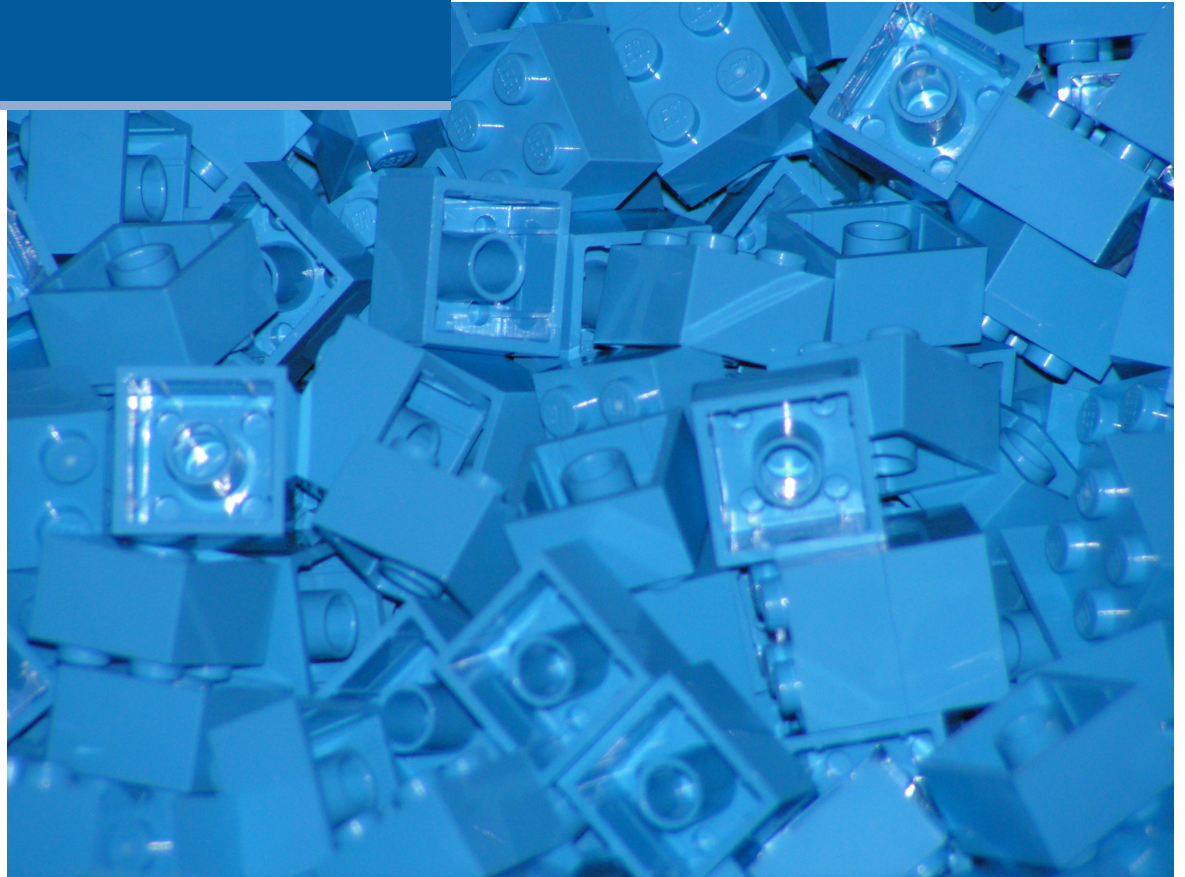


Refactoring complex domains in Ruby and Rails

Shane Harvie
shane@shaneharvie.com

I lied to you



In the outline for this presentation it said:

"This session will be light on the slides, and heavy on the code"

the second part is true

it is "heavy on the code"

but I put the code into slides

so it's also "heavy on the slides"

plus I do this partial
sentence per slide thing

which isn't helping my slide count

anyway...

Outline

- • Delegation, Inheritance, and Module extension
- • State pattern without delegation
- • Complex Domains in Rails
- • Separate Query From Modifier

Inheritance is bad, right?

Not necessarily

When would you
use inheritance?

When would you use inheritance?

