Extracted from:

From Java to Ruby

Things Every Manager Should Know

If you want to make enemies, try to change something.
► Woodrow Wilson

Chapter 4

Pilot

4.1 Building Your Plan

The first time you use a new language in anger, to solve a real business problem, there's much at stake. If you're wildly successful, you can pave the way for change. If you fail miserably, you may never get another chance. If you're convinced that Ruby can help, you'll need to carefully plan this initiation.

Identifying a Business Problem

You'll choose your business problem based on the political environment and technical demands. You'll need to strike a balance across at least two axes: the political and the technical. If you get too conservative and pick a problem that's too easy or too small, you won't prove much, and no one will take notice. Conversely, if you get so aggressive that you hit technical obstacles beyond your abilities, you could fail, or experience an ugly success and get swept away by politics. The interview on page 64 tells the story. As a problem gets more difficult, risks get higher, but you learn more. As politics and visibility increase, your potential reward gets higher, but consequences also increase.

Choosing a Technical Problem

Combined with the people you put on the project, the technical characteristics of your problem, more than any other characteristics, will determine your success or failure. A good tool, used for the wrong job, is a bad tool. Chapters 5 and 6 will explore Ruby technologies in greater detail, but let's look beyond the simplistic view of picking the right job for the tool. Figure 4.1, on the next page shows that your political goals will help determine the technical problem you choose:

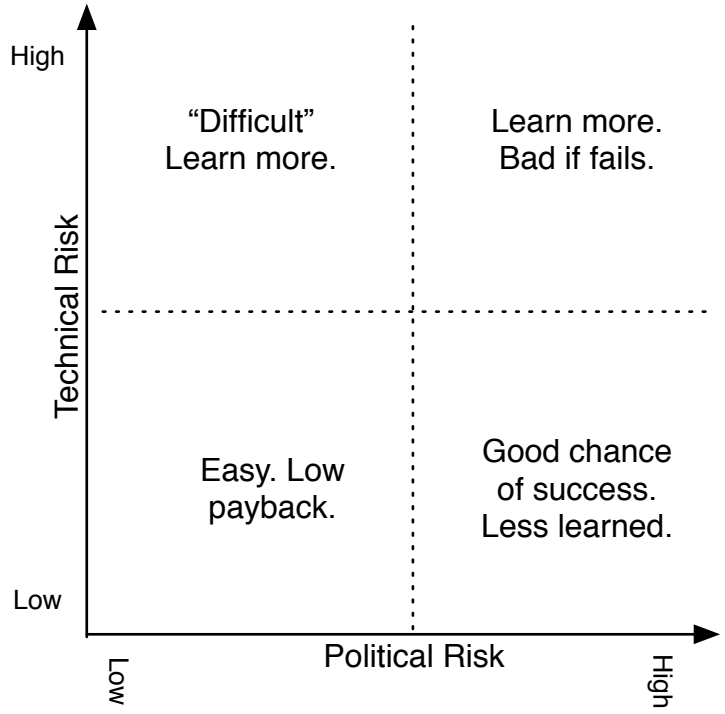


Figure 4.1: High technical risks are ideal for learning but bad for politics

- *Selling.* If you've already chosen Ruby and you're looking to establish early success to sell the framework, you want the easiest possible problem and best possible fit.
- *Learning.* If you're more interested in learning about how far you can push Ruby or whether it's the best technical fit, then you'll want to pick a more demanding technical problem, and you'll want to see how far you can bend Ruby to suit your purposes.

In other words, you can't always have the perfect pilot for both selling and learning. Since you'll usually want to do some of both, you'll have to strike a compromise. As I lay out individual scenarios from teams who have built successful Ruby pilots, you'll see how each team considered both the technical and political realities of breaking new ground. You'll have to do the same:

- You'll be learning a new technology. You'll want to show off your productivity, but make sure you allow some time for your team to play and learn so you'll know more about the environment and have a better experience once it's time to apply your knowledge to a real-world problem.
- You'll often be tempted to explore some aspect of Ruby that other teams have never tried before, but you'll be much better off if you stick to problems that others have solved before you for your first project. The Ruby community is open and accommodating. Ask them whether anyone has tried a given problem before.
- You'll be anxious to prove that you can work on a thinner budget, but be careful. If you need help, get it. Working with a Ruby expert on a short-duration pilot project will save you thousands of dollars in the long run. You won't save any money if you attack too much of a problem before you're ready.
- Keep sight of your political goals. If Java is not working in your environment, your first goal is to establish a working alternative. If serious Java proponents exist, technical failure will be devastating, so you'll want to start slow. If you have more freedom to fail, you can push the technology into more demanding applications.

