Test-Driven Development

WRITE BETTER CODE, FASTER.

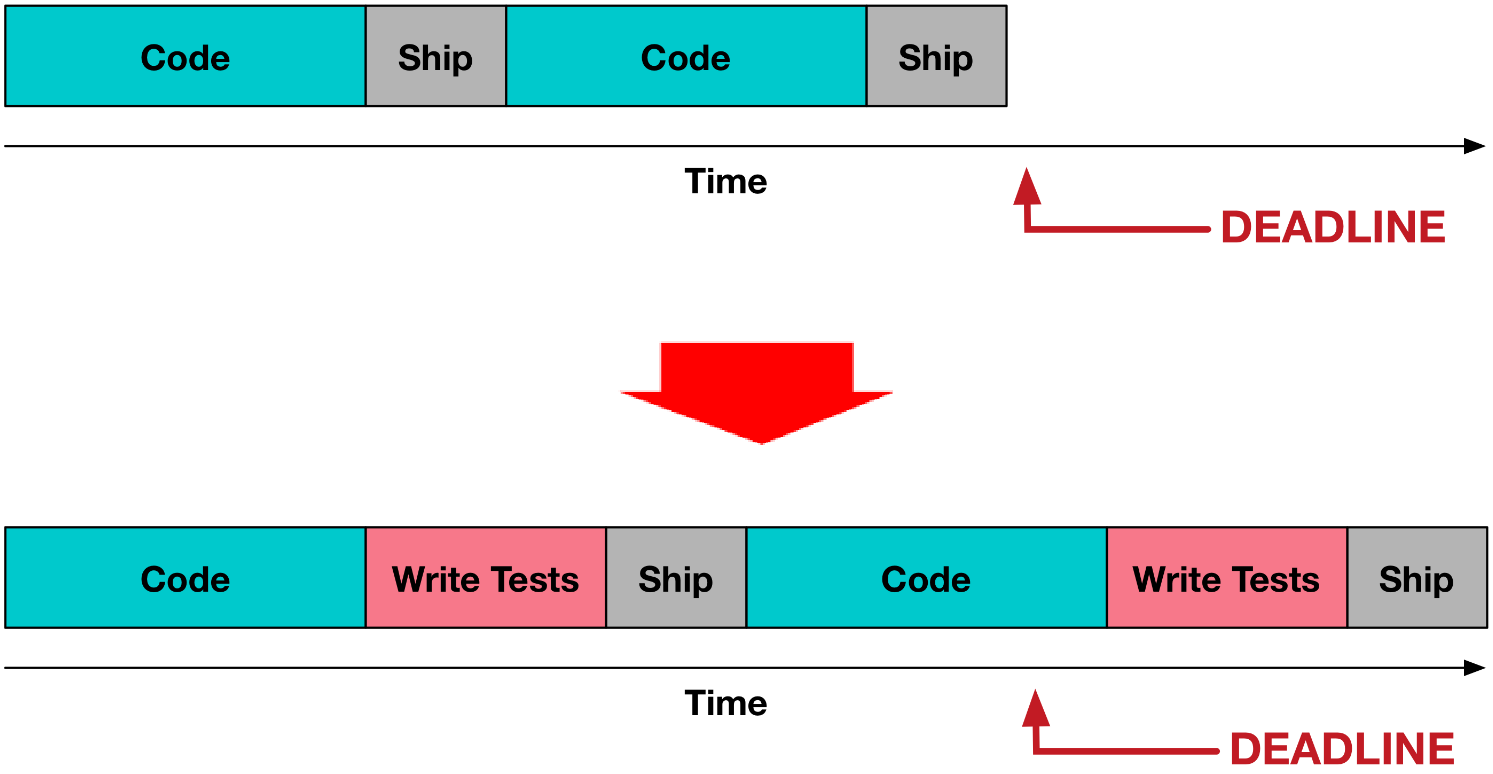
Evan Dorn - Honey

1. Introduction

"Hey, you should be writing tests!"

- Your Boss

Feels Like...



Code | Ship | Code | Ship

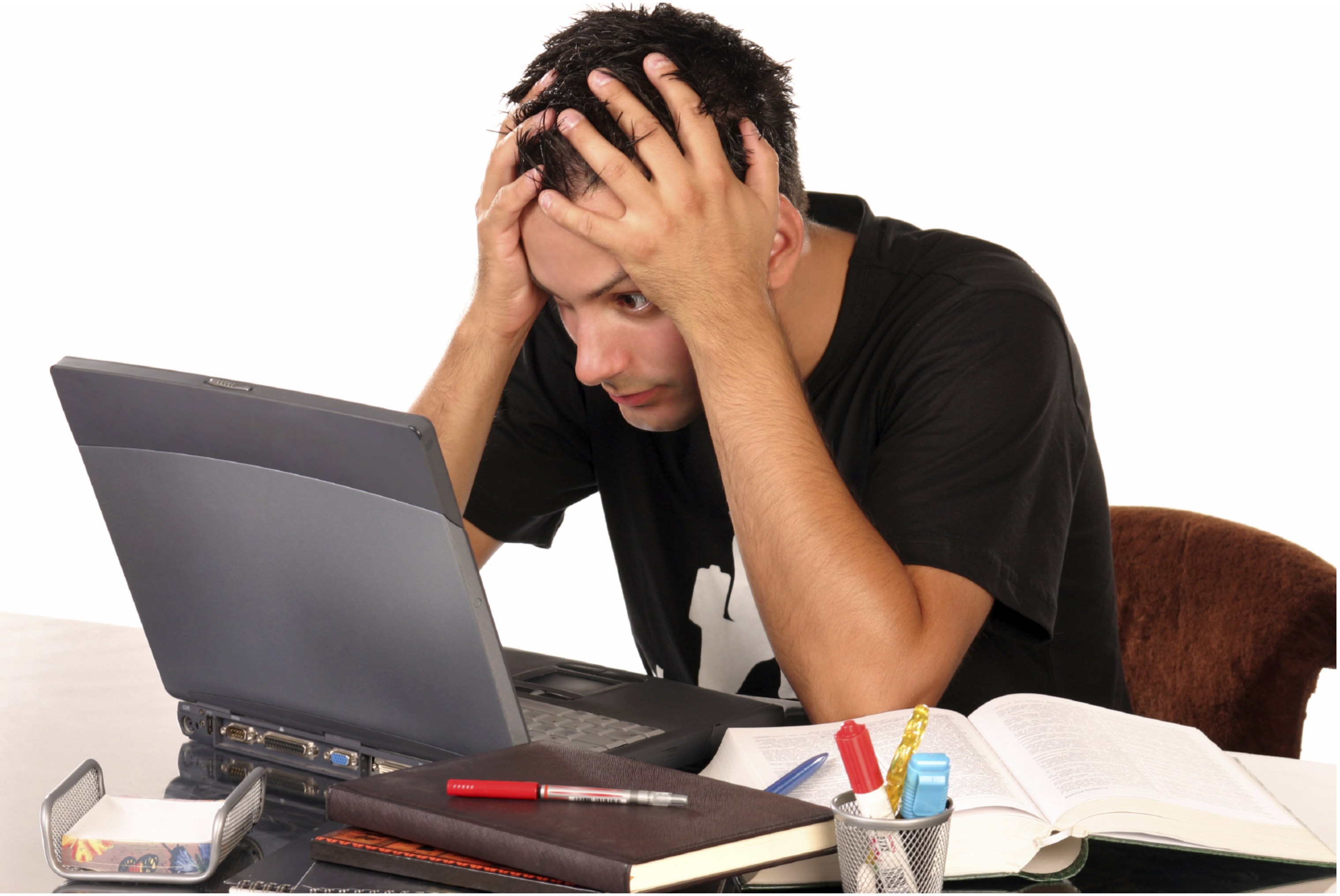
Time

DEADLINE

Code | Write Tests | Ship | Code | Write Tests | Ship

Time

DEADLINE



Fortunately

It's not like that at all

2.Planning

Programmers are Smart

But Intelligence

can be a

LIABILITY

Because Intelligence Enables

Progress

without

Process

The Professional Process:

1. Plan
2. Execute
3. Validate

"We know how skyscrapers work.

Screw blueprints, just hand us some bricks and we'll get started."

"Who has the time to organize tools beforehand and wash their hands?

I can help more patients if I just get cutting.

Engineers:

1. Blueprint
2. Construction
3. Inspection

Doctors

1. Surgical Plan
2. Surgery
3. Post-Op

Why does this process work?

Separate cognitive work into stages

To reduce errors and speed execution

1. Plan

What am I doing, and how will I do it?

… this is the hardest part!

2. Execute

This part is easy now!

Because you separated out some complexity

3. Validate

Check that execution
went to plan

So What About Programming?

Tests obviously provide

Validation

But what's the plan...?

3.Test Driven Development

TDD != "Writing Tests"

Test Driven Development

is an *engineering process*

TDD Defined:

- Describe correct behavior in a test
- Run the test, observe that it fails
- Write code
- Run the test, observe that it passes
- Refactor code
- Run the test, observe that it passes

TDD Described:

Red

Green

Refactor

It's a professional process!

Write test	Plan
Write code	Execute
Run Test	Validate

So process (incl. TDD)

Turns *Programming*

Into *Software Engineering*

4.Code Better

Starting to code before you plan

Is how spaghetti code happens

"Winging it"

Planning: Organizes your thoughts

Declares your intentions

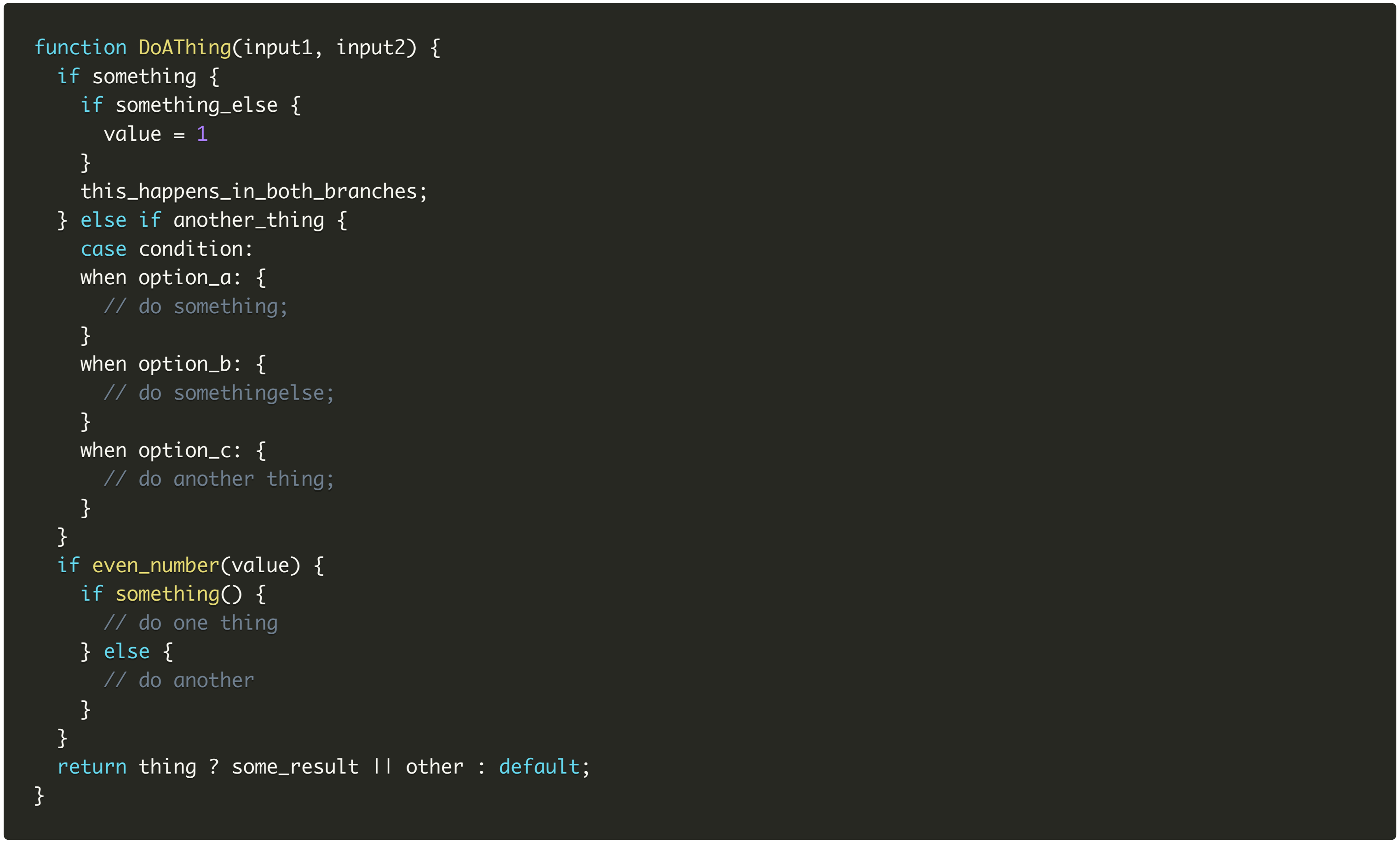
= Cleaner execution!