- • When the subclasses use a large portion of the superclass's behavior
- • When the subclasses have common state

Let's start with some base class objects which we are using to construct SQL joins:

```
class LeftOuterJoin < Join
  def join_type
    "LEFT OUTER"
  end
end

class InnerJoin < Join
  def join_type
    "INNER"
  end
end
```

They can be used like so:

```
InnerJoin.new(:equipment_listings, :on => "equipment_listings.listing_id =  
listings.id").to_sql
```

The superclass looks like this:

```
class Join...

  def initialize(table, options)
    @table = table
    @on = options[:on]
  end

  def to_sql
    "#{join_type} JOIN #{@table} ON #{@on}"
  end

end
```


A good inheritance candidate

- • Has only a few methods in its subclasses
- • This indicates a strong "is-a" relationship
- • InnerJoin *is* basically a Join, with only some minor differences
- • The interface to join also appropriately represents the interface to its subclasses

But inheritance can be
used inappropriately

Inheritance used inappropriately

- When you find that many of the superclass methods aren't used by all of the subclasses
- The superclass becomes a dumping ground for behavior that's shared between some subclasses, but not all
- In this case, the superclass does not reflect the interface of the subclasses
- The superclass often ends up with an obscure name that doesn't make sense in the domain



So we might use
Replace Inheritance
with Delegation

Replace Inheritance with Delegation

Here I have a policy class which inherits from Hash. Each hash value is an array of rules, and policy gives the hash an array-like interface by implementing the << operator:

```
class Policy < Hash
  attr_reader :name

  def initialize(name)
    @name = name
  end

  def <<(rule)
    key = rule.attribute
    self[key] ||= []
    self[key] << rule
  end

  def apply(account)
    self.each do |attribute, rules|
      rules.each { |rule| rule.apply(account) }
    end
  end
end
```

Replace Inheritance with Delegation

The Rule class has an attribute and default value:

```
class Rule...
  attr_reader :attribute, :default_value

  def initialize(attribute, default_value)
    @attribute, @default_value = attribute, default_value
  end

  def apply(account)
    ...
  end
end
```

Replace Inheritance with Delegation

Looking at the users of policy, I realize that clients do only five things:

- `<<`
- `apply`
- `[]`
- `size`
- `empty?`

The latter three are inherited from Hash

By adding the `<<` operator,
we have a confusing interface



Our code isn't very intentional

It says it's a hash, but it isn't really

Best we remove this source
of confusion

Replace Inheritance with Delegation

Forwardable makes this very easy in ruby

```
require 'forwardable'

class Policy
  extend Forwardable

  def_delegators :@rules, :size, :empty?, :[]

  def initialize(name)
    @name = name
    @rules = {}
  end
end
```

And if we needed to, we
could test the delegate
in isolation

But when is Replace
Inheritance with
Delegation difficult?

When state is used in both
the superclass and subclass

Delegation Difficulties

```
class Bicycle

  def wheel_circumference
    Math::PI * (@wheel_diameter + @tire_diameter)
  end

end

class FrontSuspensionMountainBike < Bicycle

  def off_road_ability
    @tire_diameter * TIRE_WIDTH_FACTOR + @front_fork_travel * FRONT_SUSPENSION_FACTOR
  end

end

class RigidMountainBike < Bicycle

  def off_road_ability
    @tire_diameter * TIRE_WIDTH_FACTOR
  end


end

class RoadBike < Bicycle

  def off_road_ability
    raise "You can't take a road bike off-road"
  end

end
```


And what if we wanted to change the type of bike at run-time?



Using the state pattern would be ideal - but again, the shared state makes delegation difficult.

State Pattern using module extension

State Pattern using module extension

```
mountain_bike = RigidMountainBike.new  
front_suspension_bike = FrontSuspensionMountainBike.new
```

becomes

```
mountain_bike = Bicycle.new.extend(RigidMountainBike)  
front_suspension_bike = Bicycle.new.extend(FrontSuspensionMountainBike)
```

State Pattern using module extension

```
class Bicycle

  def wheel_circumference
    Math::PI * (@wheel_diameter + @tire_width)
  end

end

module FrontSuspensionMountainBike

  def off_road_ability
    @tire_width * TIRE_WIDTH_FACTOR + @front_fork.travel * FRONT_SUSPENSION_FACTOR
  end

end

class RigidMountainBike < Bicycle

  def off_road_ability
    @tire_width * TIRE_WIDTH_FACTOR
  end

end
```

State Pattern using module extension

So we could conceivably upgrade our mountain bike at run-time to add a fork with front suspension

```
mountain_bike = Bicycle.new.extend(RigidMountainBike)
...
mountain_bike.add_front_suspension(fork)

module RigidMountainBike


  def add_front_suspension(fork)
    @front_fork = fork
    extend(FrontSuspensionMountainBike)
  end

end

end
```

State Pattern using module extension

So we now have the ability to change behavior at run-time, but we haven't introduced any duplication with our state pattern - the data and behavior is still shared.



