RubyGuides

3 Principles for Writing Great Ruby Code

How do you write **great Ruby code**?

There isn't a simple answer.

But after reading a few books on the topic, I came up with 3 principles that will guide you towards better code.

If you don't have any **guiding principles** then you don't know what to aim for.

It's like going to a new city without having a map.

I want to give you this map.

Let's start with the **1st principle**...

Principle 1 - Avoid Misunderstanding

Misunderstanding is the biggest cause of mistakes.

For example, if I tell you over the phone "buy bacon", but you get oranges instead that's a misunderstanding.

How does misunderstanding happen in code?

It happens when variables don't have descriptive names, when you read outdated comments, when you have to spend too much time trying to understand a section of code.

This confusion & misunderstanding leads to a **lack of readability**.

It makes you waste a lot of time.

You can **fix this** in a few ways:

- Use good names for your methods & variables (I know, this is harder than it sounds, but there are a few rules you can follow)
- Delete or update outdated comments
- Group & organize your code into logical sections

Example:

```
if name =~ /\A[A-Za-z]{3,}\d+\z/
  # ...
end
```

You may be a master of regular expressions, but even then you won't know the intention of this code.

Why do we need this regexp?

You want to make your code so obvious that you don't have to think about its meaning.

The meaning becomes crystal clear, comments become unnecessary, your code becomes **self-documenting**.

After refactoring:

```ruby
VALID_USER_NAME = /\A[A-Za-z]{3,}\d+\z/

if name =~ VALID_USER_NAME
  # ...
end
```

Now we know the purpose of this code, to check if some string contains a valid username or not.

Case closed.

No more investigation or guessing necessary.

Principle 2 - Create More Classes

Creating more classes is the key to breaking down big classes & methods into smaller classes.

A class can become so big that we call it a **God class**...

...it knows everything, it's everywhere & has all the most important functionality.

You can't let a single class have so much control over your system.

So don't fear **creating more classes**.

As Steve McConnell said:

> "The single most important reason to create a class is to reduce a program's complexity"

Example:

```
class Game
  def display_board
  end

  def save_state
  end

  def make_move
  end

  def valid_move?
  end
end
```

This code has methods with different concerns, different goals. Displaying the board is **very different from saving the game state**, or making a move, or checking if a move is valid in this game or not.

You can look at what these methods are doing, what data they are working with (both parameters & instance variables), and what concepts they represent.

Then extract these responsibilities into new classes.

After the **refactoring**:

```
class GameBoard
  def display_board
  end
end

class GameState
  def save
  end
end

class GameLogic
  def make_move
  end

  def legal_move?
  end
end
```

When you do this your code will be closer to following the Single Responsibility Principle (SRP).

There are **other situations where you want to create classes**.

Let's say that you have a string that represents an email address.

This string is passed around to different methods & all of these methods check if this string is a valid email address.

Not only this is a case of duplication (which I cover in **principle #3**), but it's also **prone to errors**.

What if you forget to check in some new method?

The solution is to encapsulate that string into an `EmailAddress` class that checks its own validity when it's created. Now you can remove all these extra checks & be sure that you are working with a valid email address every time.

Look for these "hidden concepts with logic/validation" in your code.

There are a lot of these.

Another example could be `Integers` representing weights.

How do you make sure that you don't mix up one integer representing kilos with one representing pounds?

Create classes for them.

Principle 3 - Remove Unecessary Elements

The biggest barrier to **code readability & maintainability** are unnecessary elements.

What kind of elements are unnecessary?

- Return statements at the end of methods
- Extra temporary variables
- Redundant comments
- Duplication
- Code that doesn't do anything (like `1.times`)

> "Duplication is the primary enemy of a well-designed system." - Robert C. Martin

It's critical that you eliminate all these elements from your code if you want it to be **cleaner & easier to read**.

Example:

```ruby
def add(a, b)
  result = a + b

  return result
end
```

After:

```ruby
def add(a, b)
  a + b
end
```

Here we don't need the temporary variable or the `return` statement because in Ruby **the last expression in a method becomes its return value** automatically.

Another example:

```
@x = 1080 / 2
@y = 1080 / 2
```

After:

```
SCREEN_WIDTH = 1080

@x = SCREEN_WIDTH / 2
@y = SCREEN_WIDTH / 2
```

We have removed duplication (**principle #3**) & misunderstanding (**principle #1**).

These 3 principles work together.

When you know what to look for you'll see yourself applying these 3 principles almost effortlessly.

Taking These Principles to The Next Level

This is the "tip of the iceberg" as they say.

There is a lot more to learn to become an effective Ruby developer.

You need a catalog of **code examples & techniques** to apply these principles for maximum benefit in minimum time.

That's why I created a new course for you.

It's called **Beautiful Ruby**.

I'll let you know when it's available, hope you look forward to it :)

- Jesus Castello (www.rubyguides.com)