Testable code

Looks a lot like
"good code"

Clean Code:

Modularized
Decoupled
Short Functions w/
Limited Scope

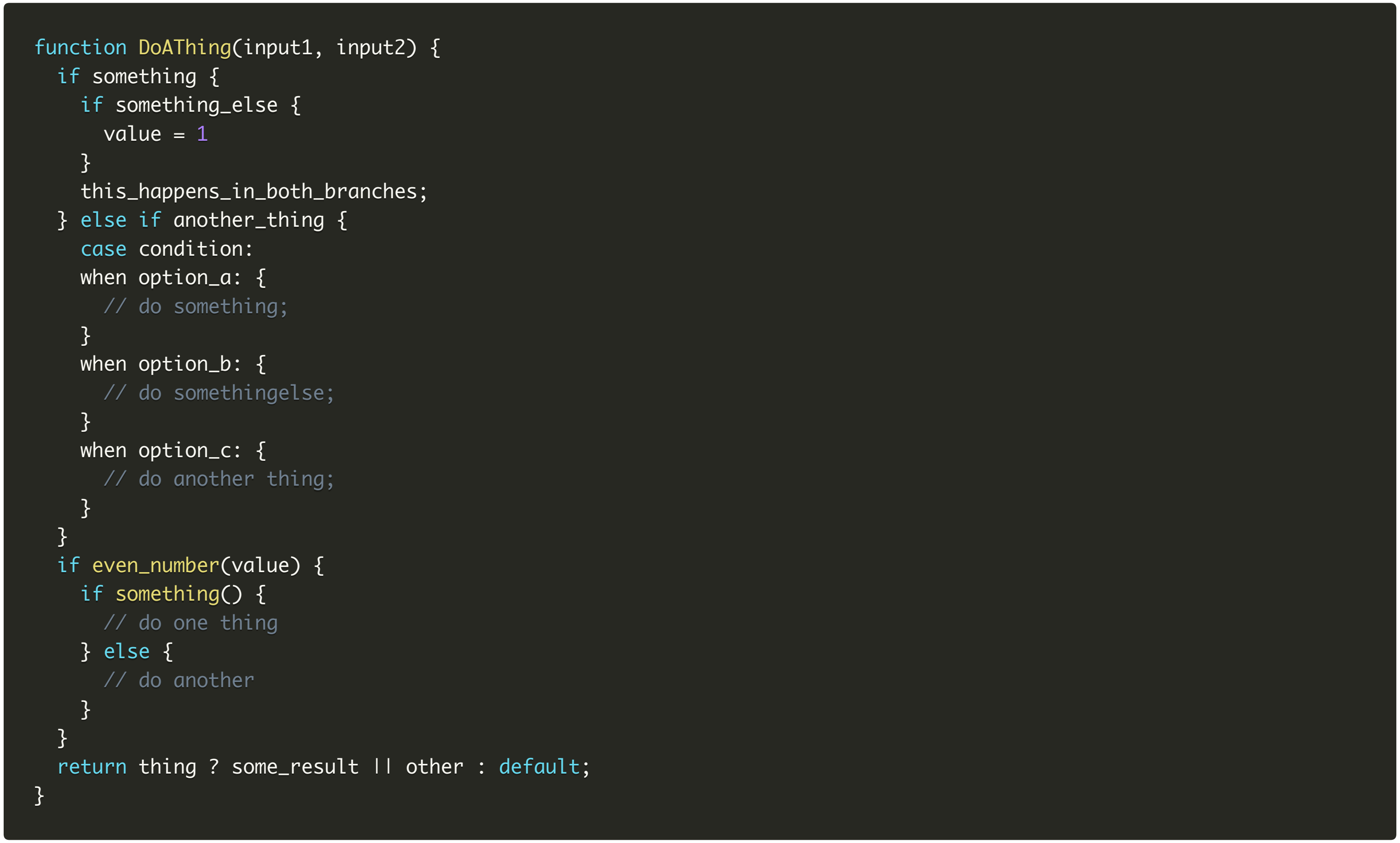


```
function DoAThing(input1, input2) {
  if something {
    if something_else {
      value = 1
    }
    this_happens_in_both_branches;
  } else if another_thing {
    case condition:
    when option_a: {
      // do something;
    }
    when option_b: {
      // do somethingelse;
    }
    when option_c: {
      // do another thing;
    }
  }
  if even_number(value) {
    if something() {
      // do one thing
    } else {
      // do another
    }
  }
  return thing ? some_result || other : default;
}
```

Cyclomatic Complexity

The number of code paths through a function

The *minimum* # of examples to prove correctness



```
function DoAThing(input1, input2) {
  if something {
    if something_else {
      value = 1
    }
    this_happens_in_both_branches;
  } else if another_thing {
    case condition:
    when option_a: {
      // do something;
    }
    when option_b: {
      // do somethingelse;
    }
    when option_c: {
      // do another thing;
    }
  }
  if even_number(value) {
    if something() {
      // do one thing
    } else {
      // do another
    }
  }
  return thing ? some_result || other : default;
}
```

Writing Tests First

Implicitly results in small, well-defined functions

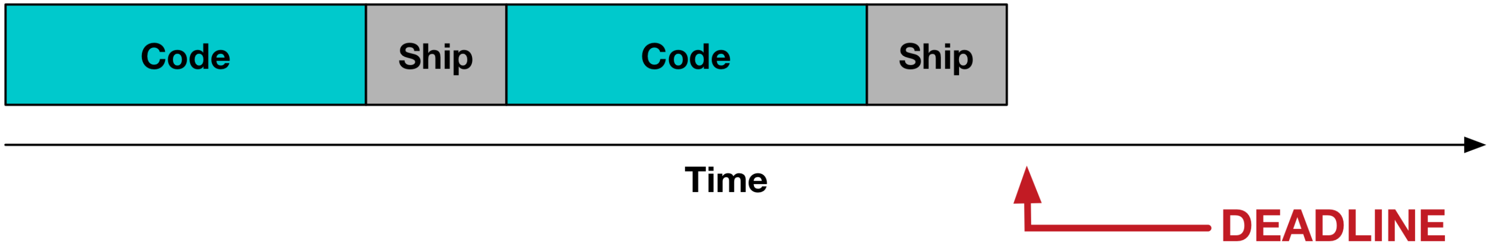
A test *describes* your code

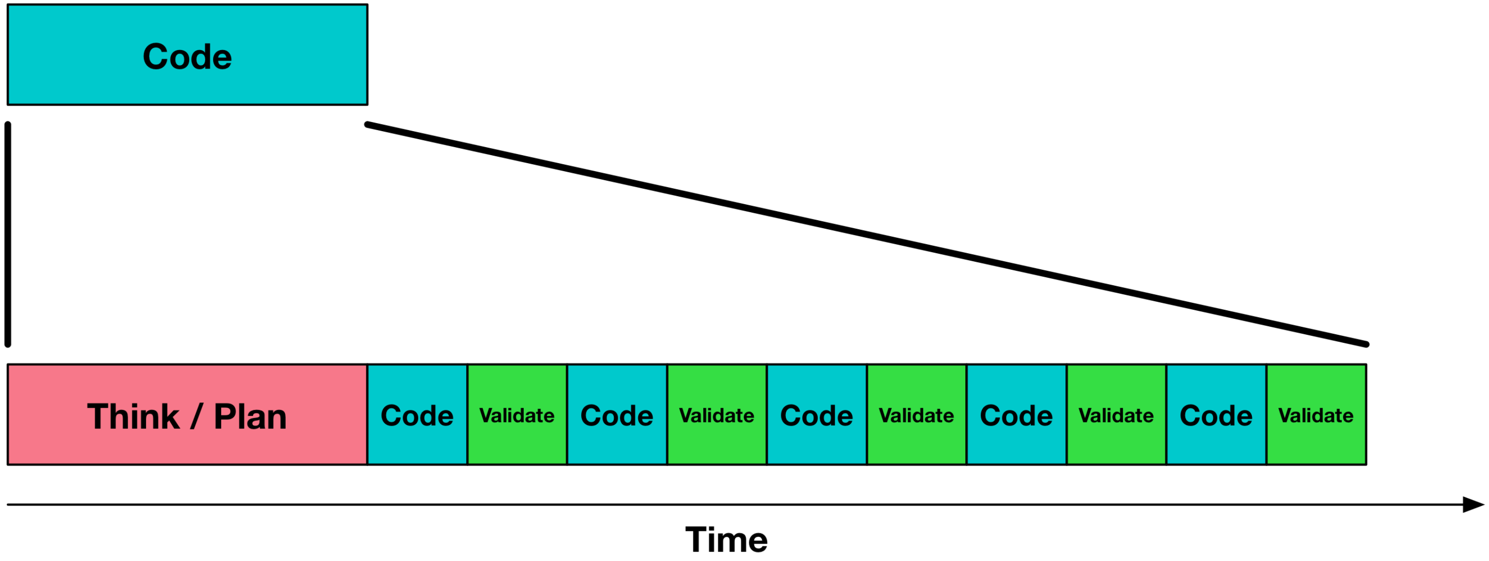
Therefore your code will be *describable*

5.Code Faster

Okay, so TDD's cleaner.