But there's a problem

State Pattern using module extension

We were able to mixin the FrontSuspensionMountainBike behavior, but the RigidMountainBike behavior still exists on the object

```
bike.kind_of?(FrontSuspensionMountainBike) => true  
bike.kind_of?(RigidMountainBike) => true
```

This isn't ideal

ruby doesn't provide the
ability to unmix a module

mixology

open-source library

State Pattern using module extension

```
require 'mixology'

module RigidMountainBike

  def add_front_suspension(fork)
    @front_fork = fork
    unmix(RigidMountainBike)
    mixin(FrontSuspensionMountainBike)
  end

end

bike.kind_of?(RigidMountainBike) => false
```

Evolution of a domain in rails

Evolution of a domain in rails

- • Fat controllers
- • Service layer
- • Behavior on domain objects

An Example

Disclaimer

Example

- Let's say we're modeling a network
- In our network, we have nodes
- A node can be physically connected to another node, or connect wirelessly at a certain frequency
- Nodes can be in different buildings, but nodes in two different buildings cannot be connected wirelessly, even if they are on the same frequency. So a building represents the boundary for a wireless connection - in order to cross from building to building, you need a physical connection
- A node can have both a physical and wireless connection to another node, but they cannot have more than one physical connection

Design

- • So let's say we have a network object to keep track of all nodes on the one network
- • When two nodes on different networks are connected, the networks merge into one
- • When two nodes are disconnected (and do not share a common frequency), the networks split
- • When the frequency of a node is changed such that it matches another node's frequency, the two node's networks merge
- • When the frequency of a node is changed such that it no longer matches another node's frequency, the two node's networks split



MVC provides a good guide
as to where to place behavior

If it's view-related, it goes in
the View

If it's related to the web

Related to the web?

- • Parsing http parameters
- • Redirecting to a different page
- • Rendering a view

It should go in the controller

If it needs to be saved in
a database

It should go in the model

After all, models inherit
from `ActiveRecord::Base`

and `ActiveRecord::Base` is
used for saving to a database

But what if an operation
involves multiple models?



We could put the logic in
the controller

```
class NodeConnectionController < ActionController::Base ...

  def create
    origin_node = Node.find(params[:origin_id])
    destination_node = Node.find(params[:destination_id])
    origin_node.connected_node = destination_node
    if destination_node
      origin_node.network.destroy
      destination_node.network.destroy
      new_network = Network.new(:label => "#{origin_node.network.label} -
#{destination_node.network.label}")
      add_nodes_to(new_network, origin_node)
    end
    redirect_to network_url(origin_node.network)
  end

  private

  def add_nodes_to(network, node)
    node.yield_nodes_destined_for_same_network do |yielded_node|
      yielded_node.network = network
      yielded_node.save!
    end
  end
end
```

- • But now we have this domain logic in our controller, which makes it difficult to reuse
- • If we have different controllers that need to trigger the connection of nodes, we're in trouble
- • We'd have to duplicate this code

our controller has also
become a little fat

or at least big boned

Service Objects

Service Objects

From Domain Driven Design:

- Service objects are objects "with no state of their own nor any meaning in the domain beyond the operation they host"

Service Object

```
class NodeConnectionController < ActionController::Base ...

  def create
    origin_node = Node.find(params[:origin_id])
    destination_node = Node.find(params[:destination_id])

    NetworkService.connect(origin_node, destination_node)

    redirect_to network_url(origin_node.network)
  end

class NetworkService

  class << self

    def connect(node1, node2)
      node1.connected_node = node2
      if node2
        node1.network.destroy
        node2.network.destroy
        new_network = Network.new(:label => "#{node1.network.label} - #{
node2.network.label}")
        add_nodes_to(new_network, node1)
      end
    end

    private

    def add_nodes_to(network, node)
      node.yield_nodes_destined_for_same_network do |yielded_node|
        yielded_node.network = network
        yielded_node.save!
      end
    end
  end
end
```

end

end

end

We could also add a disconnect method to our service

```
class NetworkService


  class << self

    def disconnect(node)
      old_connected_node = node.connected_node
      if old_connected_node
        node.connected_node = nil
        old_network = node.network
        add_nodes_to(Network.new(:label => "#{old_network.label}
B"), old_connected_node)
        add_nodes_to(Network.new(:label => "#{old_network.label} A"), node)
        old_network.destroy
      end
    end
  end
end
```

