Extracted from:

From Java to Ruby

Things Every Manager Should Know

We must all suffer one of two things: the pain of discipline or the pain of regret or disappointment.

► Jim Rohn

Chapter 2

Pain

If you want to truly know the health of your project, you have to get in touch with your pain. Athletes need to strike a balance between nagging twinges and deeper telling aches; development managers must distinguish between mere hiccups and symptoms of damaging disease. If you're succeeding with Java with only nagging pain—if you're delivering software your customers want on time within your budget and with happy developers—you probably shouldn't consider moving to Ruby. But if your aches run deep and are the symptoms of real disease, you have to act. The first step of introducing any new technology must be recognizing pain.

2.1 The House of Pain

After hearing all the hype around Ruby on Rails and other frameworks, you might be tempted to bolt for the exit too soon, but take a deep breath first. Don't let anyone tell you that Ruby is the answer to every question. Java does have some tremendous advantages over most other programming languages:

- *Java's population of programmers is huge.* With Java's massive pool of programmers, you can always find developers to hire or supplement your staff with temps, consultants, or even offshore development.
- *Java's open source community thrives.* Open source projects exist across a wide spectrum of problem spaces and fill many different niches. With Java, you can often get software for free that you'd have to build yourself or pay for on other languages.
- *Java is mature.* Java is often the safest choice.

- *Java is scalable.* We've learned enough from experience to build applications that scale.
- *Java offers choice.* You don't have to paint yourself into a corner with Java, because you have so many open standards defining many important interfaces and vendors to choose from.

Technology

In general, Java is a safe choice. It's mature, complete, and ready for outsourcing. For good reasons, Java has dominated Internet integration projects. Sure, Java can handle the most difficult enterprise integration issues. It has got features to solve notoriously hard problems:

- *Two-phase commit.* When the same application needs to coordinate two resources—such as two databases, for example—you sometimes need sophisticated software to tie the two together to keep things consistent. That software often uses two-phase commit, and Java supports it.
- *Powerful object-relational mapping.* Say your company's new DBA, a PhD student with ten years of schooling but no practical experience, proudly brings you a database model that is in 14th normal form. After they stop screaming, your programmers tell you they have never heard of 14th normal form, but they are quite sure that they don't want to subject their object model to such torture.

  Instead, your best programmers use a framework to translate data between the database schema and the objects of your application. That technique is known as object-relational mapping. Java has mature frameworks that do it well; Ruby doesn't.
- *Distributed objects.* When you need to build applications that span many different computers across the room, or even across the ocean, you sometimes need specialized software to help different pieces of the application communicate. Java can manage distributed objects in many ways. Ruby's options are more limited.

Ruby does have some simple transaction management and some rudimentary object-relational mapping, but those frameworks are nowhere near as powerful or as proven as their Java counterparts. If you were to attack any of these problems with Ruby today, you'd possibly wind up writing too much infrastructure and glue code.

With Java, a whole lot of your glue code comes for free. Treat these enterprise problems as if they were elephants. You can't bring down an elephant with a toothpick or a Swiss army knife. You need an elephant gun. The Java platform is an elephant gun.

The Hierarchy of Pain

I once talked to a customer about the problems in her enterprise. After questioning her developers and reading code, I strongly suspected that productivity was her biggest problem. In fact, I was wrong. They were horribly unproductive, but given the business climate, it didn't matter. Their group was dependent on requirements from three different business units, and the development team frequently had to wait weeks at a time for new requirements. It dawned on me that the director was telling me the truth. *Java development was simply not a bottleneck.* If Java is not the problem, don't go looking for a solution.

To be successful, you need to understand the pain in your organization and interpret it. You need to know where the pain is the most acute. Most projects don't fail for technical reasons. If you can't solve your communication problems, if you can't control scope creep, or if you can't tell what the customer actually wants, the choice of programming language is not going to matter to you. It's simply not high enough in the hierarchy of pain. Put this book down, and pick up another one. Ruby won't help; it will only introduce more risk.