But is it really *faster?*


Code
Ship
Code
Ship
Time
DEADLINE


Code
Think / Plan
Code
Validate
Code
Validate
Code
Validate
Code
Validate
Code
Validate
Time

You're *always* validating

Reloading the browser

Opening console

Reading print outputs

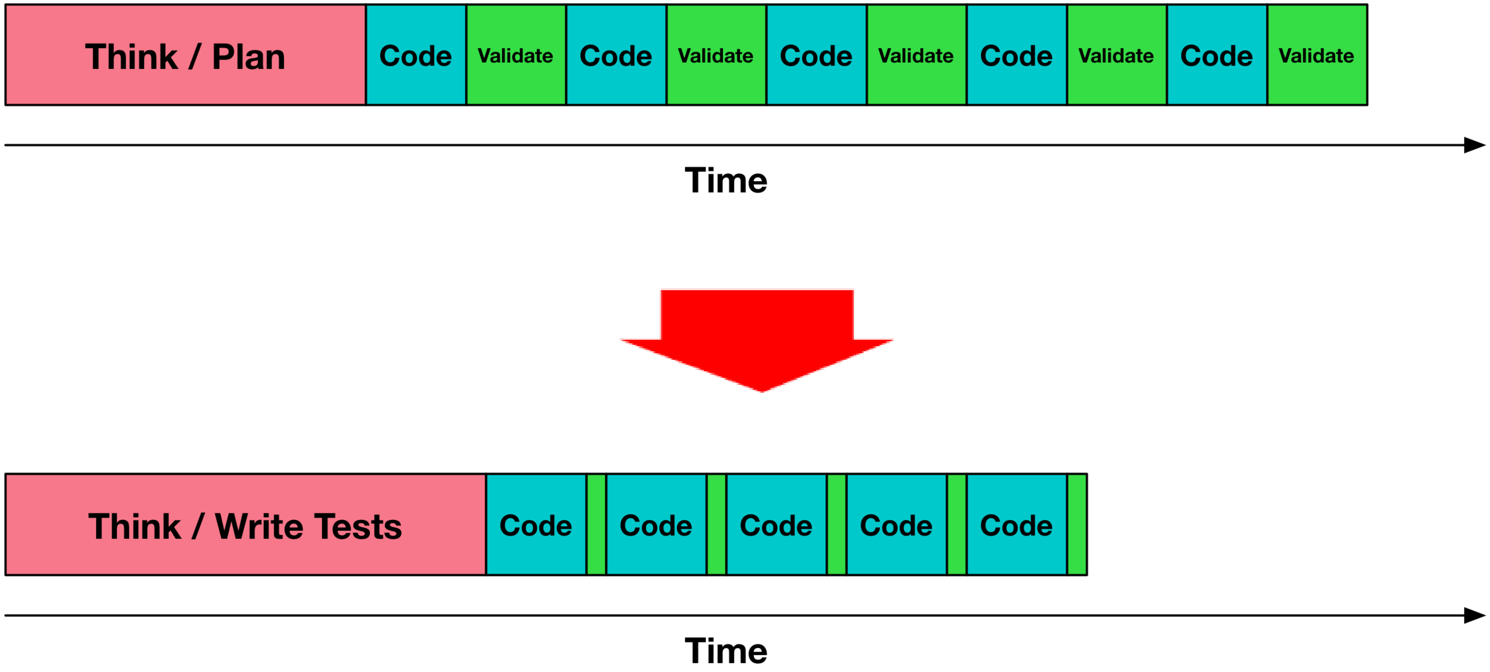
Tests are faster

And more repeatable
Than hand validation

Every time you switch windows

Your brain needs 15 seconds

To rebuild visual context



Think / Plan

Code

Validate

Code

Validate

Code

Validate

Code

Validate

Code

Validate

Time

Think / Write Tests

Code

Code

Code

Code

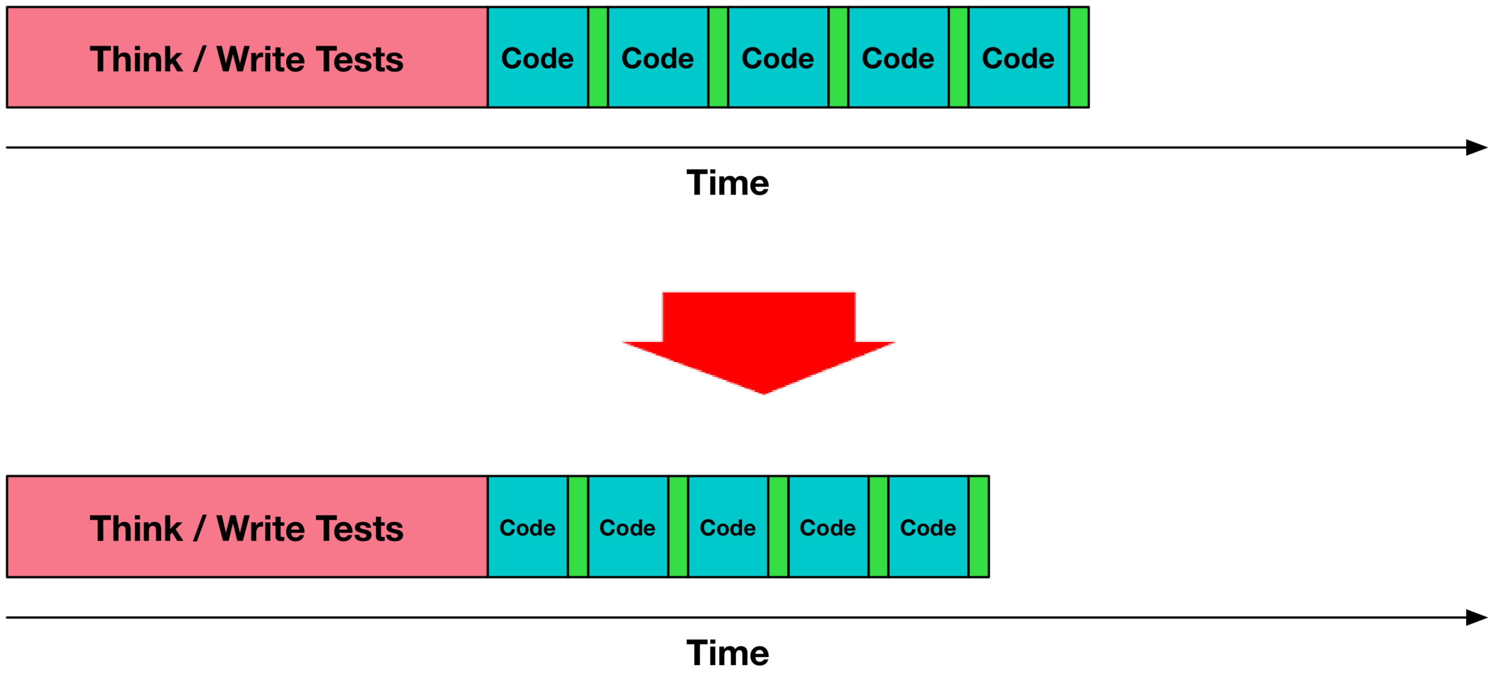
Code

Time

Once you've thought through a plan

And declared the behavior

The implementation is often obvious



Think / Write Tests

Code

Code

Code

Code

Code

Time

Think / Write Tests

Code

Code

Code

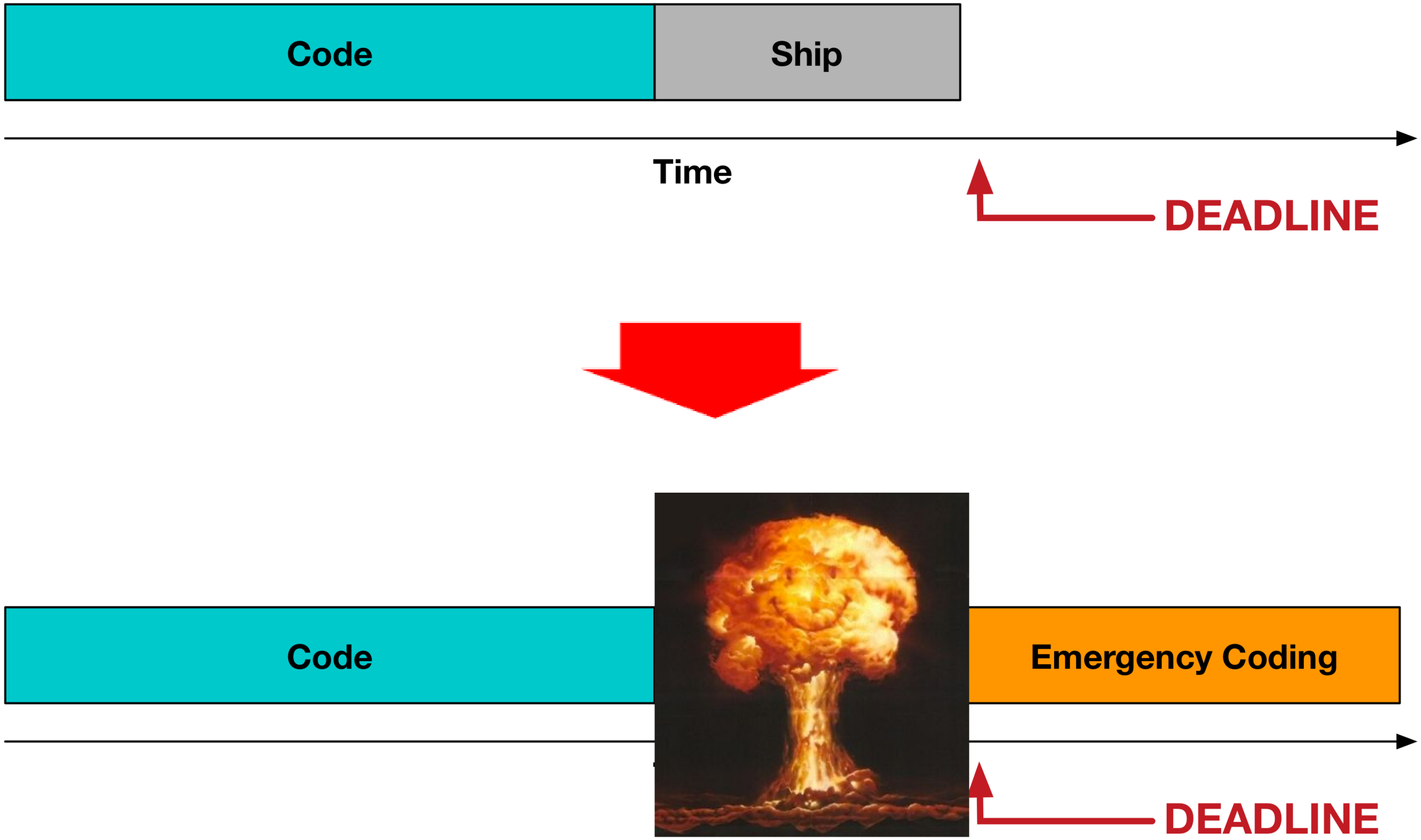
Code

Code

Time

Let's Zoom Back Out

And look at the bigger picture



Code

Ship

Time

DEADLINE

Code

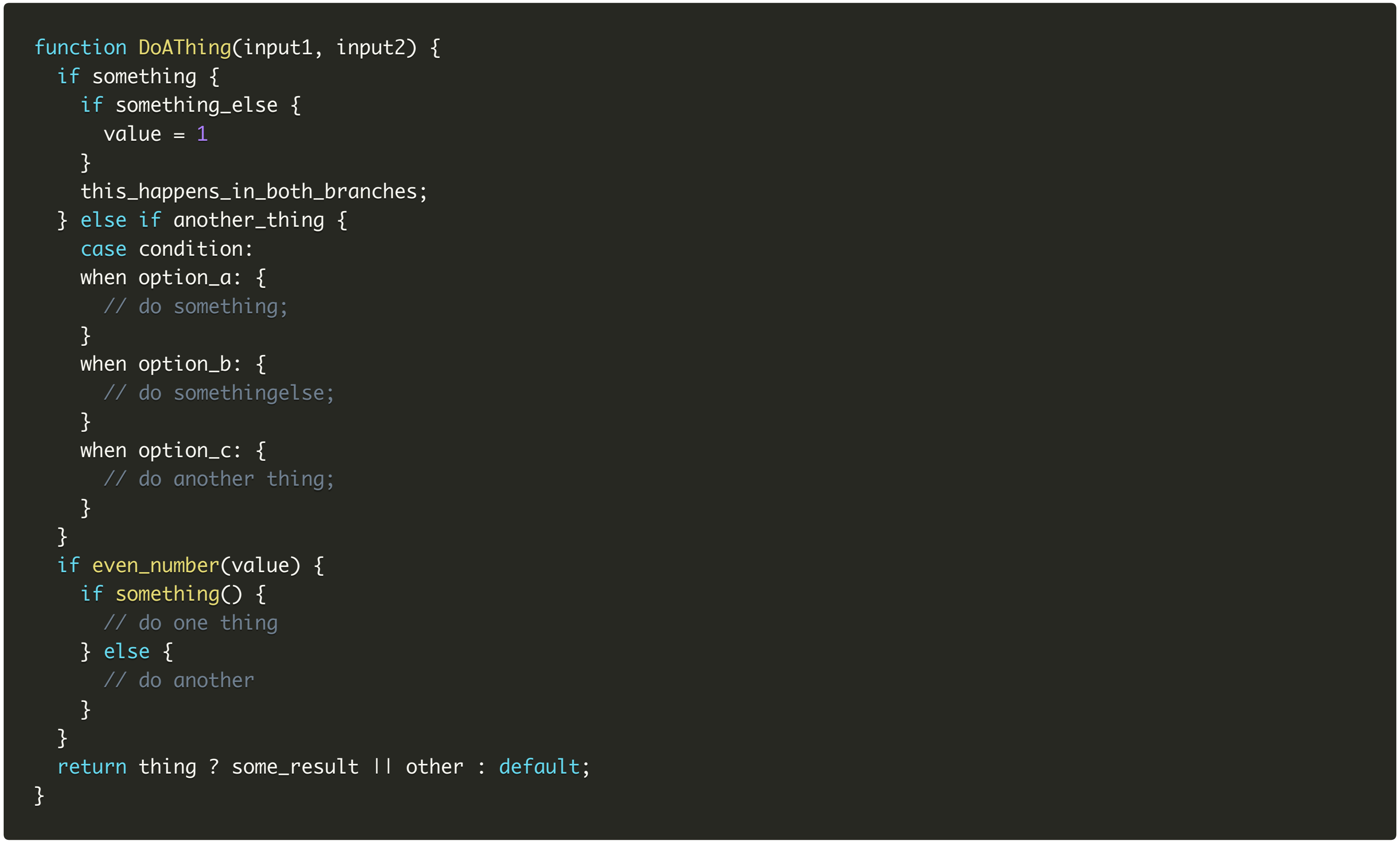
Emergency Coding

DEADLINE

If your code is Complex...

And you're validating by hand...

...You're gonna miss something.



```
function DoAThing(input1, input2) {
  if something {
    if something_else {
      value = 1
    }