Service Objects

So these services give us the opportunity for code reuse - the different controllers that need to trigger the connection and disconnection of nodes can all use these services

A service layer gave us relief
for a while



I wasn't comfortable with this
(relatively) complex logic being in a service
layer, but it solved our reuse problem.

the next requirements

New requirements

- • The user would like to preview the network change that is about to take place
- • They want the ability to confirm or cancel the change
- • And they want to be able to name the new network.

dang

New requirements

With the way the algorithm works currently, we don't know in advance the new make up of the network

- • We destroy the old networks before we start building up the new networks
- • This seems to be a common occurrence with rails apps

It can be difficult to re-wire multiple ActiveRecord objects of different type together according to some set of business rules without using the database as a temporary storage mechanism

Why is that?

This test passes

```
def test_array_append_when_associated_object_had_an_existing_parent
  book = Book.create!
  book2 = Book.create!
  chapter = Chapter.create!(:book => book)
  assert_equal 1, book.reload.chapters.size

  book2.chapters << chapter
  book2.save!
  assert_equal 0, book2.reload.chapters.size
  assert_equal book, chapter.reload.book
end
```

It can sometimes be awkward
to associate objects
without saving them as you go

separate query from modifier

The problem:

- • The logic that decides how to do the change (or whether a change is needed at all) is mixed up with the logic that makes the change
- • The algorithm relies on making a change to the network (connecting or disconnecting a port), saving that change, and then sorting out which nodes belong on each network

If we are to separate query
from modifier, we have to
be able to propose a
connection change

Let's look at our `yield_nodes_destined_for_same_network` method

```
class Node < ActiveRecord::Base...

def yield_nodes_destined_for_same_network(visited_nodes=[], &block)
  unless visited_nodes.include?(self)
    visited_nodes << self
    yield self
  end
  yield_connected_node(visited_nodes, &block)
  if frequency
    building.nodes_with_frequency(self.frequency).each do |node|
      unless visited_nodes.include?(node)
        node.yield_nodes_destined_for_same_network(visited_nodes, &block)
      end
    end
  end
end

def yield_connected_node(visited_nodes, &block)
  if connected_node && !visited_nodes.include?(connected_node)
    connected_node.yield_nodes_destined_for_same_network(visited_nodes, &block)
  end
end
```

This method works fairly well. It doesn't modify anything, but it relies on the network having already been modified



We can refactor this method
to take a proposed connection

yield_nodes_destined_for_same_network with proposed connection

Calling code:

```
origin_node.yield_nodes_destined_for_same_network(node2 => node3) do |yielded_node|
  ...
end
```

And the method:

```
def yield_nodes_destined_for_same_network(proposed_connection={}, visited_nodes=
[], &block)
  unless visited_nodes.include?(self)
    visited_nodes << self
    yield self
  end
  yield_connected_node(proposed_connection, visited_nodes, &block)
  if frequency
    building.nodes_with_frequency(self.frequency).each do |node|
      unless visited_nodes.include?(node)
        node.yield_nodes_destined_for_same_network
        (proposed_connection, visited_nodes, &block)
      end
    end
  end
end

def yield_connected_node(proposed_connection, visited_nodes, &block)
  if proposed_connection.keys[0] == self
    if !proposed_connection[self].nil?
      proposed_connection[self].

```

```
yield_nodes_destined_for_same_network(proposed_connection, visited_nodes, &block)
  end
  return
end
if connected_node && !visited_nodes.include?(connected_node)
  connected_node.yield_nodes_destined_for_same_network
(proposed_connection, visited_nodes, &block)
end
end
```

We can then write a class method on Network that will return the new networks that would be created if we completed the proposed connection

```
class Network...

  class << self
    def new_networks_for_connection(proposed_connection)
      origin_node = proposed_connection.keys.first
      old_connected_node = origin_node.connected_node

      allocated_nodes, old_networks, new_networks = [], [], []

      [origin_node, old_connected_node].compact.each do |node|
        unless allocated_nodes.include?(node)
          new_network = Network.new
          node.yield_nodes_destined_for_same_network(proposed_connection) do
            |yielded_node|
              unless allocated_nodes.include?(yielded_node)
                allocated_nodes << yielded_node
                old_networks << yielded_node.network unless old_networks.
include?(yielded_node.network)
                new_network.nodes << yielded_node
              end
            end
            new_networks << new_network
          end
        end
      end
      [new_networks, old_networks]
    end
  end
end
```