Building a Team

After you've chosen a problem, you'll need to build a team. The best teams for Ruby have a few common characteristics:

- *They often have some experience with dynamic languages.* If your programmers have written Smalltalk, Lisp, Python, or Perl before, they'll be able to take better advantage of Ruby. If they've used Java's dynamic features such as reflection or aspect-oriented programming, they'll fare better than those who haven't.
- *They are small.* You don't need nearly as many developers as you need for similar Java problems.
- *They have freedom.* If your technical staff is free to make their own decisions, they'll make progress more quickly.

Small, smart teams play to the strengths of dynamic languages. In the rest of the chapter, we'll lay out scenarios that have been successful for other Ruby teams. They will range from simple to complex on a technical scale and from low to high visibility on a political scale. See how others introduced Ruby.

4.2 Scenario 1: Classic Pilot

Using this scenario, your goal is to learn enough about Ruby to make a go or no-go decision. Frankly, this scenario is not quite as common as you'd expect. Since Java is near its peak popularity, language advocates typically need to be stronger and more creative than those for other new technologies. Still, a pilot project can often tell you everything you need to know about whether Ruby is a technical fit.

Profile

This scenario differs from the others in this chapter because the primary goal is to learn, rather than sell. Figure 4.2, on the following page, shows that when you look at the profile for this type of application, the critical axis is the technical one. Although different people in your organization may well have different motivations, if your goal is to make a decision rather than validate your decision with a successful pilot, you're going to want to choose an application with enough technical challenges to make your decision. The political visibility of the project doesn't matter as much. For the best of cases, you should reduce your risk by picking a project with limited consequences for failure.

Example: A Manufacturing Monitor at Autobar Flexible Holland B.V.

Sake Lemstra, a managing director at Autobar Flexible Holland B.V., asked Henri ter Steeg of LinkIT Group to develop an application to help collect data on his manufacturing process. The interview on page 64 tells the story. Originally, Henri used Java with Swing to develop the application, based on the popularity of Java and the excellent enterprise integration features, including good Oracle integration. The performance was adequate and the interface was rich, but the application became increasingly complex as new requirements came.

They decided to pick a new technology and considered several Java solutions, but none of them seemed to offer enough of an advantage to make a rewrite worthwhile. Henri noticed the new Ajax support in Ruby on Rails, so he decided to give Rails a try. Henri was concerned about several aspects of the application that would stretch the Ruby platform:

- The performance would have to be good, because the application would post graphs with thousands of data points every minute.

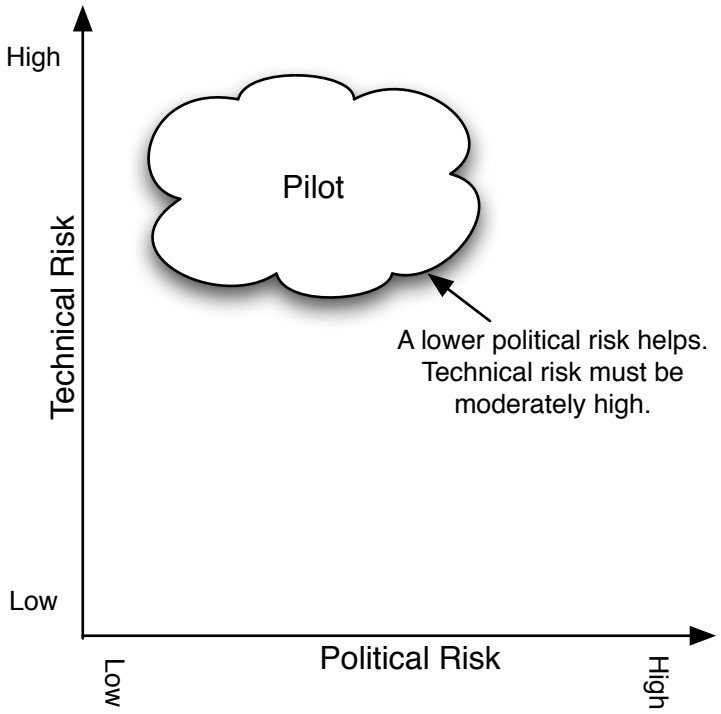


Figure 4.2: Classic pilots need medium to high technical risk

- The customer had a version of Oracle that was not yet supported by Ruby on Rails, so they'd need to add specific support for Oracle features such as their outer join syntax.
- The user interface would need to be much more powerful than typical HTML applications.

Henri decided to implement a small slice of the application but one that dove deeply into technical details. He implemented a classic spike, which is a thin but technically demanding slice of the application to make sure Ruby on Rails would work. A custom caching layer solved the performance concerns. Ajax support, in conjunction with existing C code, handled the graphing functions needed by the sophisticated interface requirements. The flexibility of both Ruby and Rails allowed him to change the outer join syntax and add a simple real-time cache to his objects by changing the definition of classes in the core frame-

works. This capability, called *open classes*, lends an incredible flexibility to Ruby, compared to Java. Because of excellent productivity and a much simpler application base, they decided to move ahead with Ruby on Rails. His customer was more interested in a clean, working application on a better timeline than forcing a Java agenda, so he moved development operations to Ruby.