    this_happens_in_both_branches;
  } else if another_thing {
    case condition:
    when option_a: {
      // do something;
    }
    when option_b: {
      // do somethingelse;
    }
    when option_c: {
      // do another thing;
    }
  }
  if even_number(value) {
    if something() {
      // do one thing
    } else {
      // do another
    }
  }
  return thing ? some_result || other : default;
}
```

Speed Benefits:

- Faster Validation Step
- Faster Coding Step
- Fewer Disasters

6.Caveats

First Caveat:

You won't be faster right away
Mastering TDD takes practice

Second Caveat:

It doesn't work for all cases

Some things are too hard to test

Sometimes you don't have a plan yet

Exploratory Coding

When the "How" isn't clear

Go ahead and fiddle around

But then test and refactor

Sometimes you can test at a broader level

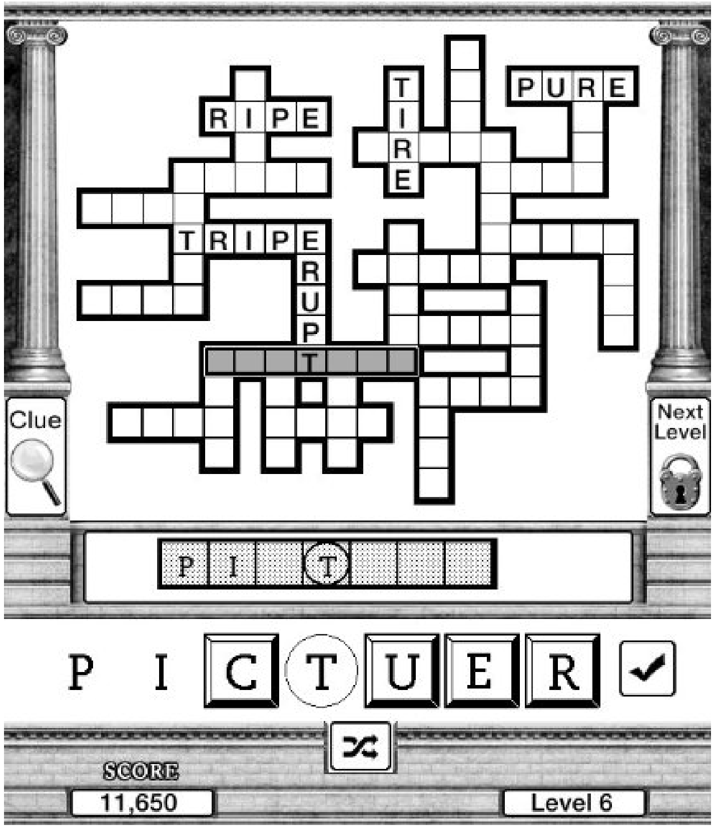
If the implementation is unclear, try an integration test

7. Demo

Goal: an app to help Roxane cheat at word games

Write it on an airplane in 30 minutes

This is so easy I don't need tests!



SCORE 11,650

Level 6

pattern: ____E_

letters: onyh

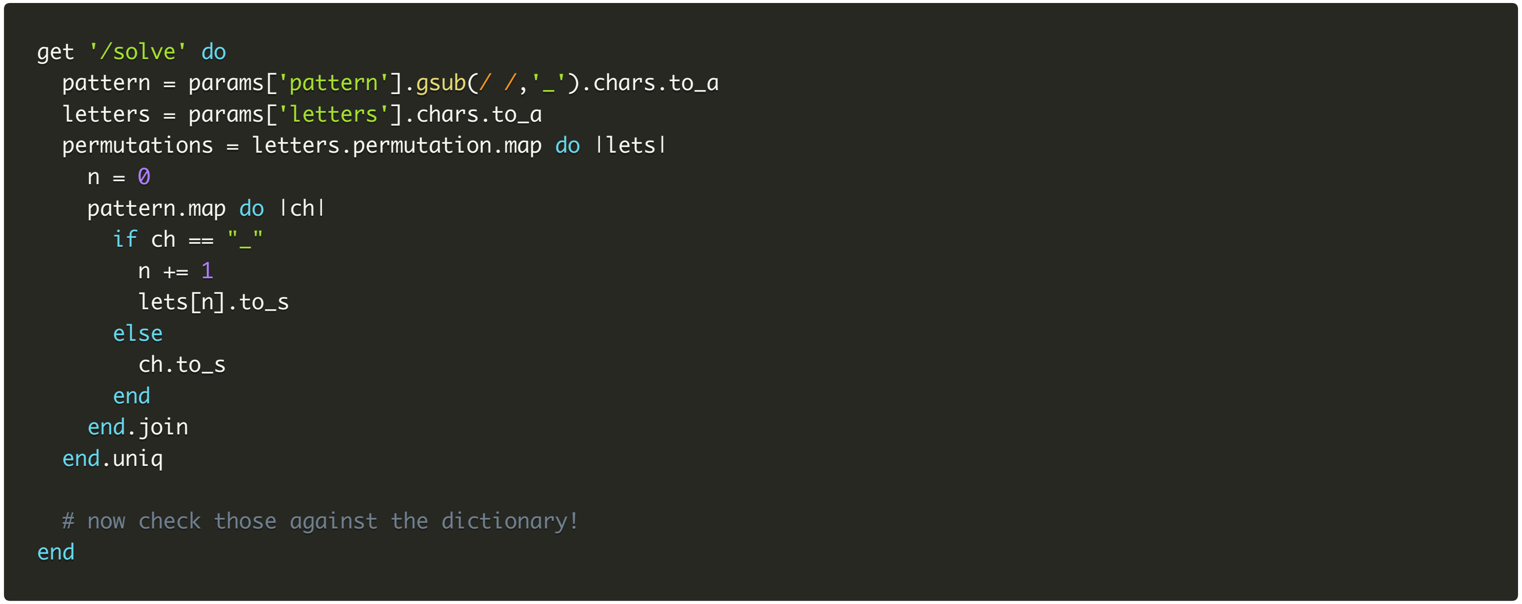
hnoEy

hnyEo

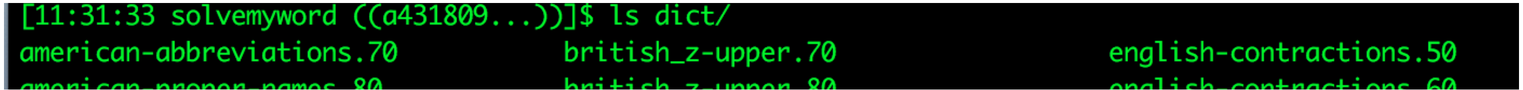
honEy

hynEo etc...