But there's a small problem
with this code

If we wanted to save the new networks that get returned


```
proposed_networks, old_networks = Network.new_networks_for_connection(node1  
=> node2)  
proposed_networks.each  
(&:save!)
```

we can't, because we need a handle on the nodes to associate them with the Networks

We'd have to do something like this

```
proposed_networks, old_networks = Network.new_networks_for_connection(node1
=> node2)

proposed_networks.each do |network|
  network.nodes.each do |node|
    node.network = network
    node.save!
  end
end
```



But this is getting a bit clunky. We're returning two arrays from the method, and we need to perform complex logic on the result to get it to do something useful for us

It's time for a...

Results Object

Results object

One of the simplest refactorings is a results object, but it so often gets overlooked. We seem to like to piggy back on a language's collections to do the heavy work for us. And then we've nowhere to place behavior like the saving logic above

Results object

We could do something like this:

NewNetworkResult

```
unless allocated_nodes.include?(node)
  new_network = NewNetworkResult.new
  node.yield_nodes_destined_for_same_network(proposed_connection) do |yielded_node|
    unless allocated_nodes.include?(yielded_node)
      allocated_nodes << yielded_node
      new_network.old_networks << yielded_node.network unless
new_network.old_networks.include?(yielded_node.network)
      new_network.nodes << yielded_node
    end
  end
  new_networks << new_network
end
```

Results object

And our NewNetworkResult is a simple object:

```
class NewNetworkResult

  attr_accessor :nodes, :old_networks

  def initialize
    @nodes, @old_networks = [], []
  end

  def save!
    new_network = Network.new
    @nodes.each do |node|
      node.network = new_network
      node.save!
    end
    new_network.save!
  end

end
```

Results object

We can then do this:

```
proposed_networks = Network.new_networks_for_connection(node1 => node2)
proposed_networks.each(&:save!)
```

And our new networks are saved

So can we use our
new approach for our
original connection logic?

This is where we left our NetworkService

```
class NetworkService

  class << self

    def connect(node1, node2)
      node1.connected_node = node2
      if node2
        node1.network.destroy
        node2.network.destroy
        new_network = Network.new(:label => "#{node1.network.label} - #{node2.network.label}")
        add_nodes_to(new_network, node1)
      end
    end

    def disconnect(node)
      old_connected_node = node.connected_node
      if old_connected_node
        node.connected_node = nil
        old_network = node.network
        add_nodes_to(Network.new(:label => "#{old_network.label} B"), old_connected_node)
        add_nodes_to(Network.new(:label => "#{old_network.label} A"), node)
        old_network.destroy
      end
    end

    private

    def add_nodes_to(network, node)
```

```
    node.yield_nodes_destined_for_same_network do |yielded_node|
      yielded_node.network = network
      yielded_node.save!
    end
  end

end

end
```

If we extract a network building method

```
class Network < ActiveRecord::Base
  def self.new_network_results_for(nodes, proposed_connection={})
    new_network_results, allocated_nodes = [], []
    nodes.each do |node|
      unless allocated_nodes.include?(node)
        new_network_result = NewNetworkResult.new
        node.yield_nodes_destined_for_same_network(proposed_connection) do
|yielded_node|
          unless allocated_nodes.include?(yielded_node)
            allocated_nodes << yielded_node
            new_network_result.old_networks << yielded_node.network
          unless new_network_result.old_networks.include?(yielded_node.network)
            new_network_result.nodes << yielded_node
          end
        end
        new_network_results << new_network_result
      end
    end
    new_network_results
  end
end
```

We can modify the connect method to use our extracted method

```
def connect(node1, node2)
  node1.connected_node = node2
  new_network_results = Network.new_network_results_for(node1.network.nodes)
  old_networks = new_network_results.map(&:old_networks).flatten.uniq
  if node2
    new_network_results.each do |new_network_result|
      new_network_result.label = old_networks.collect(&:label).join("-")
      new_network_result.save!
    end
  end
  old_networks.each(&:destroy)
end
```

