Extracted from:

Learn to Program

The Pragmatic Programmers

Learn to Program

Chris Pine

The Facets of Ruby Series

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragmaticprogrammer.com

Printed in the United States of America.

ISBN 0-9766940-4-2

Printed on acid-free paper.

Fourth printing, January 2009

Version: 2009-3-13

Introduction

I vividly remember writing my first program. (My memory is pretty horrible; I don't vividly remember many things, just things like waking up after oral surgery, or watching the birth of our children, or that time I was trying to flirt with this girl when she tells me that my zipper is down, or setting my shoes on fire in my middle-school P.E. class, or writing my first program…you know, things like that.)

I suppose, looking back, that it was a fairly ambitious program for a newbie (20 or 30 lines of code, I think). But I was a math major, after all, and we are supposed to be good at things like "logical thinking." So I went down to the Reed College computer lab, armed only with a book on programming and my ego, sat down at one of the Unix terminals there, and started programming. Well, maybe "started" isn't the right word. Or "programming." I mostly just sat there, feeling hopelessly stupid. Then ashamed. Then angry. Then just small. Eight grueling hours later, the program was finished. It worked, but I didn't much care at that point…it was not a triumphant moment.

It has been more than a decade, but I can still feel the stress and humiliation in my stomach when I think about it.

Clearly, this was *not* the way to learn programming.

But why was it so hard? I mean, there I was, this reasonably bright guy with some fairly rigorous mathematical training—you'd think I would be able to get this! And I did go on to make a living programming, and even to write a book about it, so it's not like I just "didn't have what it took" or anything like that. No, in fact, I find programming to be pretty easy these days, for the most part.

So why was it so hard to tell a computer to do something only mildly complex? Well, it wasn't the "mildly complex" part that was giving me problems; it was the "tell a computer" part.

In any communication with humans, you can leave out all sorts of steps or concepts and let them fill in the gaps. In fact, you have to do this! We'd never be able to get anything done otherwise. The typical example is making a peanut butter and jelly sandwich. Normally, if you wanted someone to make you a peanut butter and jelly sandwich, you might simply say, "Hey, could you make me a peanut butter and jelly sandwich?" But if you were talking to someone who had never done it before, you would have to tell them how:

1. Get out two slices of bread (and put the rest back).
2. Get out the peanut butter, the jelly, and a butter knife.
3. Spread the peanut butter on one slice of bread and the jelly on the other one.
4. Put the peanut butter and jelly away, and take care of the knife.
5. Put the slices together, put the sandwich on a plate, and bring it to me. Thanks!

I imagine those would be sufficient instructions for a small child. Small children are needlessly, recklessly clever, though. What would you have to say to a computer? Well, let's just look at that first step:

1. a) Locate bread.
   b) Pick up bread.
   c) Move to empty counter.
   d) Set down bread on counter.
   e) Open bag of bread.

   ...

But no, this isn't nearly good enough. For starters, how does it "locate bread"? We'll have to set up some sort of database associating items with locations. The database will also need locations for peanut butter, jelly, knife, sink, plate, counter....

Oh, and what if the bread is in a bread box? You'll need to open it first. Or in a cabinet? Or in your fridge? Perhaps behind something else? Or what if it is *already on the counter??* You didn't think of that one, did you? So now we have this:

- Initialize item-to-location database.
- If bread is in bread box:
  - Open bread box.
  - Pick up bread.
  - Remove hands from bread box.
  - Close bread box.

- If bread is in cabinet:
  - Open cabinet door.
  - Pick up bread.
  - Remove hands from cabinet.
  - Close cabinet door.

  ...

And on and on it goes. What if no clean knife is available? What if there is no empty counter space at the moment? And you'd better pray to whatever forces you find comfort in that there's no twist-tie on that bread!

Even steps such as "open bread box" need to be explained...and this is why we don't have robots making sandwiches for us yet: it's not that we can't build the robots; it's that we can't program them to make sandwiches. It's because making sandwiches is *hard* to describe (but easy to do for smart creatures like us humans), and computers are good only for things that are (relatively) *easy* to describe (but hard to do for slow creatures like us humans).

And that is why I had such a hard time writing that first program: computers are way dumber than I was prepared for.

What Is Programming?