Then check those against a dictionary

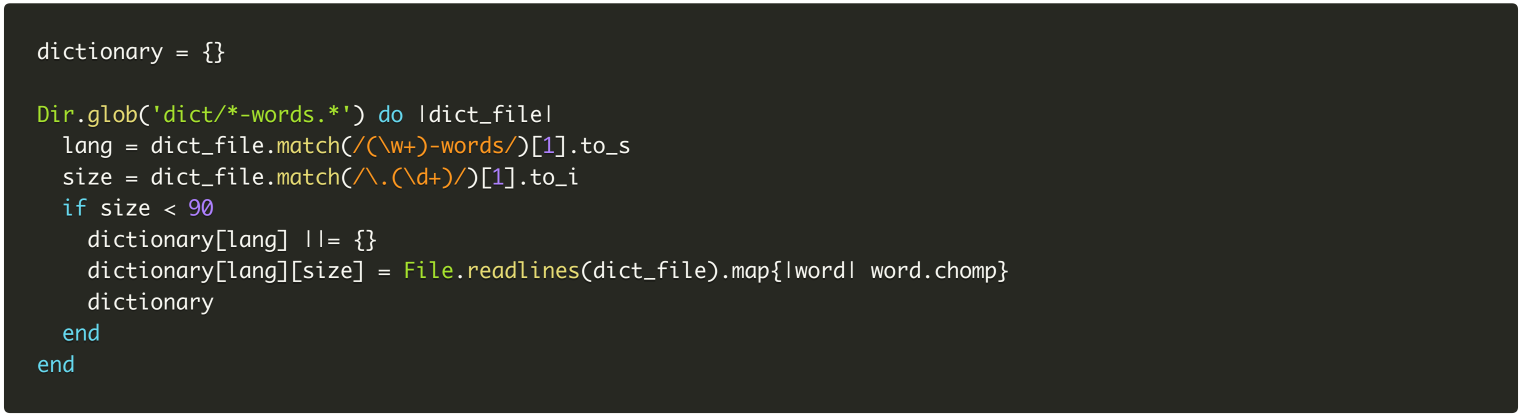


```ruby
get '/solve' do
  pattern = params['pattern'].gsub(/ /,'_').chars.to_a
  letters = params['letters'].chars.to_a
  permutations = letters.permutation.map do |lets|
    n = 0
    pattern.map do |ch|
      if ch == "_"
        n += 1
        lets[n].to_s
      else
        ch.to_s
      end
    end.join
  end.uniq

  # now check those against the dictionary!
end
```



```
[11:31:33 solvemyword ((a431809...))]$ ls dict/
american-abbreviations.70          british_z-upper.70                 english-contractions.50
american-proper-names.80           british_z-upper.80                 english-contractions.60
```

```
{
  english: {
   10: [ list, of, words ],
   20: [ list, of, words ],
   30: [ list, of, words ]
   # etc....
  },
  british: {
   10: [ list, of, words ],
   20: [ list, of, words ],
   # etc....
  },
  american: {
   10: [ list, of, words ],
   20: [ list, of, words ],
   # etc....
  }
}
```



```ruby
dictionary = {}

Dir.glob('dict/*-words.*') do |dict_file|
  lang = dict_file.match(/(\w+)-words/)[1].to_s
  size = dict_file.match(/\.(\d+)/)[1].to_i
  if size < 90
    dictionary[lang] ||= {}
    dictionary[lang][size] = File.readlines(dict_file).map{|word| word.chomp}
    dictionary
  end
end
```

Result: DISASTER

I wasted over an hour

Trying to retrieve dictionary words

Every. Single. Return. Was. Nil.

It doesn't seem that hard

But I was loading a nested data structure

With regular expressions

Then reading it in the same function

And the only way I had to test it

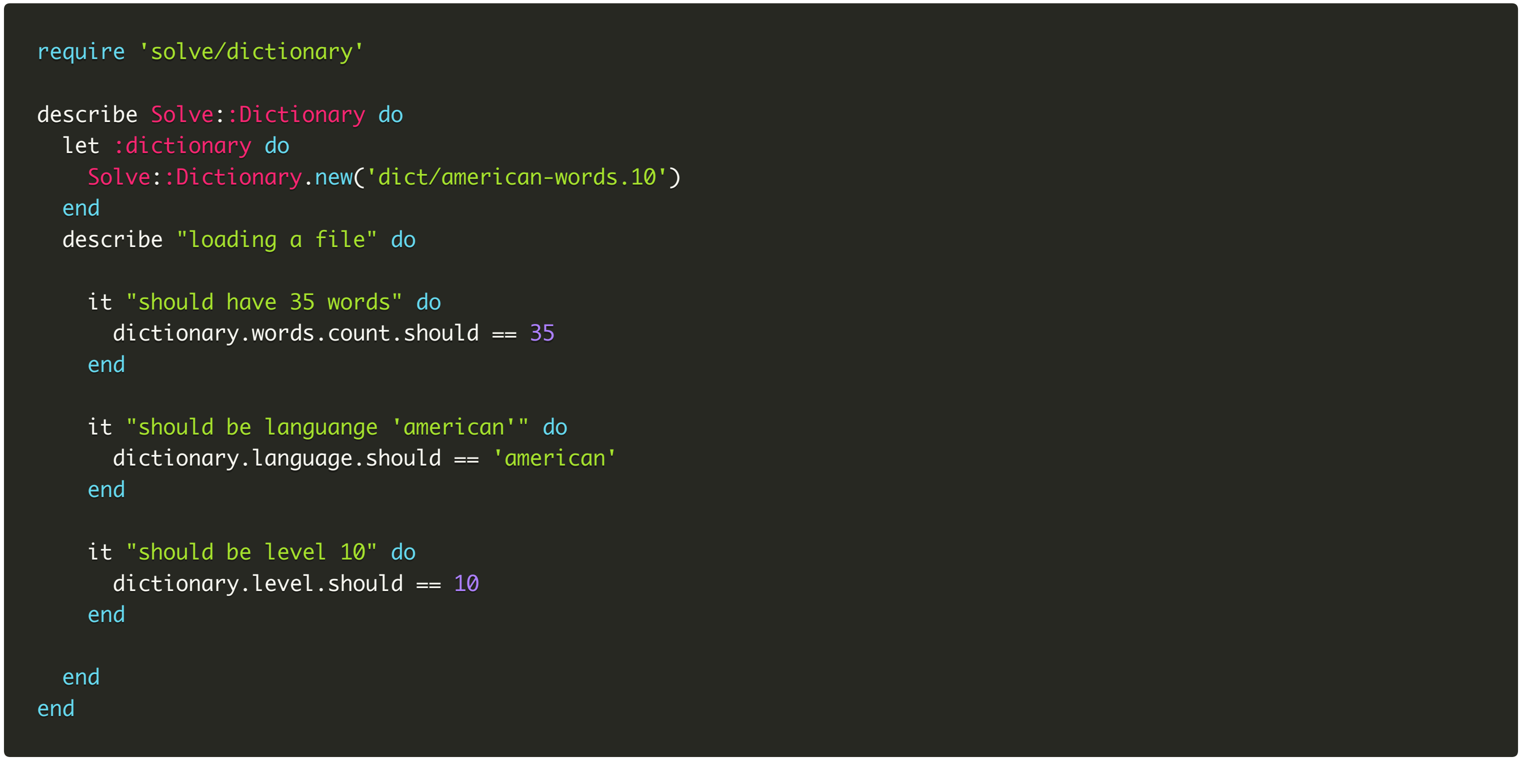
Was to reload the page
Enter in patterns and letters
And resubmit the form

Okay.

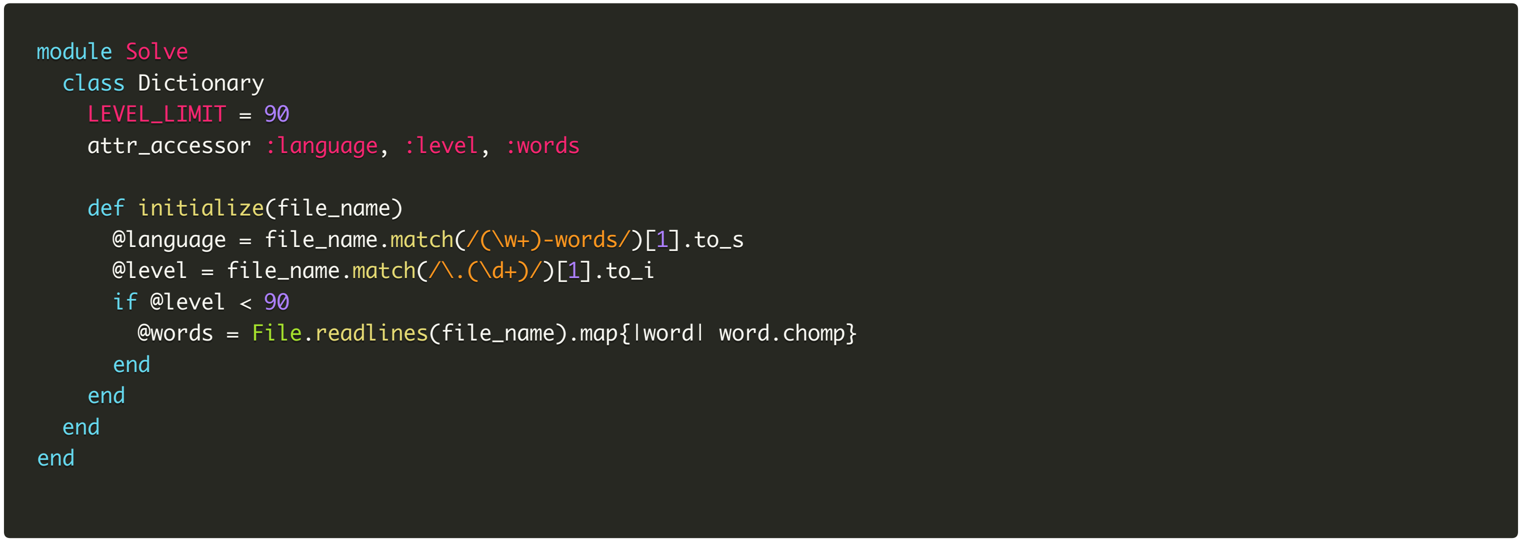
Let's structure the code and write tests.

The Result:

I Finished the app in 35 minutes.



```ruby
require 'solve/dictionary'

describe Solve::Dictionary do
  let :dictionary do
    Solve::Dictionary.new('dict/american-words.10')
  end
  describe "loading a file" do

    it "should have 35 words" do
      dictionary.words.count.should == 35
    end

    it "should be languange 'american'" do
      dictionary.language.should == 'american'
    end

    it "should be level 10" do
      dictionary.level.should == 10
    end

  end
end
```



```ruby
module Solve
  class Dictionary
    LEVEL_LIMIT = 90
    attr_accessor :language, :level, :words

    def initialize(file_name)
      @language = file_name.match(/(\w+)-words/)[1].to_s
      @level = file_name.match(/\.(\d+)/)[1].to_i
      if @level < 90
        @words = File.readlines(file_name).map{|word| word.chomp}
      end
    end
  end
end
```