But if you're inclined to believe that a simpler, more productive language would help, read on. Too many people worship Java, and too many vendors tell you you that Java can be all things to all people, and therein lies another kind of risk. Using the wrong tool for the job, even when it's the most popular tool, costs you money. Many of the problems that we solve with Java simply aren't elephants. I'd argue the problem we solve *most often* with Java—putting a web-based user interface on a relational database—isn't an elephant. It's a fuzzy little bunny rabbit. Or a squirrel. Although you can probably kill a rabbit with an elephant gun, bad things usually happen when you do.

Solve the wrong problem with the wrong technology, and the real pain begins. When Java was invented, it was simple, nimble, and robust when compared to most alternatives. But the pain crept up on us, slowly building from an itch to a piercing, throbbing crescendo. Let's look at the types of problems you're likely to find with the Java platform.

2.2 Poor Productivity

More than any other characteristic of any programming language, you can translate productivity to bottom-line dollars. Let each worker do more, and you can carry less staff. Work faster, and your application can deliver value to the business sooner. In most cases, productivity is *the most important consideration* for software development. Whether your project emphasizes quality, features, availability, or performance, productivity is the key to get you there. The best development teams build software in three stages:

- Make it work (delivery).
- Make it right (quality).
- Make it fast (performance).

You can't attack quality or performance without first getting your base running. And you certainly need to make an application available before you can make it highly available. It's all about evolution. You need to deliver tangible business value with every step. In this industry, we've learned that the most productive software development happens in smaller iterations. You simply can't be productive by building fast, clean applications with all possible features the first pass through your development cycle. You'll drown in the details, and you'll likely throw too much code away. It's far better to get something running and then improve it quickly.

You may be using Java because you think it's a clean language and it will save you time in the long run by improving your productivity over the long haul. You are betting that you can introduce new features faster, introduce fewer bugs, and fix the ones that sneak in more quickly.

Why Is Productivity So Important?

Here's an example from another industry. When Japan started building cars, they didn't build them very well. In fact, Japan had a reputation for building junk. They needed to improve quality, and the best path to do so was one iteration at a time.

Japan's quality improved after they applied the technique of Statistical Process Control (SPC) to their manufacturing. The inventor of SPC, Edward Walter Demming, tried to get the United States to adopt these methods in post–World War II America, and it didn't take hold; so,

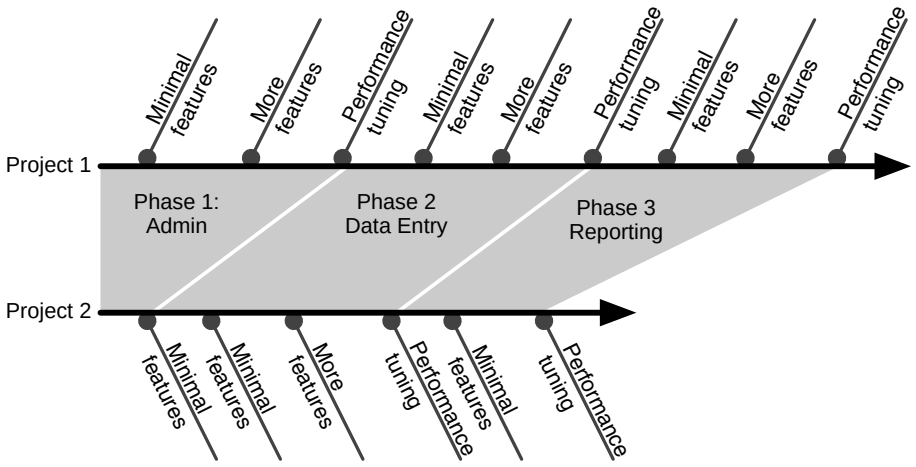


Figure 2.1: Project 1's team concentrated on unnecessary detail too soon

he took it to Japan, where they needed to rebuild their manufacturing infrastructure after the war. Demming was treated like a god in Japan. They applied his SPC techniques religiously in their automotive industry and eventually unseated Detroit as the top-selling auto manufacturer. They improved slowly, focusing on improving process, especially their cycle time between generations. With a shorter cycle time and leadership focused on process improvement, Tokyo's cars improved faster than Detroit's.