Then we can do the same for the disconnect method

```
def disconnect(node)
  node.connected_node = nil
  new_network_results = Network.new_network_results_for(node.network.nodes)
  old_networks = new_network_results.map(&:old_networks).flatten.uniq

  label_appendage = 'A'
  new_network_results.each do |new_network_result|
    new_network_result.label = "#{old_networks.collect(&:label).join(" - ")}
#{label_appendage}"
    new_network_result.save!
    label_appendage.next!
  end
  old_networks.each(&:destroy)
end
```

We can then make our connect method handle nil as the second argument, and make disconnect a convenience method

Extract method helps clean up the code a little bit

```
class NetworkService

  class << self

    def connect(node1, node2)
      node1.connected_node = node2

      new_network_results = Network.new_network_results_for(node1.network.nodes)
      old_networks = new_network_results.map(&:old_networks).flatten.uniq
      if new_network_results.size > old_networks.size
        split_networks(new_network_results, old_networks)
      elsif new_network_results.size < old_networks.size
        merge_networks(new_network_results, old_networks)
      end
      old_networks.each(&:destroy)
    end

    def disconnect(node)
      connect(node, nil)
    end

  private

    def split_networks(new_network_results, old_networks)
      label_appendage = 'A'
      new_network_results.each do |new_network_result|
        new_network_result.label = "#{old_networks.collect(&:label).join(" - ")}

```

```
#{label_appendage}"
  new_network_result.save!
  label_appendage.next!
end
end

def merge_networks(new_network_results, old_networks)
  new_network_results.each do |new_network_result|
    new_network_result.label = old_networks.collect(&:label).join(" - ")
    new_network_result.save!
  end
end
end
end
```

And you can see that the network object is quite instrumental in this service method

```
def connect(node1, node2)
  node1.connected_node = node2

  new_network_results = Network.new_network_results_for(node1.network.nodes)
  old_networks = new_network_results.map(&:old_networks).flatten.uniq
  if new_network_results.size > old_networks.size
    split_networks(new_network_results, old_networks)
  elsif new_network_results.size < old_networks.size
    merge_networks(new_network_results, old_networks)
  end
  old_networks.each(&:destroy)
end
```

We can use Extract Method and Move Method to move the majority of the connect method onto our Network object:

```
class NetworkService

  class << self

    def connect(node1, node2)
      node1.connected_node = node2
      node1.network.update_network
    end

    def disconnect(node)
      connect(node, nil)
    end
  end
end
```

```
class Network < ActiveRecord::Base

  def self.new_networks_for_connection(proposed_connection)
    origin_node = proposed_connection.keys.first
    old_connected_node = origin_node.connected_node

    new_network_results_for([origin_node, old_connected_node].
compact, proposed_connection)
  end

  def update_network
    new_networks = self.class.new_network_results_for(nodes)
    old_networks = new_networks.map(&:old_networks).flatten.uniq
    if new_networks.size > old_networks.size
      split_networks(new_networks, old_networks)
    elsif new_networks.size < old_networks.size
      merge_networks(new_networks, old_networks)
    end
    old_networks.each(&:destroy)
  end

  private

  def self.new_network_results_for(nodes, proposed_connection={})
    new_network_results, allocated_nodes = [], []
    nodes.each do |node|
      unless allocated_nodes.include?(node)
        new_network_result = NewNetworkResult.new
        node.yield_nodes_destined_for_same_network(proposed_connection) do
|yielded_node|
          unless allocated_nodes.include?(yielded_node)
            allocated_nodes << yielded_node
            new_network_result.old_networks << yielded_node.network

```

```
unless new_network_result.old_networks.include?(yielded_node.network)
  new_network_result.nodes << yielded_node
end
end
new_network_results << new_network_result
end
end
new_network_results
end

def self.split_networks(new_network_results, old_networks)
  label_appendage = 'A'
  new_network_results.each do |new_network_result|
    new_network_result.label = "#{old_networks.collect(&:label).join(" - ")}
#{label_appendage}"
    new_network_result.save!
    label_appendage.next!
  end
end

def self.merge_networks(new_network_results, old_networks)
  new_network_results.each do |new_network_result|
    new_network_result.label = old_networks.collect(&:label).join(" - ")
    new_network_result.save!
  end
end
```

Conclusion

- • We have our service object which provides convenient use if we're dealing with nodes
- • But the service object itself is very simple
- • By separating query from modifier we've been able to move the domain logic onto the model objects
- • We've also been able to get more reuse - we use the 'query' code in two places, and the modification in one
- • The results object was instrumental in getting us around the awkwardness of modifying associations

Questions?