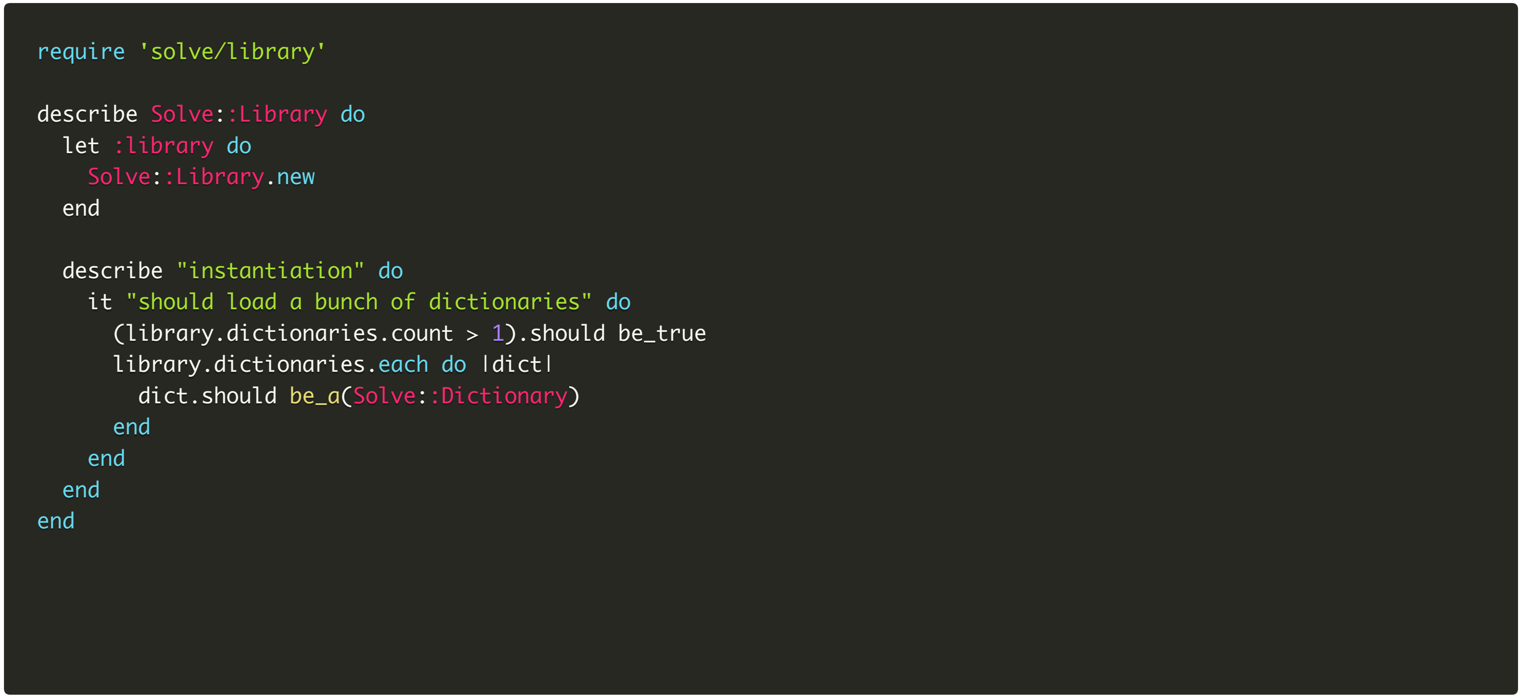
Once I had the test written & running

I never had to load the page to test it
And could stay focused on the code file

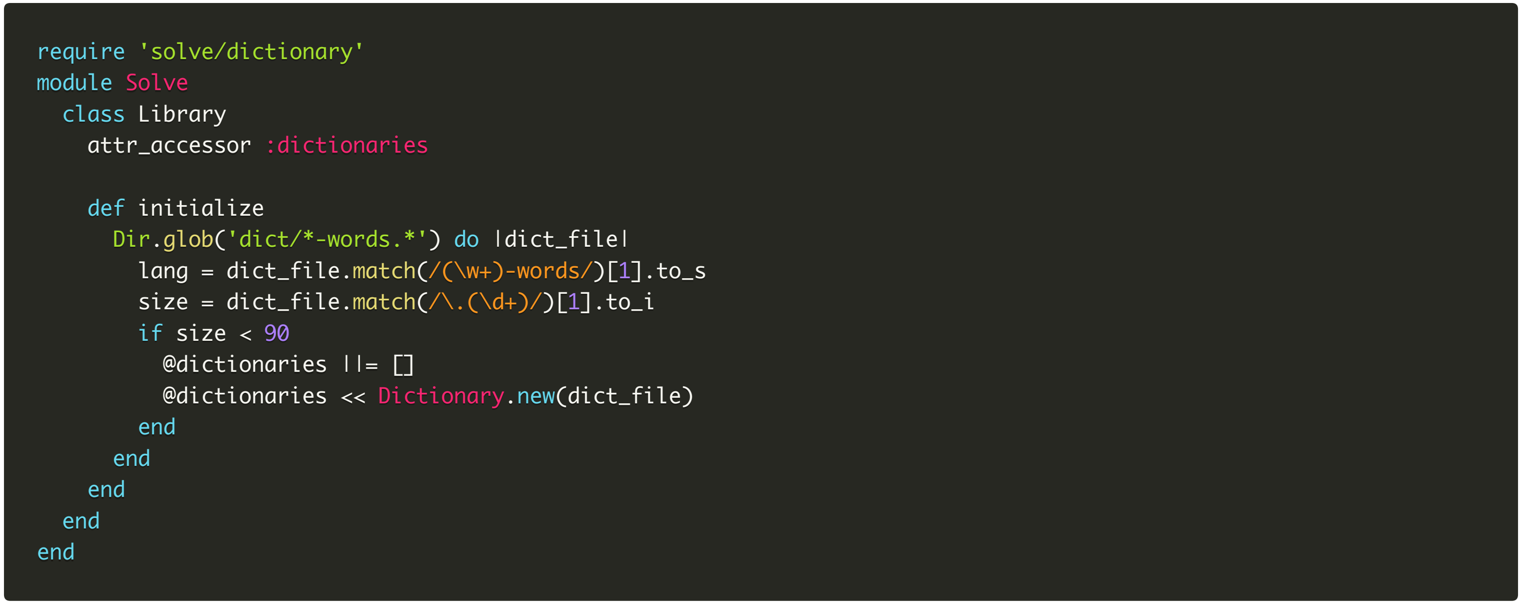
The Result:

Class written in ~90 seconds

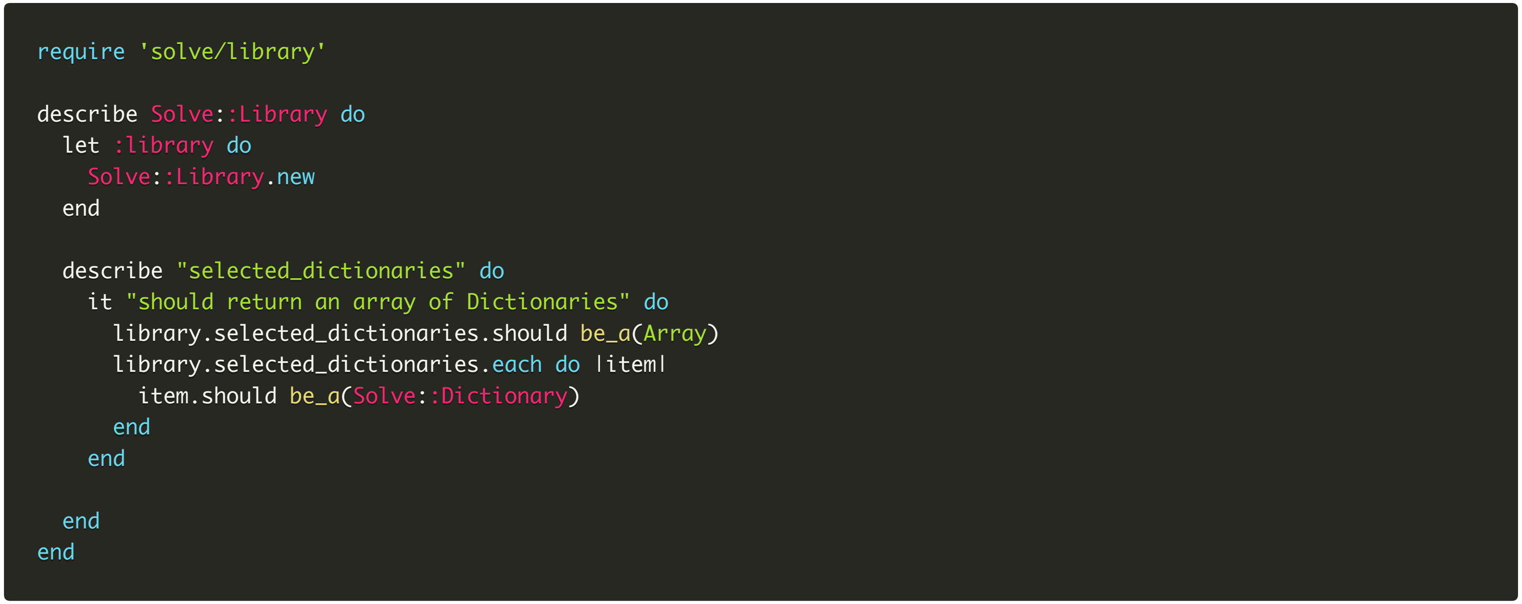
And the tests proved correctness



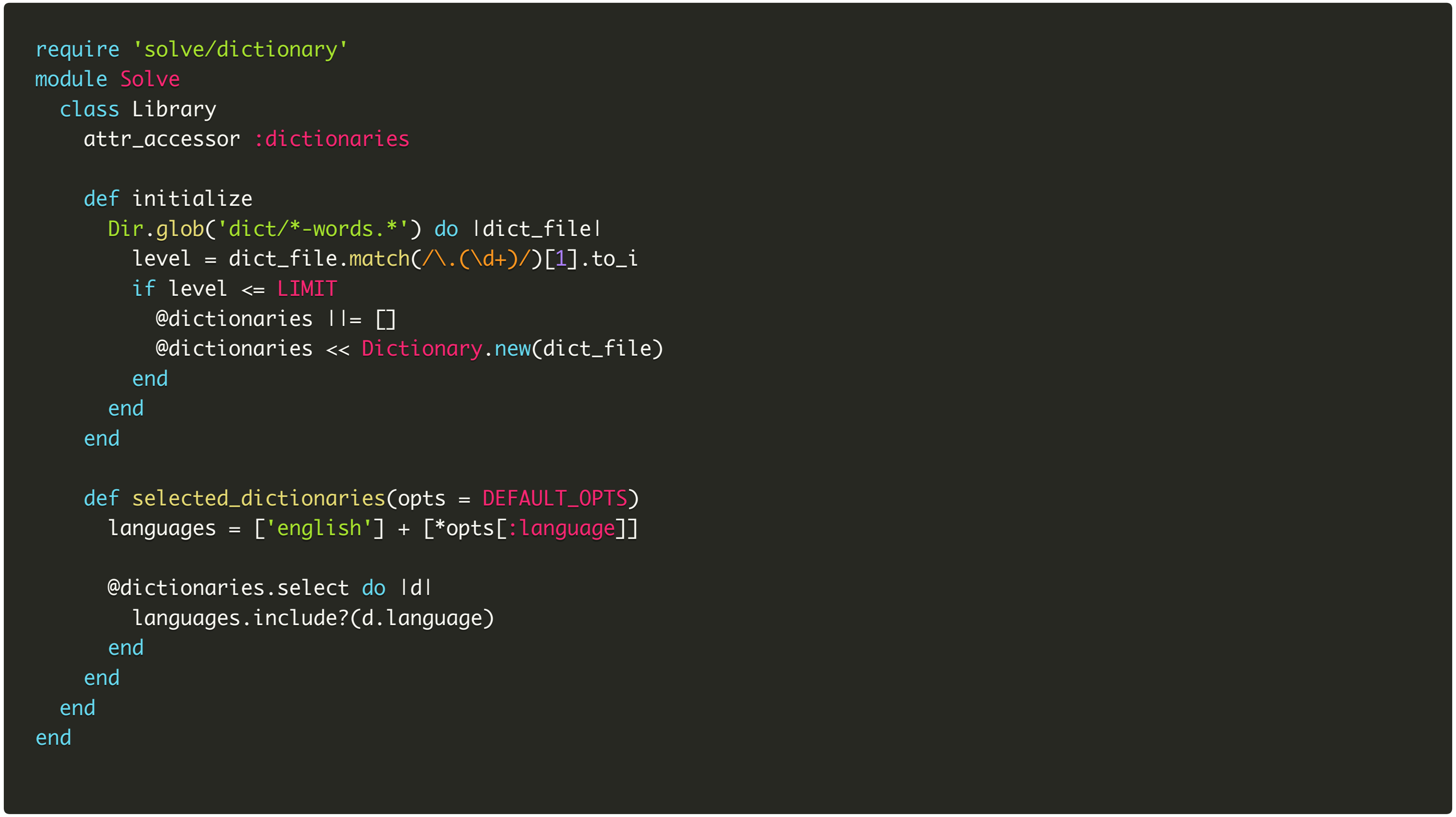
```ruby
require 'solve/library'

describe Solve::Library do
  let :library do
    Solve::Library.new
  end

  describe "instantiation" do
    it "should load a bunch of dictionaries" do
      (library.dictionaries.count > 1).should be_true
      library.dictionaries.each do |dict|
        dict.should be_a(Solve::Dictionary)
      end
    end
  end
end
```