You can apply the same principle to software. With better productivity, you have more time to focus on improvements to features, performance, and quality, *based on the needs of the business*. Figure 2.1 shows the story. If you had three major pieces of an application to deliver (admin, data entry, and reporting) and you focused on building fast, perfect, feature-rich software for each major component, it would take you a certain amount of time—arbitrarily call it 15 units, if nothing went wrong.

If instead you put something in front of your customers after satisfying your initial set of requirements, you'd learn things from your customers that could save you time. Let's say that each initial iteration takes one unit of time. Assume your administrators say that the new admin con-

sole is fine; in fact, it's far better than the one they're using. Still, you know of some bugs to fix, so you allocate one unit of time for polishing work. Now, let's say your analysts do not like the reporting feature, finding it crude, limiting, and slow (four units). You'd need to do some rework, combined with some polishing and performance work. Customer service reps liked the look and feel of the system, but it could not keep up with the call volume. You'd have to improve performance (two units). You have delivered your software in less than half the time.

So, your goal isn't to be perfect right out of the gate—your goal is to get out of the gate quickly. Then, you iteratively improve. The quicker your iterations, the better off you are. With most applications, rather than anticipating the needs of your users, you should instead strive to get code in front of your users quickly. Then, based on their feedback and the needs of your business, you can apply your resources to address performance optimization, features, and other improvements. Take a lesson from Japan. Strive to cycle faster, and improve with each iteration. If you can do both, you'll beat your competition. That's true in manufacturing, and that's true in software.

Productivity of Core Java

In terms of productivity, in order to understand how Java comes up short, you have to know where it came from. In 1996, C++ was the dominant programming language for application development on server platforms. C++ was not a very productive language, but it *was* fast. At the time, we thought speed was more important than productivity. C++ had all the marketing momentum (from all the Unix vendors, Microsoft, and IBM, among others). C++ had the community. But when conditions are right, new programming languages emerge and old ones fade.

Any new programming language needs a catalyst to get the community rolling. Programming languages need a community in order to achieve widespread success, but it's hard to get new users without a community. When Sun created Java and embedded it into the Netscape Navigator Internet browser, they made some excellent compromises to ramp up a Java community in a hurry:

- Sun made Java look like C++. Java adopted a syntax like that of C++. With a C++-like language, Java didn't have to establish its own community from scratch. It could simply lure in C++ developers.

- Sun made Java act like C++ in a few important ways. Object-oriented languages let you build applications with a certain kind of building block: an object. Objects have both behavior and data, rolled up together. C++ cheats on object orientation, because some C++ elements, like characters and numbers, are not really objects. Java cheats in the same way.

- Sun copied a C++ feature called *static typing*. Static typing means that certain pieces of an application have one type, and you have to declare that type in advance. Many of the most productive languages for building applications use a different strategy, called *dynamic typing*.

In *Beyond Java*, I assert that Java's creators had to make these compromises to succeed. But compromises have two sides. By building a language that was closer to C++ than alternatives such as Smalltalk or Lisp, Sun was able to attract C++ users to the fledgling language. The downside of these compromises is productivity. C, and C++ by extension, was never designed to build applications. It was designed to build operating systems such as Unix. C++ was designed to be flexible and to produce fast systems code, not to productively build applications. We're now paying for the compromises:

- The C++ syntax, combined with Java's static typing, means programmers have to type too much—Java programs have two to four times the number of characters of similar programs in more dynamic languages such as Ruby. Many believe that shorter programs reduce maintenance costs proportionately.

- Java's static typing requires a compiler, so the compiler can check certain details for programmers, such as several forms of compatibility between two parts of a program. As a consequence, Java developers have to go through an extra compile step hundreds of times a day. Ruby developers don't.

- Java's primitives, are not object-oriented; this means that Java libraries must often be many times larger than libraries for purely object-oriented languages. For example, object-oriented languages have features to turn objects into XML. Similar Java programs have to deal with objects, but also characters, numbers, Booleans, and several other primitive types.