Drawing Conclusions

Henri's implementation was a classic pilot. He was most interested in improving productivity and saw Ruby on Rails as one possible candidate to get him there. He was not concerned about political fallout, because had he failed, he would have simply chosen an alternative technology. This was strictly a learning exercise. The pilot was quite successful. Even after dealing with these framework limitations, Henri estimates that he is roughly four times as productive on Rails as he was in Java.

Rescue scenario—A discussion with Henri ter Steeg

LinkiT Group

Q: Would you describe your application?

We had a suffering Java application. The application was written in Swing. It worked OK, but it was getting too complicated. The application had to communicate with other clients, and we couldn't get it to query the database with good performance. It got to the point where we were afraid to add more features, because the application was out of control. So, we decided to move the application to another framework.

We looked at other Java frameworks first. We considered the *Spring Rich Client* framework, but it looked unfinished. We also considered a handful of other Java frameworks.

We ran across Rails. We considered it briefly and then put it aside until the Ajax stuff came along. I thought, "This is really impressive." I did a spike and was really surprised with the speed of development. I could develop client-server applications very fast. I got back to Java and said, "This is too difficult, and it's too much work."

On Rails, my development speed was excellent.

Q: What were some of the limitations?

We needed to use abstract data types for performance reasons. In JDBC, you can use them, but you can't in Rails. So, we created some views and "instead of" triggers and got past the problem. We also used Oracle drivers that did not support the outer join syntax. Active Record did not support outer join syntax, so we just created some classes and changed one of the Rails methods at run time. It was easier than I expected to work around what looked like serious problems at the time.

Q: What are you using for security?

At the moment, we're just using Apache security. We did not use the Rails generator. For now, Apache security is all we need.

Q: Have you been pleased with the performance?

For the most part, it has been good. We had one major performance issue. With Java, we could use a Swing app, which did some caching for us for performance reasons. This approach is difficult with a web app. We solved it with time-based caching.

Q: Did you experience any resistance to Rails?

No. The business environment trumped everything else. Our code is simpler and easier to maintain, and we can work faster with it.

Q: What were your top business priorities?

We wanted a stable application. We wanted to develop faster, and we wanted a simpler code base. As a very rough guess, I'd say we're coding three to four times faster.

Q: What was the most interesting part of your application?

We wrote a C fragment to produce our business graph. The business logic gets one data point for each minute, and we accumulate many data points over time. Doing a fresh database query every minute was not an option. In the Swing application, we cached the entire graph. Now, database triggers write the new data points to a file. We have a CGI script that reads the file and plots the graph.

4.3 Scenario 2: Trojan Horse

With the Trojan horse strategy, your goal is to get a working pilot established with minimal visibility and leverage that tiny success to increase Ruby penetration. Unlike the classic pilot scenario, you're not as interested in learning from the experience. You've already established that Ruby can help by other means, and you're seeking to establish some success with an easy project with a political climate that will not provide much resistance. If you later choose to do so, you can grow your advocates internally and leverage your success for better Ruby penetration on more important applications in the future.

Profile

The key to the Trojan horse scenario is to get your initial pilot, or Trojan horse, established with as few technical and political obstacles possible. To do so, you work Ruby into an organization with very little management visibility and feed the Ruby development culture. You'd prefer an application with little technical or political risk, as in Figure 4.3, on the following page, choosing instead to fly Ruby in beneath the radar. Culture is often the most important element. You want to build a groundswell of support for the language from the bottom up.

Keep in mind that you'll need to take on enough technical risk to prove something important—you won't prove anything by building an application to manage your ten contacts with a primitive web page. Often, you're looking to save time and money by efficiently handling tedious, unimportant jobs, where technology choice might not be as much of an issue. These applications are often inward facing and may support other applications. Admin consoles, tests, and build tools are likely targets for the Trojan. You can then promote your success and leverage that success to take on other Ruby on Rails projects.

Example: Amazon.com

The Amazon.com interest in Ruby has been well publicized by several bloggers, including Steve Yegge, who worked at Amazon.com for seven years (and provided much of the background for this scenario), and David Heinemeier Hansson, the creator of Ruby on Rails. Ruby was initially not a popular or approved language at Amazon.com, but they have since hosted the Seattle-area Ruby user group. To establish Ruby, its proponents simply got Ruby into the hands of developers who could use it. The tools support team made sure Ruby was installed on all