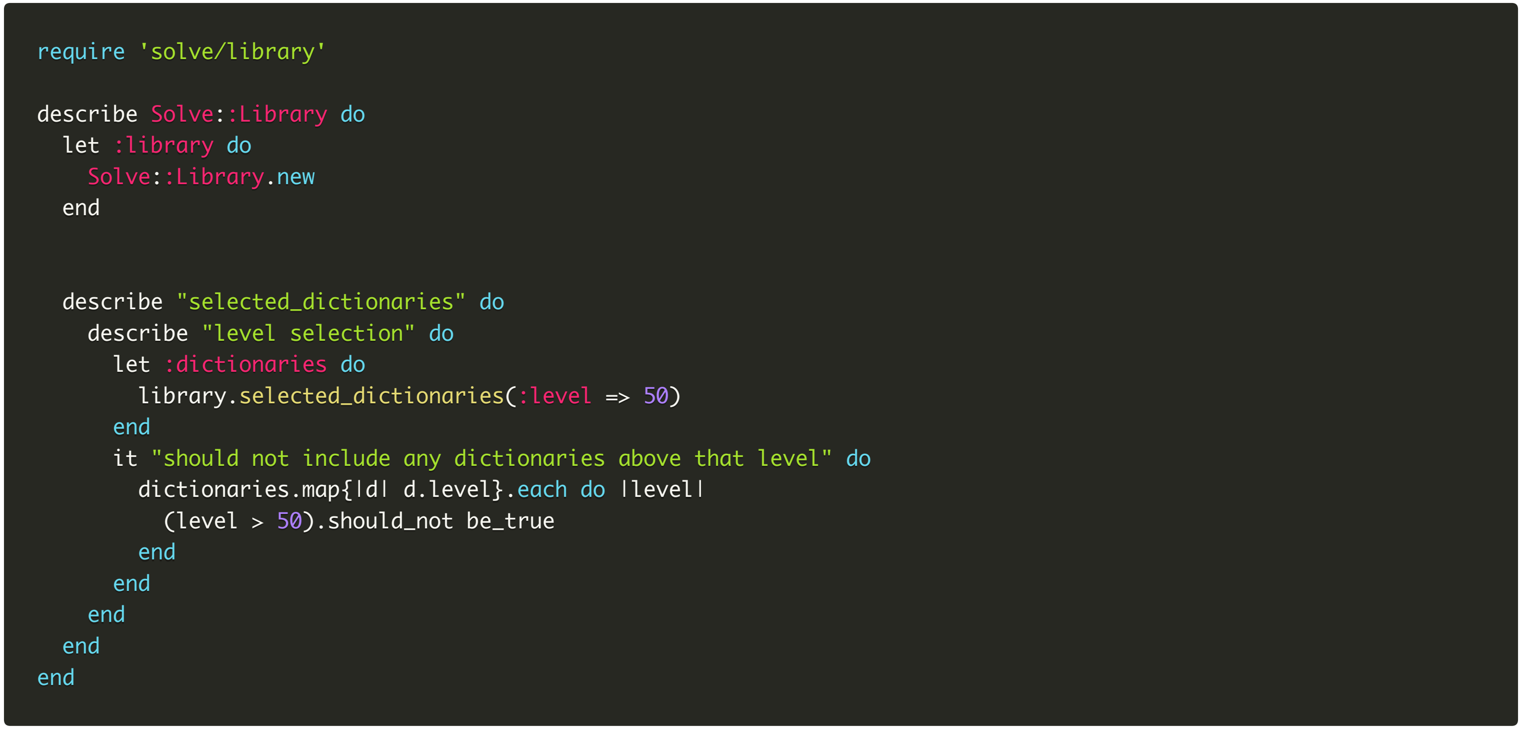
```ruby
require 'solve/dictionary'
module Solve
  class Library
    attr_accessor :dictionaries

    def initialize
      Dir.glob('dict/*-words.*') do |dict_file|
        lang = dict_file.match(/(\w+)-words/)[1].to_s
        size = dict_file.match(/\.(\d+)/)[1].to_i
        if size < 90
          @dictionaries ||= []
          @dictionaries << Dictionary.new(dict_file)
        end
      end
    end
  end
end
```



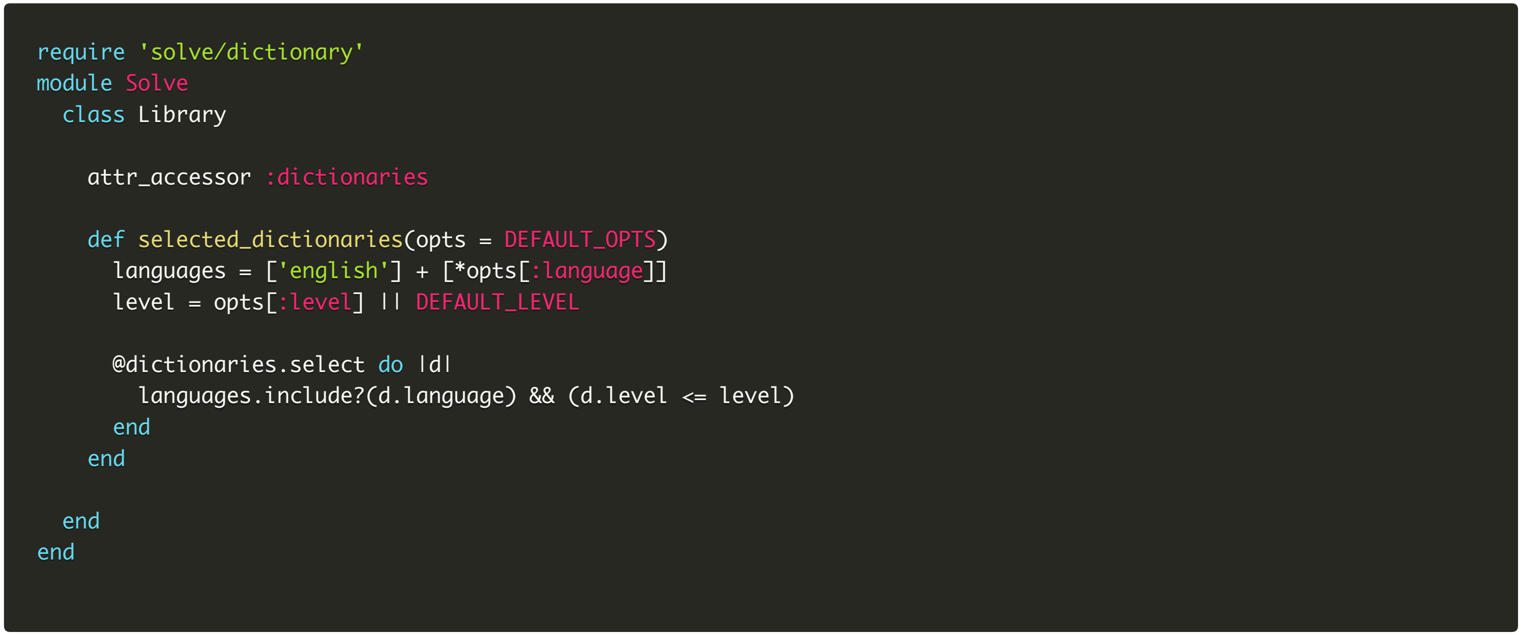
```ruby
require 'solve/library'

describe Solve::Library do
  let :library do
    Solve::Library.new
  end

  describe "selected_dictionaries" do
    it "should return an array of Dictionaries" do
      library.selected_dictionaries.should be_a(Array)
      library.selected_dictionaries.each do |item|
        item.should be_a(Solve::Dictionary)
      end
    end

  end
end
```



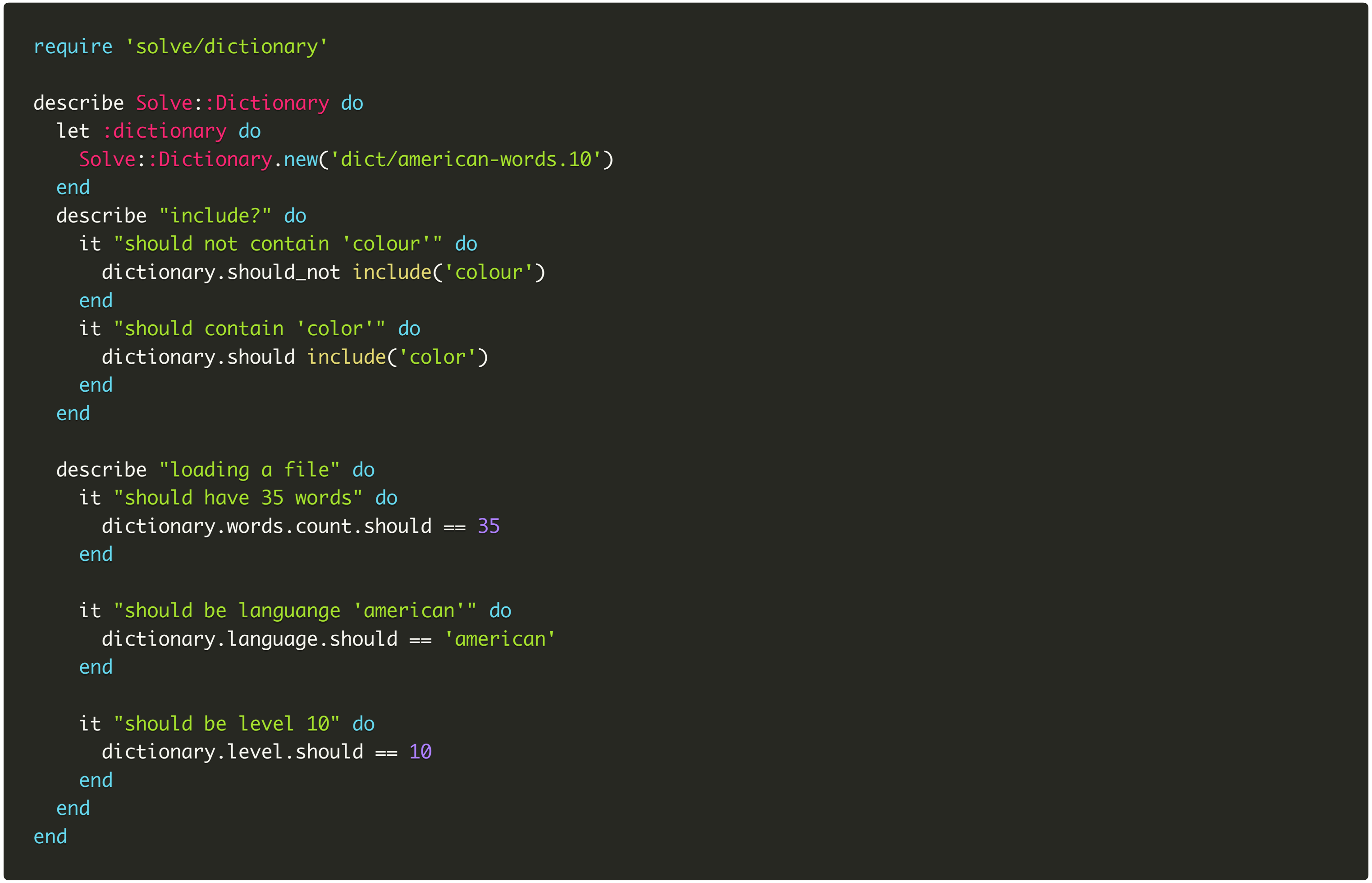
```ruby
require 'solve/dictionary'
module Solve
  class Library
    attr_accessor :dictionaries

    def initialize
      Dir.glob('dict/*-words.*') do |dict_file|
        level = dict_file.match(/\.(\d+)/)[1].to_i
        if level <= LIMIT
          @dictionaries ||= []
          @dictionaries << Dictionary.new(dict_file)
        end
      end
    end

    def selected_dictionaries(opts = DEFAULT_OPTS)
      languages = ['english'] + [*opts[:language]]

      @dictionaries.select do |d|
        languages.include?(d.language)
      end
    end
  end
end
```



