliments. As the leader, you are striving to make the group successful. When the group is successful, you praise it. Who praises you? Sometimes, the answer is "no one," and you may wish for a pat on the back from time to time yourself. This position, where your efforts and successes are the reason for the good reputation of the team, can be awkward if you seek fame for yourself. Leaders must learn to measure their confidence by how well those they lead perform.

---

10. This is a Southern expression for a trip to the woodshed, usually involving a spanking.

If you've risen among the ranks to become the leader, your job may be doubly difficult since you may be in charge of deciding the professional fate of your friends. Don't let friendship get in the way of business, but rather use it to motivate for the benefit of all. No one will feel you're manipulating friendship if in the end everyone enjoys the pleasure of success.

Motivating with Money

Well, I mentioned money previously, didn't I? Dang. I might as well get this over with since, as the Good Book says, ". . . money is the answer to everything."[11] A recent salary survey concerning programmers shows hourly rates range from $30 to over $150 per hour, with most of us in the middle of this wide spectrum. What determines if your folks are worth their rate? Performance, experience, effectiveness, timeliness, the current local market rate, and economic conditions are all factors, as well as your company's tradition of pay for high-tech workers. Your challenge when hiring new people or giving raises is to be fair and prudent at the same time.

Fair and prudent. Hmm, this is a difficult task because you may want to dole out the money as if it grew on trees thinking this helps job performance. Think again—a luxury today is a necessity tomorrow. Money is like power: It can corrupt, and don't make me quote another famous line from the newer portion of the Bible.[12] Getting back on track, how do you achieve a balance in monetary compensation matters? If you consider the task as one of balance, envision the scales of justice: On one side, you have a tray for fairness, on the other side, you have a tray for prudence. Fairness accepts weights that are equivalent to the programmer's experience and performance. Prudence accepts typical business smarts, such as watching the bottom line and the average salary of the programming staff. Keep these in mind as you make decisions about money—it's a good theory.

Theory? What about application? This is why money can be such a tough area to administer properly in your job. You have principles in mind you believe should guide your efforts to reward your staff, but at the same time, the economics of the current business climate and corporate policies you must live by may frustrate your planning. Salary can be supplemented with bonuses based on merit and/or corporate profits in some organizations. These incentives can work as long as the staff member's contributions can truly carry enough weight to affect the formula used to determine the bonus. You can try quarterly bonuses if you want, but I've

11. Actually, it accuses the fool of saying this. See Ecclesiastes 9:14–19 in a modern version for the context. Try not to get too depressed when you read this.

12. See the New Testament, 1 Timothy 6:10, where love, money, and evil are related together in a nice, logical syllogism.

found them problematic because once you start giving them, they begin to be expected. You should consult with your boss to work out a plan that fits within your organization. If you are the ultimate decision maker, do what you think is appropriate and keep in mind the issues of fairness and prudence.

The Thunking[13] Layer

Are you feeling all warm and fuzzy now? Probably not. You're most likely waiting for the bulleted list of things to avoid and things to embrace. There will be plenty of lists to look at in the chapters ahead, but at the moment I want to emphasize the role of thinking in your new role, rather than give you a long list of dos and don'ts. Thinking is perhaps the hardest thing we have to do as mangers and leaders, but it is crucial—absolutely crucial—to our success. As Jim McCarthy writes in *Dynamics of Software Development*:

> *The real task of software development management is to marshal as much intellect as possible and invest it in the activities that support the creation of the product.*[14]

Think while you're in the shower. Think while you ride your bike, take a walk, or go in-line skating. Think while you're listening to the dilemmas posed by design decisions. Think instead of watching TV or surfing the Web—they both may have 500 channels, but nothing is on, even if they relieve you of thinking. Think yourself full, work until you're empty, and then do it all over again. The result will surprise you.

Okay, let's do a little thinking about how to handle some typical situations.

Say you have a cat who is primarily a minimalist, but a very sharp one. You need him to enhance a product that he didn't write but that is critical for the current business goals. How do you motivate him when he takes one look at the other guy's code and says, "This is too complicated. It needs to be rewritten." He says this, of course, staring at code that took 2 years to write and is making the company money. The "other guy" is no longer around to explain the code, either. You have two choices: Give in, make him happy, and ruin any chance of meeting the deadline, or help him see how he can learn the existing architecture and make a significant contribution. Appeal to his sense of wanting a tidy architecture by asking him to document the existing code with the view that in the future, as time

---

13. You know what "thunking" is if you've been under the covers with compilers. You'll soon figure out my play on words.

14. Jim McCarthy, *Dynamics of Software Development* (Redmond, WA: Microsoft Press, 1995), p. 5.

permits, he can rewrite some of the objects to make them easier to follow. If he is a sharp programmer he ought to be able to figure out what another one has accomplished. Never hesitate to let competition serve a useful goal. For the minimalist, anything he didn't write is junk, but the truth may be that he is afraid he can't understand the code and doesn't want to admit it. Look for the hidden motivations that all of us share as humans and realize that programmers often hide behind intellectual excuses rather than admit their brains are not up to the task at hand.