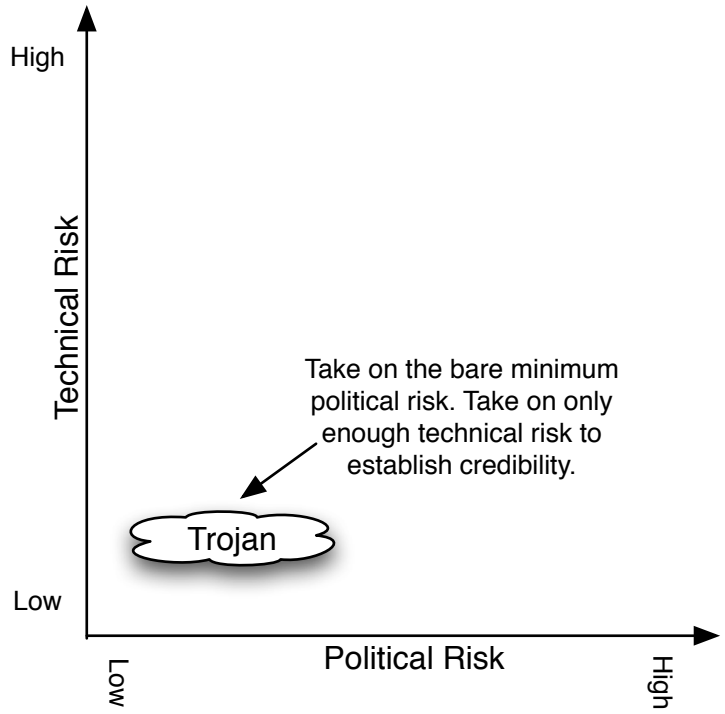


Figure 4.3: The Trojan Horse scenario requires low risk on both axes

developer machines. They brought in Dave Thomas to do a brown-bag talk on Ruby. Dave didn't actually promote using Ruby at Amazon.com; instead, he promoted learning alternative languages to learn how to program better. They gave away 120 free copies of *Programming Ruby* [TFH05] and eventually built a ground swell of support at Amazon.com. Ruby found increasing use in small, inwardly facing projects. Ruby is thriving today at Amazon.com as an accepted language, though it does not do most of the heavy lifting.

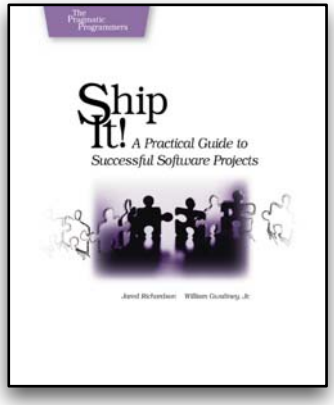
Example: An Administrative Console

I recently encountered another excellent opportunity for the Trojan horse strategy. I was doing a Java training course at a consultancy. They provided application developers to conservative companies. Nearly all their development was done in the Java programming language. One application in particular was done with old technologies, including EJB.

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!



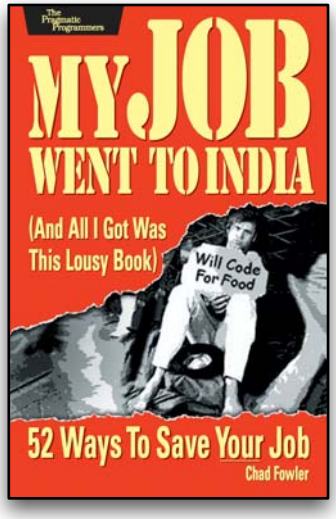
Agility for teams. The next step from the individual focus of *Practices of an Agile Developer* is the team approach that let's you *Ship It!*, on time and on budget, without excuses. You'll see how to implement the common technical infrastructure that every project needs along with well-accepted, easy-to-adopt,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

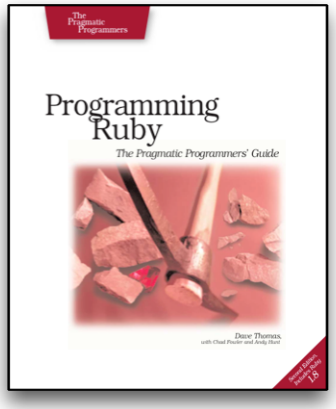
Learn how to use the popular Ruby programming language from the Pragmatic Programmers: your definitive source for reference and tutorials on the Ruby language and exciting new application development tools based on Ruby.

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.

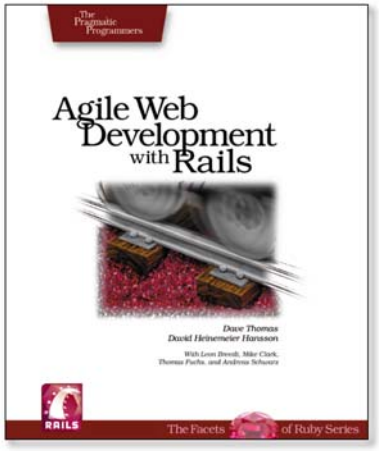
Programming Ruby (The PickAxe)

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`