# Method Decomposition & Design

## Methods should be atomic

**Each method should do one thing.** A method should do a single, atomic thing (this is sometimes called the Single Responsibility Principle). This may be one line of code, or three, but rarely more than ten. **Methods should be short.** Let's take a look at an example of refactoring one long method into short, atomic methods, **NB** this is sometimes called "iterative stiffening". We'll use an implementation of the Towers of Hanoi exercise we worked on in the array section.

Here's the problem description in case you don't remember it:

Write a Towers of Hanoi game.

Keep three arrays, which represents the piles of discs. Pick a representation

of the discs to store in the arrays; maybe just a number representing their size.

In a loop, prompt the user (using gets) and ask what pile to select a disc

from, and where to put it.

After each move, check to see if they have succeeded in moving all the discs,

to the final pile. If so, they win!

Here's an example of a one-method solution:

def hanoi

disks = (1..3).to\_a.reverse

stacks = [disks, [], []]

until stacks[0].empty? && stacks[1..2].any?(&:empty?)

max\_height = stacks.map(&:count).max

render\_string = (max\_height - 1).downto(0).map do |height|

stacks.map do |stack|

stack[height] || " "

end.join("\t")

end.join("\n")

puts render\_string

move\_hash = { "a" => 0, "b" => 1, "c" => 2 }

while true

print "Move From: "

from\_stack\_num = move\_hash[gets.chomp]

break unless from\_stack\_num.nil?

puts "Invalid move!"

end

while true

print "Move To: "

to\_stack\_num = move\_hash[gets.chomp]

break unless to\_stack\_num.nil?

puts "Invalid move!"

end

from\_stack, to\_stack = stacks.values\_at(from\_stack\_num, to\_stack\_num)

raise "cannot move from empty stack" if from\_stack.empty?

unless (to\_stack.empty? || to\_stack.last > from\_stack.last)

raise "cannot move onto smaller disk"

end

to\_stack.push(from\_stack.pop)

end

puts "You did it!"

end

Let's start breaking this method into smaller methods. What are the steps that we take in this one fairly long method?

1. Set up a stack of disks
2. Set up a set of 3 stacks
3. Loop until its over
4. Display the stacks
5. Get move to position
6. Get move from position
7. Move the disk while checking to see if move is valid

Now that we've listed the atomic steps, it will be easy to split the method into smaller methods. Let's go step by step, and start by extracting the disks method:

def disks

(1..3).to\_a.reverse

end

def hanoi

stacks = [disks, [], []]

# ...

Notice that the disks method has one job of returning a stack of disks.

Lets extract stacks into a method that builds the stacks using disks.

def disks

(1..3).to\_a.reverse

end

def stacks

[disks, [], []]

end

def hanoi

until stacks[0].empty? && stacks[1..2].any?(&:empty?)

# ...

Notice that stacks has one job of building the stacks from the disks.

Now we're looping until the game is over, but it looks like this over condition is starting to get a bit complex. Lets break out over into its own method.

def over?

stacks[0].empty? && stacks[1..2].any?(&:empty?)

end

def hanoi

until over?

# ...

While we're taking turns moving disks we'll probably want to display the current state of the board each time. Extracting that block of code might look like this:

def display

max\_height = stacks.map(&:count).max

render\_string = (max\_height - 1).downto(0).map do |height|

stacks.map do |stack|

stack[height] || " "

end.join("\t")

end.join("\n")

end

def hanoi

until over?

puts display

# ...

The next step is to get the from and to stacks. This logic for getting a stack is mostly the same, less the prompt. We can write a method that takes a prompt as an argument.

def get\_stack(prompt)

move\_hash = { "a" => 0, "b" => 1, "c" => 2 }

while true

print prompt

stack\_num = move\_hash[gets.chomp]

return stack\_num unless stack\_num.nil?

puts "Invalid move!"

end

end

def hanoi

until over?

puts display

from\_stack\_num = get\_stack("Move from: ")

to\_stack\_num = get\_stack("Move to: ")

from\_stack, to\_stack = @stacks.values\_at(from\_stack\_num, to\_stack\_num)

raise "cannot move from empty stack" if from\_stack.empty?

unless (to\_stack.empty? || to\_stack.last > from\_stack.last)

raise "cannot move onto smaller disk"

end

to\_stack.push(from\_stack.pop)

# ...

Our methods are starting to look leaner :). The next step is to extract the work of moving the disk into its own move\_disk method.

def move\_disk(from\_stack, to\_stack)

from\_stack, to\_stack = @stacks.values\_at(from\_stack\_num, to\_stack\_num)

raise "cannot move from empty stack" if from\_stack.empty?

unless (to\_stack.empty? || to\_stack.last > from\_stack.last)

raise "cannot move onto smaller disk"

end

to\_stack.push(from\_stack.pop)

end

def hanoi

until over?

puts display

from\_stack = get\_stack("Move from: ")

to\_stack = get\_stack("Move to: ")

move\_disk(from\_stack, to\_stack)

end

end

**It reads like plain English.** Hiding away our implementation details in well-named helper methods both reduced the length of hanoi and made its structure more transparent. If somebody looks at this code, they will immediately understand what is going on, even without reading the definitions of get\_stack and move\_disk. This makes it a lot easier to understand code.

If they are interested in the implementation of a specific action, they know where to find it: localized to an atomic, helper method.

## Do not use global state

A good general goal is that your methods should be like a mathematical function: it should take something in and return something. It should not rely on anything besides what is explicitly passed in, and it should not have side-effects; it should communicate its result through the return value.

Some languages are stricter than Ruby: they don't allow you to use any data except what is passed in as an argument, and they don't let you change any outside ("global") variables variables, or communicate outside except through the return value. Ruby is more flexible, but the majority of methods should be written in this style.

Simply put: **the only way in should be the arguments, the only way out should be the return value**.

Here's an example of something super terrible:

# create a global i variable

$i = nil

def square\_then\_add\_two(num)

$i = num

square

$i = $i + 2

end

def square

# get global variable, square it, and reset

$i = $i \* $i

nil

end

First, square relies on a global variable. This should have been passed in directly. Everything that a method needs should be declared up-front as a parameter to the method. This makes it easier to know how to call the method: just pass in the listed arguments.

Second, square doesn't return anything useful; instead, it communicates by setting a global variable. That is unnecessarily convoluted; just give the answer back directly.

A general guideline, avoid global state. Don't use global variables to get around passing in arguments or return values. I basically never use $ variables.

## Don't modify arguments

Callers do not typically expect you to modify an argument. For instance:

def combine\_ingredients(alcohols, mixers)

drinks = []

alcohols.length.times do

drinks << [alcohols.pop, mixers.pop]

end

drinks

end

This destroys the caller's arrays. Did they expect this? If modification of the argument is essential to what you're doing, fine, but otherwise don't do something potentially unexpected and dangerous like this.

Instead do something like:

def combine\_ingredients(alcohols, mixers)

drinks = []

alcohols.each\_index do |i|

drinks << [alcohols[i], mixers[i]]

end

drinks

end