How do you deal with an architect who thinks her object designs are far superior to anything that has been invented before and yet you think you see some weaknesses? Don't tell her that her design is flawed from the get-go or she will be on you like white on rice.[15] Ask her to explain how all the moving parts will work and to build some prototypes or test programs to illustrate the functions. If her prototypes don't show any problems, maybe you were wrong to see design flaws. Ask her to "componentize" her architecture if it seems like one massive and monolithic edifice. If the components can work together, maybe she has some good ideas. If the objects are too coupled and intertwined, she is too in love with complexity, which can lead to expensive software maintenance costs. A really great architect can create a framework that anyone who applies him- or herself can follow and extend. It also doesn't break when the next set of enhancements need to be added to the code. The key to working with an architect is to try to see the code through her eyes, not yours, even if she is blind in one and can't see out of the other.

Listen first and seek to understand before you use your authority as the manager to railroad a solution.

What is the common technique I'm suggesting for dealing with these people-induced situations? It is to listen first and seek to understand before you use your authority as the manager to railroad a solution. Programmers are no different from the rest of the human race when it comes to confrontations: They want to have a fair hearing of their point of view. As Stephen Covey writes in *The 7 Habits of Highly Effective People*, "Seek first to understand . . . then to be understood." Building consensus in technical decisions is an art whose canvas is openness to the ideas of others. It takes patience to construct such a canvas, even though you often feel you don't have time to build software and have everyone agree about

15. The phrase "like white on rice" is Southern for "in your face."

the methods. This may be your *feeling* in many cases, but consensus should still be your goal.[16] I have more to say about building consensus in Chapter 5, which deals with leading and managing a design meeting. You may be surprised to learn that consensus is not built through compromise.

Another example will illustrate the concept of understanding before judging. Some languages, such as Visual Basic (VB), don't allow true object constructors. I've seen an artist use the VB class initialize event to do a reasonable amount of work for getting the object set to be used by the consuming (parent) class.[17] In VB, if an object can't be instantiated, the error is difficult to trap—you will simply get a failure to create the object. When I asked him why he chose to use this event to handle error-prone activity, he replied that it was elegant, clean, and didn't require any action on the part of the calling object to be ready to use the interface. My opinion, of course, is that safe error handling should precede any attempt at beauty. I didn't tell him this first. I listened to his reasons, described how things could go wrong with his object, and then demonstrated an example in code. He learned something from the code example lesson that he would not have learned as well if I had just said, "Change it, this is not right." Again, when you give someone an opportunity to explain his or her point of view, that person will open up to your perspective.

How Are You Adapting?

You've taken in a lot of ideas in this chapter. You may feel overwhelmed by the scope of adaptation required to become an effective leader. Not to worry. We humans are still 99 percent genetically identical to monkeys, and the 1 percent that makes us different didn't appear overnight. The chapters ahead will help you adapt, overcome, and succeed as you look at other aspects of leadership and management.

Let's review what's been covered so far to help cement the key principles in your heart and mind.

You must adapt. Learning new management skills so you can lead requires adapting yourself to a new social context at work. You're the boss and this changes your relationship to those on your team.

16. Many would say that if everyone agrees then when things go bad, everyone is to blame. This may be true, but as managers we should be more concerned with fixing problems than affixing blame. Success has many parents—be one.

17 The initialize event in VB accepts no parameters and does not return any.

Character development is more important than image refinement. Leadership comes from within and is not manifested by an image. You're still a programmer, but your role as a leader of programmers requires you to work on deep issues involving insight into others' behavior as well as your own.

Know your staff. Become an anthropologist of programmer culture and people. Learn why your staff writes code the way they do and how you can work with the best qualities and rehabilitate the areas that do not lead to productivity. Herding cats means getting them to move in the same direction: This is what a leader strives to do.

Reward your staff appropriately. Act on the need to praise with words and compensate with money. Consider all the factors that constrain you financially, but be fair and prudent. Lavish praise in public when appropriate and reprove in private.

Think. Learn to apply what you know about people to build consensus. Listen to and understand other points of view before you make judgments. Cultivate your life of the mind: Make learning second nature in your new role as leader. Off-the-shelf plans to solve people problems are no substitute for you crafting plans and methods to deal with the problems and opportunities unique to your organization.

What Lies Ahead

In the next chapter, you'll turn an introspective eye on you, the manager, just as you did on your staff in this chapter. I'll ask some hard questions about how you approach your job as a leader of the critically important endeavor of building software in today's business climate. You must manage your staff to be a good leader, but managing yourself comes first.