The available evidence to support programmer productivity for any language is remarkably scarce. One of the most compelling studies I've

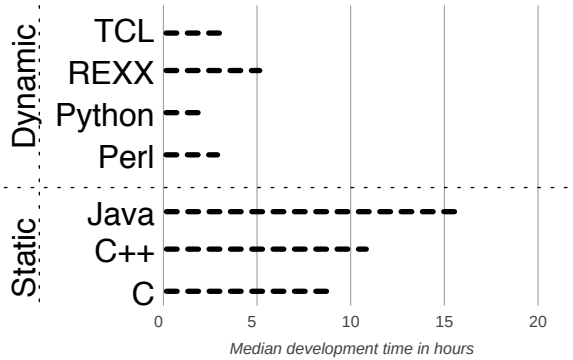


Figure 2.2: The productivity of seven different languages

seen on the topic is very old and does not relate directly to Ruby. I mention it here only because every single dynamically typed language, the so-called scripting languages, did much better than its statically typed peers. Figure 2.2 shows the results of a report[1] comparing productivity of dynamic languages such as Python, REXX, and Tcl to static languages such as C++ and Java. Surprisingly, Java is the worst language in the study, being around one third as productive as the scripting alternatives. Further, Java programs were generally two to three times as long as programs from the scripting languages. You could argue that tools and frameworks have gotten better since then, but we've also seen extensive bloating and a proliferation of frameworks. To many, these problems represent a productivity gash too large for any IDE to patch.

In Chapter 4, *Pilot*, on page 59, we'll look at a number of reasons that Ruby development can be many times as productive as Java development for certain problem domains. If you're interested but not convinced, you can do a prototype and measure productivity for yourself.

Productivity in Frameworks

You might think you could sacrifice some productivity when you're dealing with the low-level Java programming language and make up that time by using one of the thousands of Java frameworks. Usually, you'd be wrong.

[1] http://page.mi.fu-berlin.de/~prechelt/Biblio/jccpprt_computer2000.pdf

Essential complexity is the complexity required to do a job. If your application does tax returns, your application is going to be at least as complex as the requirements in the tax law. *Nonessential complexity*, also called *accidental complexity*, deals with the complexity you introduce to your environment. Martin Fowler, one of the most influential consultants in the computing profession, suggests Java frameworks introduce too much nonessential complexity. When you look at the history of enterprise computing in Java, you have to conclude he's right.

What's wrong with Java?—A discussion with Martin Fowler

Chief scientist of ThoughtWorks, author of Patterns of Enterprise Application Architecture

Q: What was your first object-oriented programming language?

Early in my career, I had an interest in objects. I worked with Smalltalk and C++. Most people at that time started with one or the other, but knowing both gave me a certain advantage. I got a distinctly schizophrenic view of the world. I was always conscious of the benefits of Smalltalk over C++. I didn't believe that people should use C++ for enterprise applications, but they did. When Java came along, we gained some, primarily because Java killed C++ for enterprise application development. But like many old Smalltalkers, I felt Java was a step backward from Smalltalk.

Q: What are the advantages of Java over Ruby?

With Java, you get sophisticated tools. I feel the pain when I have to leave the Intellij IDE. For companies, the stability of Java (and .NET, Java's separated-at-birth twin) is important, because post-COBOL, things were unstable with many languages and tools where it was hard to see what would last.

Q: What are the limitations of Java as you see them?

I usually hear people complain about static typing, which is important but not the whole story. Java guys spend too much time dealing with all the technical stuff that surrounds the core business behavior; this is complexity that's not helping to deal with the business problem. Our CEO (ThoughtWorks CEO Roy Singham) likes to taunt Java by saying it has failed the enterprise. The fact that we have to do all of this machinery means developers are

not thinking enough about business issues. They're thinking about how to move data in and out of a database, and in and out of a GUI.

Q: Where did Java go wrong?

There's so much of a push in the Java world to keep lesser-skilled developers from making messes, and that idea has failed. The idea that you can keep people from shooting themselves in the foot is appealing, but it's not what we've seen in practice. There are far too many overly complex bad Java apps out there.

