Chapter 1

A single paragraph

However, it isn’t sufficient just to have a group of people studying the craft of software development: those have existed as long as software has existed and if they were capable of closing this gap, they would have done so by now. A lack of sufficient context is what most frequently leads to tribalism in software communities, and the end result is a polarizing effect that causes direct harm to real progress by generating far more heat than light whenever a certain tool or technique is discussed.

Chapter 2

Multiple paragraphs

Becoming a better software developer isn't easy. While there are plenty of resources to help beginners understand some fundamental skills, this only establishes a basic level of proficiency. Expert developers are orders of magnitude more productive than even somewhat advanced beginners, but rarely produce resources that help folks climb out of the vast chasm that represents the population of intermediate developers.

Passive learning materials are of marginal assistance to intermediate developers. While reading a new book or blog might help familiarize someone with a new tool or technique, the gains from this are superficial and do not lead to the sort of plateau-busting realizations that are necessary in order to break through the many walls that stand between proficiency and mastery.

What intermediate developers really need is a bit of guidance to discover the right kinds of problems to be working on, the sorts of things that will cause them to ask the right questions, even if the answers to those questions lie somewhere off in the distance. They also need support from other developers who have traveled a bit farther down the path, so that they can learn about common pitfalls or get warned about dead ends before barreling down them at breakneck speeds.

Software development is hard, and that's not likely to change any time soon. The necessary complexity of most computing problems is significant enough that there will still be more than enough challenges for a lifetime even if we never repeat the same mistakes others have made. A single experienced developer can make a massive impact simply by listening to what problems others are experiencing and giving helpful feedback based on what he or she has learned the hard way. A community of folks dedicated to providing the same sort of support is even better.

However, it isn't sufficient just to have a group of people studying the craft of software development: those have existed as long as software has existed and if they were capable of closing this gap, they would have done so by now. A

lack of sufficient context is what most frequently leads to tribalism in software communities, and the end result is a polarizing effect that causes direct harm to real progress by generating far more heat than light whenever a certain tool or technique is discussed.

It is necessary to seek deeper, more personalized connections so that both student and mentor have a stronger shared understanding of the surrounding context of any given problem.

RbMU exists because we understand just how much of an impact that software developers have on our world. While technology itself is neutral, we feel like given the skills and resources, those who write code can make a massive positive impact on the way that we live our lives. By empowering those who may not have otherwise reached their full potential, we can touch the world in a much more significant way as a community than any of us could on our own.

To sum it all up, we believe that teaching changes lives, and software changes lives. So teaching software developers changes lives that change lives so that we can change lives while we change lives. Of course, we also do this because it’s fun. :)

Chapter 3

Preformatted text blocks

I decided to start off this newsletter with one of the most basic but essential pieces of knowledge you can have about Ruby's object model: the way it looks up methods. Let's do a little exploration by working through a few examples.

Below we have a simple report class who's job is to perform some basic data manipulations and then produce some text output.

```
class Report
  def initialize(ledger)
    @balance          = ledger.inject(0) { |sum, (k,v)| sum + v }
    @credits, @debits = ledger.partition { |k,v| v > 0 }
  end

  attr_reader :credits, :debits, :balance

  def formatted_output
    "Current Balance: #{balance}\n\n" +
    "Credits:\n\n#{formatted_line_items(credits)}\n\n" +
    "Debits:\n\n#{formatted_line_items(debits)}"
  end

  def formatted_line_items(items)
    items.map { |k, v| "#{k}: #{'%.2f' % v.abs}" }.join("\n")
  end
end
```

The following code demonstrates how we'd make use of this newly created class.

```
ledger = [ ["Deposit Check #123", 500.15],
           ["Fancy Shoes",       -200.25],
           ["Fancy Hat",          -54.40],
           ["ATM Deposit",       1200.00],
           ["Kitteh Litteh",       -5.00] ]

report = Report.new(ledger)
puts report.formatted_output
```

Amazing, right? (Yadda... yadda.. yadda)

Chapter 4

Section headings

When we know something is ugly or "evil", we're quick to replace it with what we know to be a better solution. But if we don't know *why* the solution is better, it makes it hard for us to investigate those things that look reasonable on the surface, yet have weaknesses just beneath the skin.