When you teach someone how to make a sandwich, your job is made much easier because they already know what a sandwich is. It is this common, informal understanding of "sandwichness" that allows them to fill in the gaps in your explanation. Step 3 says to spread the peanut butter on one slice of bread. It doesn't say to spread it on only one side of the bread or to use the knife to do the spreading (as opposed to, say, your forehead). You assume they just know these things.

Similarly, I think it will help to talk a bit about what programming is, in order to give you a sort of informal understanding of it.

Programming is telling your computer how to do something. Large tasks must be broken up into smaller tasks, which must be broken up into still smaller tasks, down until you get to the most basic tasks that you don't have to describe, the tasks your computer already knows how to do. (These are *really* basic things such as arithmetic or displaying some text on your screen.)

My biggest problem when I was learning to program was that I was trying to learn it backward. I knew what I wanted the computer to

do and tried working backward from that, breaking it down until I got to something the computer knew how to do. Bad idea. I didn't really know what the computer *could* do, so I didn't know what to break the problem down to. (Mind you, now that I do know, this is exactly how I program these days. But it just doesn't work to start out this way.)

That's why you're going to learn it differently. You'll learn first about those basic things your computer can do (a few of them), and then find some simple tasks that can be broken down into a few of these basic things. *Your* first program will be so easy, it won't even take you a minute.

Programming Languages

In order to tell your computer how to do something, you must use a programming language. A programming language is similar to a human language in that it's made up of basic elements (such as nouns and verbs) and ways to combine those elements to create meaning (sentences, paragraphs, and novels). There are many languages to choose from (C, Java, Ruby, Perl...), and some have a larger set of those basic elements than others. Ruby has a fine set and is one of the easiest to learn (as well as being elegant and forgiving and the name of my daughter, and so forth), so we'll use that one.

Perhaps the best reason for using Ruby is that Ruby programs tend to be short. For example, here's a small program in Java:

```
public class HelloWorld {
  public static void main(String []args) {
    System.out.println("Hello World");
  }
}
```

And here's the same program in Ruby:

```
puts 'Hello World'
```

This program, as you might guess from the Ruby version, just writes `Hello World` to your screen. It's not nearly as obvious from looking at the Java version.

How about this comparison: I'll write a program to do *nothing*! Nothing at all! In Ruby, you don't need to *write* anything at all; a completely blank program will work just fine.

In Java, though, you need all this:

```
public class DoNothing {
  public static void main(String[] args) {
  }
}
```

You need all that just to do nothing, just to say, "Hey, I am a Java program, and I don't do anything!" So that's why we'll use Ruby. (My first program was *not* in Ruby, which is another reason why it was so painful.)

The Art of Programming

An important part of programming is, of course, making a program that does what it's supposed to do. In other words, it should have no bugs. You know all this. However, focusing on correctness, on bug-free programs, misses a lot of what programming is all about. Programming is not just about the end product; it's about the process that gets you there. (Anyway, an ugly process will result in buggy code. This happens every time.)

Programs aren't just built in one go, like a bridge. They are talked about, sketched out, prototyped, played with, refactored, tuned, tested, tweaked, deleted, rewritten....

A program is not built; it is grown.

Because a program is always growing and always changing, it must be written with change in mind. I know it's not really clear yet what this means in practical terms, but I'll be bringing it up throughout the book.

Probably the first, most basic rule of good programming is to avoid duplication of code at all costs. This is sometimes called the *DRY rule*: Don't Repeat Yourself.

I usually think of it in another way: a good programmer cultivates the virtue of laziness. (But not just any laziness: you must be aggressively, proactively lazy!) Save yourself work whenever possible. If making a few

changes now means you'll be able to save yourself more work later, do it! Make your program a place where you can do the absolute minimum amount of work to get the job done. Not only is programming this way much more interesting (it's very boring to do the same thing over and over and over...), but it produces less buggy code, and it produces it faster. It's a win-win-win situation.

Either way you look at it (DRY or laziness), the idea is the same: make your programs flexible. When change comes (and it *always* does), you'll have a much easier time changing with it.

Well, that about wraps it up. Looking at other technical books I own, they always seem to have a section here about "Who should read this book" or "How to read this book" or something. Well...I think *you* should read it, and front-to-back always works for me. (I mean, I did put the chapters in this order for a reason, you know.) Anyway, I never read that crap, so let's program!

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

`pragmaticprogrammer.com/titles/fr_ltp`
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/titles/fr_ltp`.

Contact Us

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