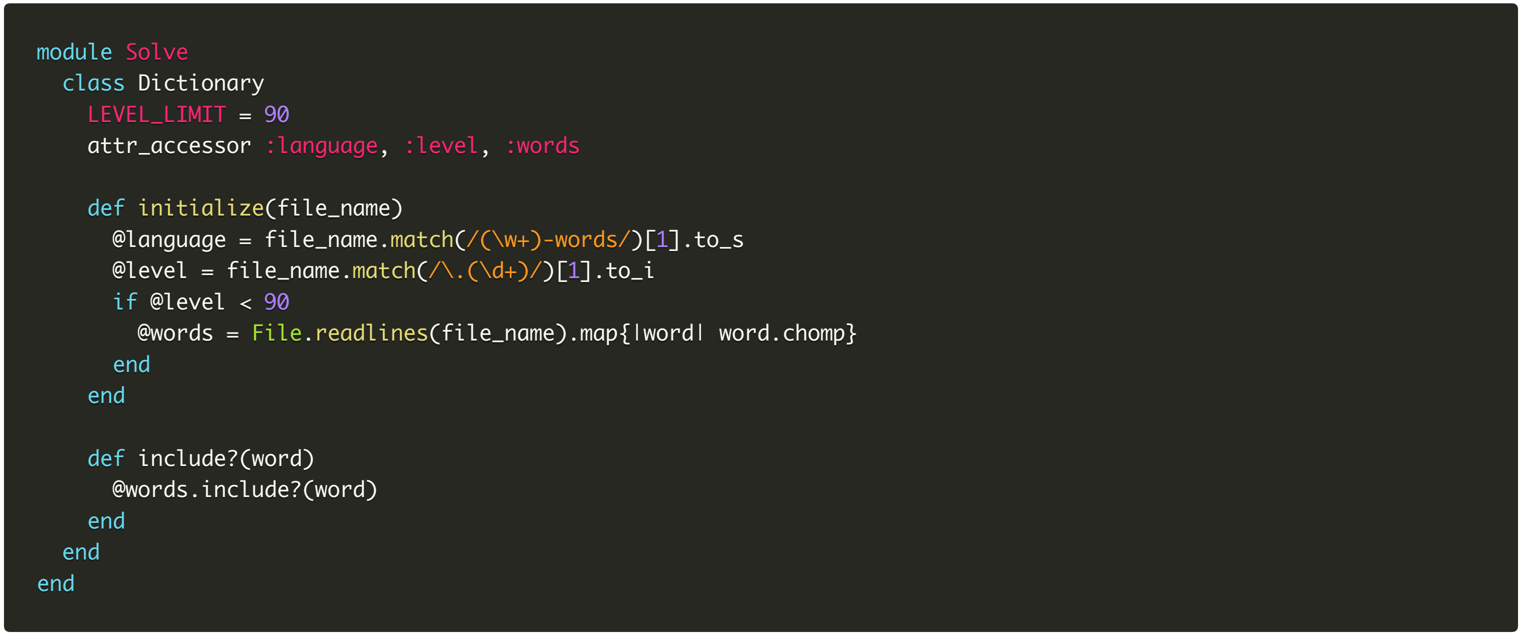
```
require 'solve/library'

describe Solve::Library do
  let :library do
    Solve::Library.new
  end


  describe "selected_dictionaries" do
    describe "level selection" do
      let :dictionaries do
        library.selected_dictionaries(:level => 50)
      end
      it "should not include any dictionaries above that level" do
        dictionaries.map{|d| d.level}.each do |level|
          (level > 50).should_not be_true
        end
      end
    end
  end
end
```



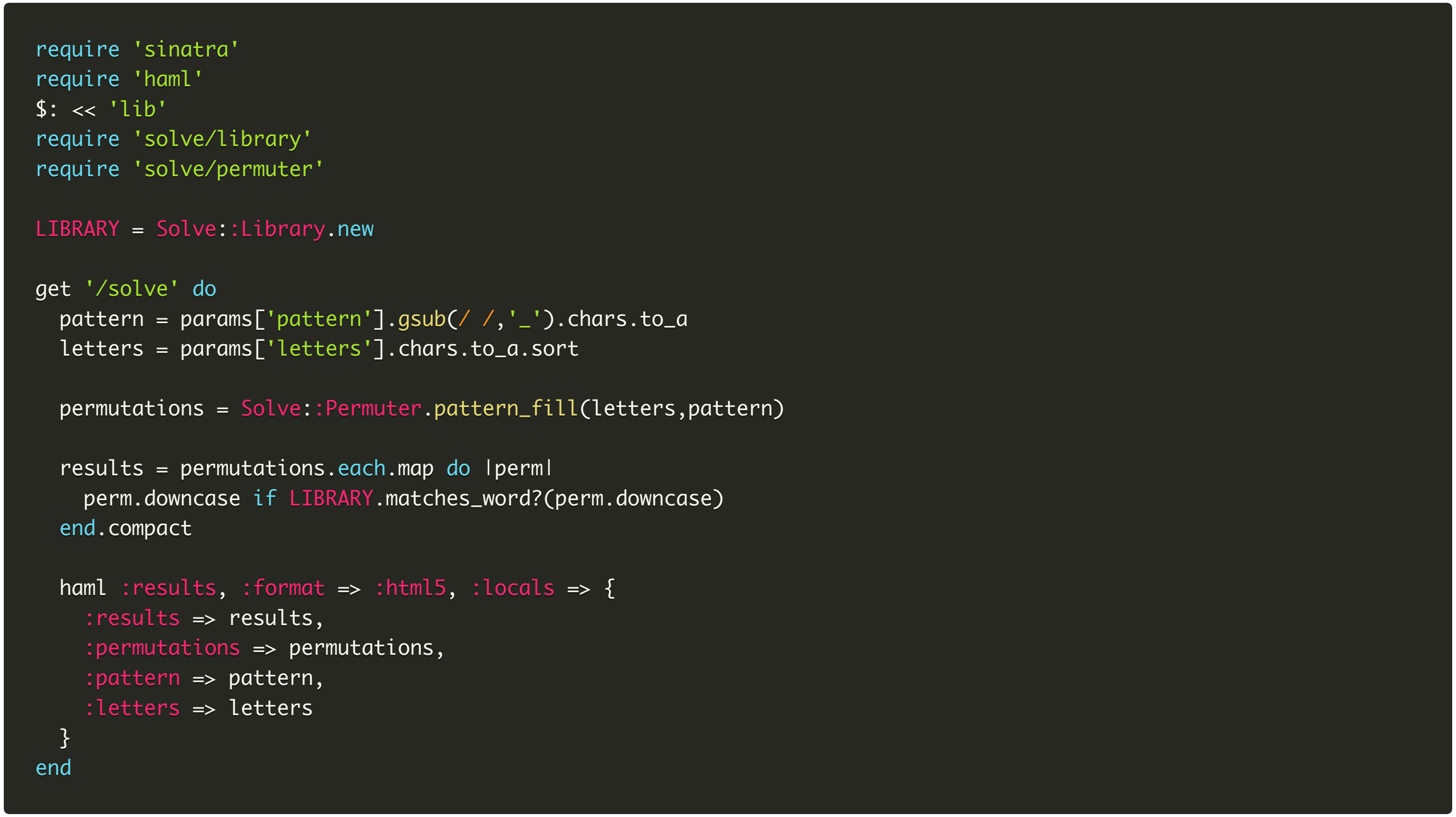
```ruby
require 'solve/dictionary'
module Solve
  class Library

    attr_accessor :dictionaries

    def selected_dictionaries(opts = DEFAULT_OPTS)
      languages = ['english'] + [*opts[:language]]
      level = opts[:level] || DEFAULT_LEVEL

      @dictionaries.select do |d|
        languages.include?(d.language) && (d.level <= level)
      end
    end

  end
end
```



```ruby
require 'solve/dictionary'

describe Solve::Dictionary do
  let :dictionary do
    Solve::Dictionary.new('dict/american-words.10')
  end
  describe "include?" do
    it "should not contain 'colour'" do
      dictionary.should_not include('colour')
    end
    it "should contain 'color'" do
      dictionary.should include('color')
    end
  end

  describe "loading a file" do
    it "should have 35 words" do
      dictionary.words.count.should == 35
    end

    it "should be languange 'american'" do
      dictionary.language.should == 'american'
    end

    it "should be level 10" do
      dictionary.level.should == 10
    end
  end
end
```



```ruby
module Solve
  class Dictionary
    LEVEL_LIMIT = 90
    attr_accessor :language, :level, :words

    def initialize(file_name)
      @language = file_name.match(/(\w+)-words/)[1].to_s
      @level = file_name.match(/\.(\d+)/)[1].to_i
      if @level < 90
        @words = File.readlines(file_name).map{|word| word.chomp}
      end
    end

    def include?(word)
      @words.include?(word)
    end
  end
end
```



```ruby
require 'sinatra'
require 'haml'
$: << 'lib'
require 'solve/library'
require 'solve/permuter'

LIBRARY = Solve::Library.new

get '/solve' do
  pattern = params['pattern'].gsub(/ /,'_').chars.to_a
  letters = params['letters'].chars.to_a.sort

  permutations = Solve::Permuter.pattern_fill(letters,pattern)

  results = permutations.each.map do |perm|
    perm.downcase if LIBRARY.matches_word?(perm.downcase)
  end.compact

  haml :results, :format => :html5, :locals => {
    :results => results,
    :permutations => permutations,
    :pattern => pattern,
    :letters => letters
  }
end
```

Thanks!

Evan Dorn - Honey