I'll run through a quick example of what I mean, and then, it'll be your turn to run with it and report back here. While I don't use this approach all the time, I find it's a neat tool to have handy when you're trying to push your understanding of your code just a little bit farther.

Continuations are Evil?

It's pretty common knowledge that continuations in Ruby are evil. But let's pretend we didn't know that. We might end up catching ourselves writing code like this:

```
table.each do |row|
  call_cc do |next_row|
    row.each do |field|
      if field.valid?
        do_something_with(field)
      else
        STDERR.puts "skipping a bad row"
        next_row[]
      end
    end
  end
end
```

The continuation approach actually doesn't look so bad. All we're doing is iterating over a two-dimensional structure and skipping ahead to the next row if there is a problem with any of the fields. But having to pass along the next_row callback seems a bit heavy handed, and also makes one wonder what sort of magic it might be concealing. Preserving the same basic flow, we could be using catch / throw instead, lowering the conceptual baggage:

```
table.each do |row|
  catch(:next_row) do
    row.each do |field|
      if field.valid?
        do_something_with(field)
      else
        STDERR.puts "skipping a bad row"
        throw :next_row
      end
    end
  end
end
```

So, now, we have something that looks more-or-less like a transactional block, which gets aborted when you throw :next_row. Not too bad, but if you're like me, you probably get annoyed about having your eyes bounce up and down while you trace the flow, in both of these examples. We can fix that, of course:

```
def process(row)
  row.each do |field|
    if field.valid?
      do_something_with(field)
    else
      STDERR.puts "skipping a bad row"
      return
    end
  end
end

table.each { |row| process(row) }
```

Now this is more like it! This code is probably the same as what a beginner might write, but it's also the clearest out of what we've seen here. But without it as a point of reference, neither the catch / throw or callcc solutions would look terrible.

Through this quick little set of examples, I found a good reason *not* to use continuations for this particular use case. Arriving back at a simple solution by starting with a "clever" one and reducing it down to the fundamentals was a refreshing experience for me.

Your Homework

Now it's your turn. I'm encouraging you to ignore "Best Practices" and the common idioms that we often accept as gospel.

Pick one technique that's typically considered a no-no and do the best you can to use it in a somewhat reasonable way. You don't want to waste your time with something that not even the most muddy-eyed coder would touch with a ten-foot pole, as you won't learn much that way. Instead, work with something

that looks fine or even clever at a first glance.

Then, try to fix up what you built by using more idiomatic, simpler code. Your goal is to create something that still looks good, but isn't as much as a liability (conceptually) as what you started with. If you fail to do so, the worst thing that could happen is that you may have come across a new idiom worth sharing with others. But more likely, with a little effort you'll have no trouble uncovering the wisdom behind whatever idiom you were putting to the test.

What's the difference? Now, something that used to be a somewhat arbitrary rule to you is now common sense. As contexts shift, you may need to conduct new experiments, but repeat this exercise often enough and you'll start to see a powerful intuition develop that allows you to internalize your design decisions rather than running through a punch list of patterns. And at least for me, I find that experience a lot more fun. I hope you will, too.

If you try out this exercise, please share what you've done, either via a link to your own blog, or a gist I'm interested in how many bad ideas we can rack up here.

Chapter 5

Unordered lists

Please note the following guidelines when making your proposal.

- You will be evaluated on work done via the 3/7-3/28 period only, so be sure to focus your proposal on the areas of your project that you plan to make progress on during that time period
- The core skills course involves a considerable amount of assigned work that needs to be completed in addition to your individual project. To avoid feeling overwhelmed, choose an individual project that is only slightly challenging based on your current level of experience.
- You are not expected to build a feature-complete complete application, but each checkpoint requires you to produce real, functioning features rather than just raw structures or rough prototypes.
- Your project must use a GPLv3 compatible free software license. Both the MIT and BSD licenses are popular in Ruby projects, and are the best choice if you don't know or care about the finer details of the other available options.
- Your project can be pretty much anything Ruby related, as long as it does something useful, and involves writing a reasonable amount of Ruby code.
- Up to three students can work on the same project, but each must have a clearly defined proposal for the work they plan to do, and each proposal will be evaluated individually.
- You must use a public git repository for developing your individual project (typically done via github)

If you have any questions about these requirements feel free to let me know. The individual project is one of the most important things you can do at RbMU, so we want to do all that we can to help you succeed.