Overall, Java is just too complicated. *No Silver Bullet* (Bro86) made the distinction between essential and accidental complexity. For example, if you're building a payroll system, the payroll business rules represent real complexity. But with Java, accidental complexity is most of the total effort. The EJB fiasco made this worse. EJB is a stunningly complex framework but is far too complex for most of the applications we see. Spring and Hibernate are a huge step forward, but there's still this nagging feeling that there's too much accidental complexity.

Q: Do you see any real alternatives to Java?

In the enterprise space in the last six years, a dark horse is stalking the .NET/Java duopoly: LAMP. LAMP stands for Linux, Apache, MySQL, and one of the *P-scripting* languages from PHP, Perl, and Python. In reality, LAMP has come to signify some combination of open source frameworks, plus a scripting language. By that definition, Ruby is an honorary part of the LAMP equation; you just have to ignore the downstroke on the *R*.

I used Perl for a little while, but I gave up when I couldn't even read some of the stuff I wrote. Some things in LAMP are now getting more interesting because the scripting languages are getting closer—application designs are getting better. I got into Python for a while, because objects were much more fluent, and it was much more dynamic than Java.

But don't forget that these are still early days. As I talk about this, we don't have much experience of Rails in the field.

At the moment it looks good, good enough to be worth exploring further, but until we see the practice in the field, we won't know for sure.

Q: Can Java be fixed?

That's an interesting question. You need to first answer the question, "What do they need to do to evolve the language to look more like the languages that feel more comfortable?" I'm not completely sure what that list is, but we could come up with such a list. But it's not just the language. It's all of the libraries. Can you meld those to take advantage of the new languages? What impact will new language fixes have on the frameworks? Are the frameworks going to be able to become more dynamic? Or are there so many frameworks that we've become locked into a static mind-set?

On another level, can people really replicate the design decisions that David Heinemeier Hansson (creator of Ruby on Rails) made? Take convention over configuration. It's one of those things where people can worry about all of the things that can go wrong instead of concentrating on what's going right. I like that decision. It's gutsy and so much against conventional wisdom. Lots of evidence suggests that it does work. Can that decision be brought into Java frameworks?

Q: Will Java be fixed?

I don't really care. Because one way or another, we win. My heart is behind the underdog, though.

EJB

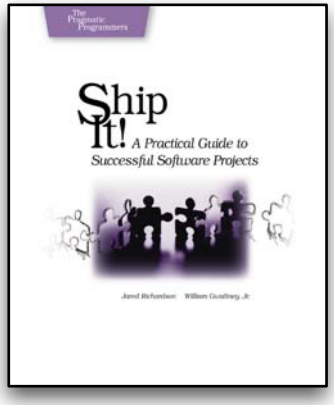
Since about 1999, most of the commercial Java brainpower has been focused on the hardest enterprise problems. In December 1998, the Java Community Process (JCP), unleashed the EJB (Enterprise JavaBeans) framework onto the Java developer population.

This terrible creature was very powerful, but that beast brought along too many of its own problems, including unmanageable complexity and poor performance.

Competitive Edge

Now that you've gotten an introduction to the individual practices of an agile developer, you may be interested in some of our other titles. For a full list of all of our current titles, as well as announcements of new titles, please visit www.pragmaticprogrammer.com.

Ship It!



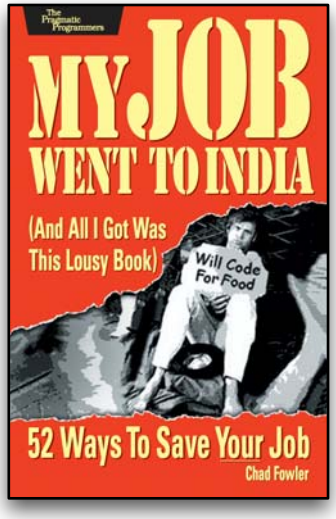
Agility for teams. The next step from the individual focus of *Practices of an Agile Developer* is the team approach that let's you *Ship It!*, on time and on budget, without excuses. You'll see how to implement the common technical infrastructure that every project needs along with well-accepted, easy-to-adopt, best-of-breed practices that really work, as well as common problems and how to solve them.

Ship It!: A Practical Guide to Successful Software Projects
Jared Richardson and Will Gwaltney
(200 pages) ISBN: 0-9745140-4-7. $29.95

My Job Went to India

World class career advice. The job market is shifting. Your current job may be outsourced, perhaps to India or eastern Europe. But you can save your job and improve your career by following these practical and timely tips. See how to: • treat your career as a business • build your own brand as a software developer • develop a structured plan for keeping your skills up to date • market yourself to your company and rest of the industry • keep your job!

My Job Went to India: 52 Ways to Save Your Job
Chad Fowler
(208 pages) ISBN: 0-9766940-1-8. $19.95



Visit our secure online store: http://pragmaticprogrammer.com/catalog

Cutting Edge

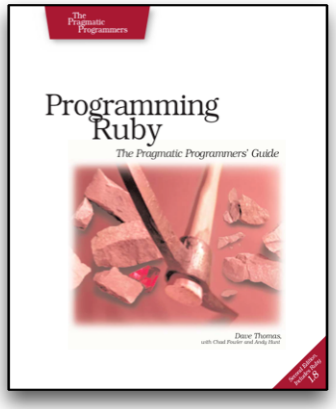
Learn how to use the popular Ruby programming language from the Pragmatic Programmers: your definitive source for reference and tutorials on the Ruby language and exciting new application development tools based on Ruby.

The *Facets of Ruby* series includes the definitive guide to Ruby, widely known as the PickAxe book, and *Agile Web Development with Rails*, the first and best guide to the cutting-edge Ruby on Rails application framework.

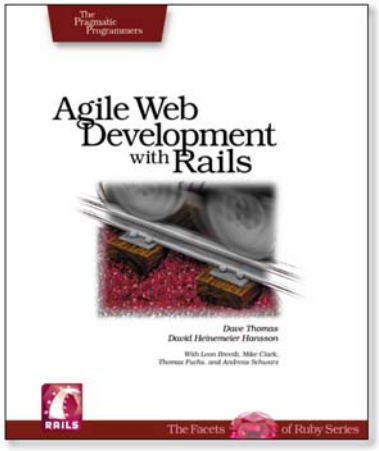
Programming Ruby (The PickAxe)

The definitive guide to Ruby programming. • Up-to-date and expanded for Ruby version 1.8. • Complete documentation of all the built-in classes, modules, methods, and standard libraries. • Learn more about Ruby's web tools, unit testing, and programming philosophy.

Programming Ruby: The Pragmatic Programmer's Guide, 2nd Edition
Dave Thomas with Chad Fowler and Andy Hunt
(864 pages) ISBN: 0-9745140-5-5. $44.95



Agile Web Development with Rails



A new approach to rapid web development. Develop sophisticated web applications quickly and easily • Learn the framework of choice for Web 2.0 developers • Use incremental and iterative development to create the web apps that users want • Get to go home on time.

Agile Web Development with Rails: A Pragmatic Guide
Dave Thomas and David Heinemeier Hansson
(570 pages) ISBN: 0-9766940-0-X. $34.95

Visit our secure online store: http://pragmaticprogrammer.com/catalog

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help programmers stay on top of their game.

Visit Us Online

Java to Ruby Home Page
`pragmaticprogrammer.com/title/fr_j2r`
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
`pragmaticprogrammer.com/updates`
Be notified when updates and new books become available.

Join the Community
`pragmaticprogrammer.com/community`
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
`pragmaticprogrammer.com/news`
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: `pragmaticprogrammer.com/title/fr_j2r`.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	`www.pragmaticprogrammer.com/catalog`
Customer Service:	`support@pragmaticprogrammer.com`
Non-English Versions:	`translations@pragmaticprogrammer.com`
Pragmatic Teaching:	`academic@pragmaticprogrammer.com`
Author Proposals:	`proposals@pragmaticprogrammer.com`