# Domain Driven Design Hexagonal Architecture Rails

Declan Whelan @dwhelan



Eric Roberts

@eroberts



## 1. Embrace complexity





# 2. Know where you're going

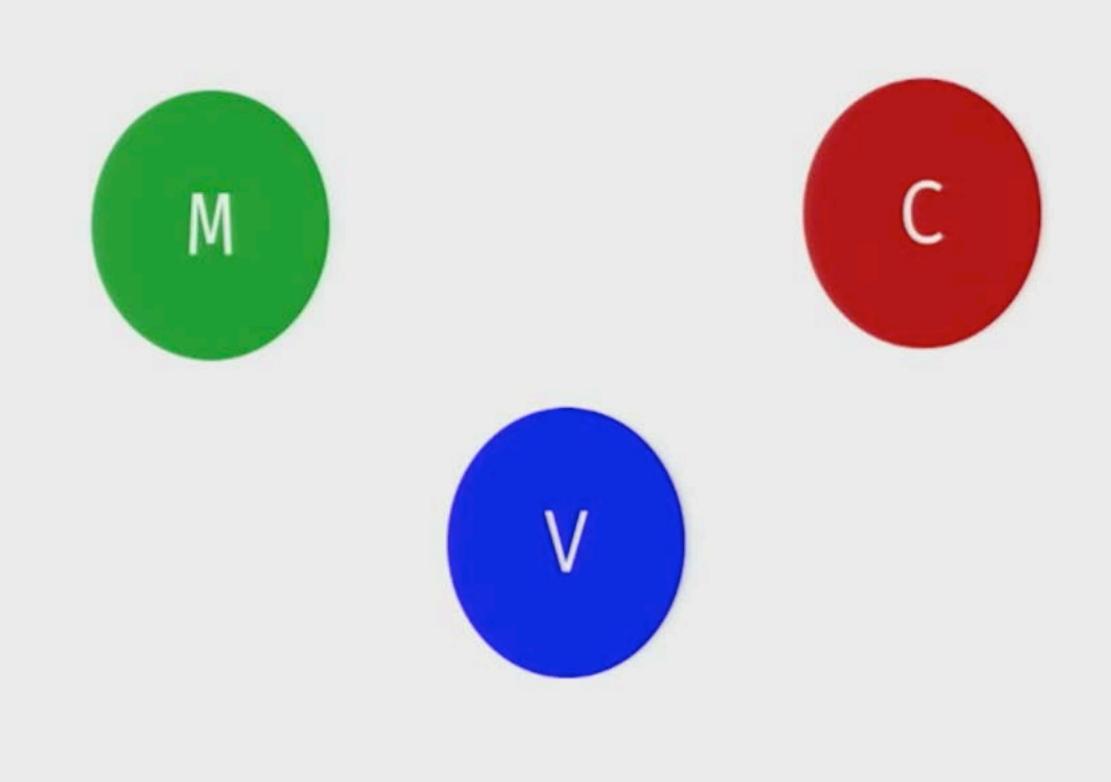


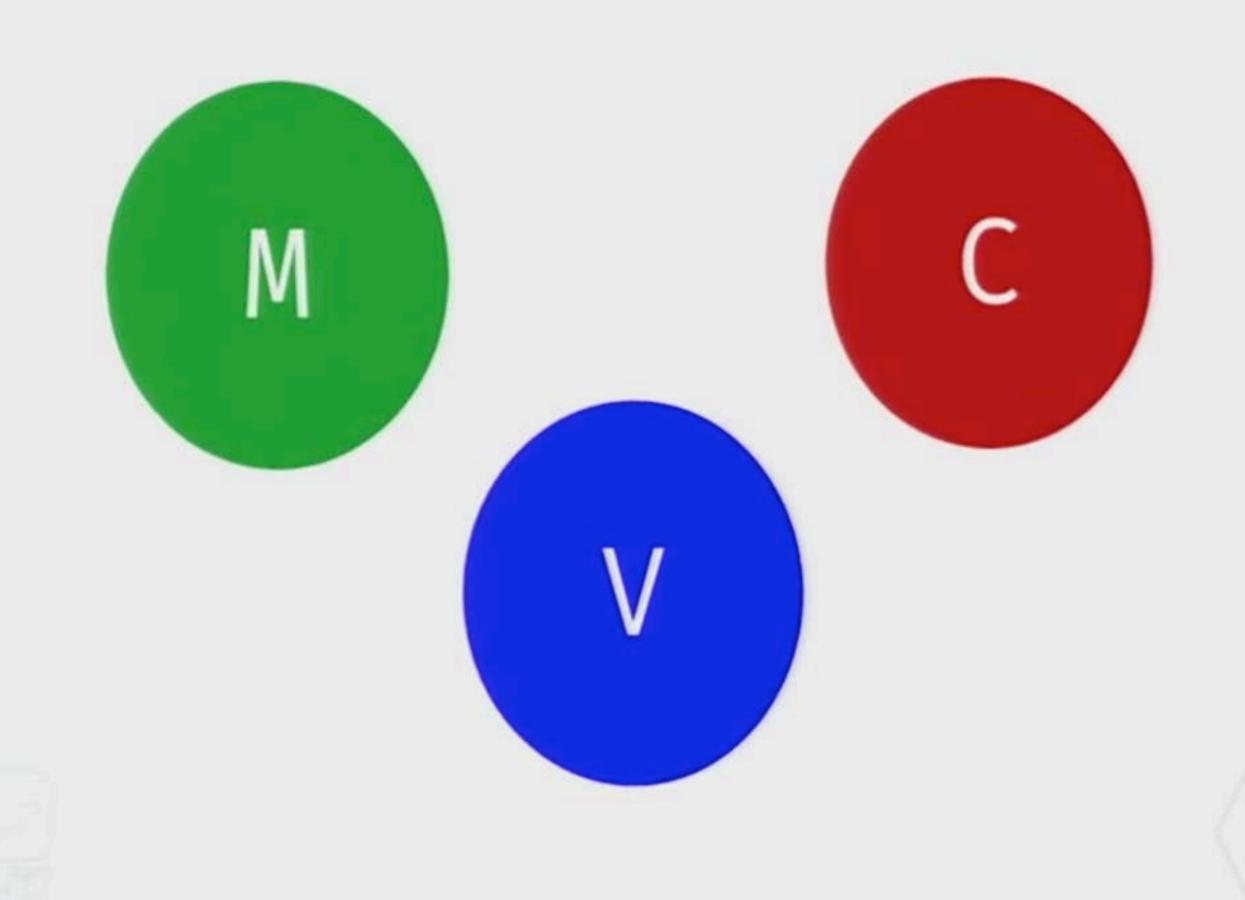


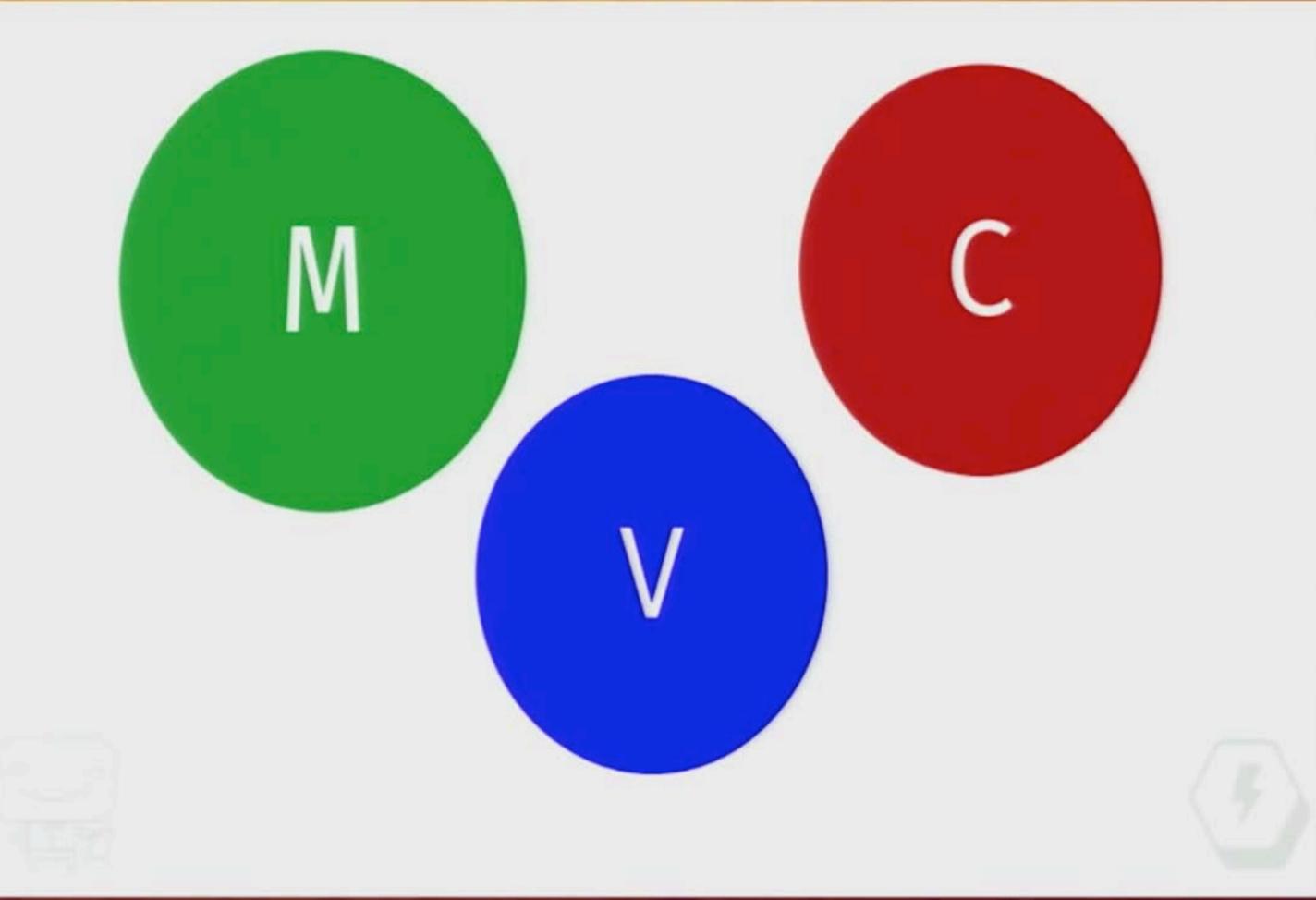
# 3. Be more than just a "Rails Developer"

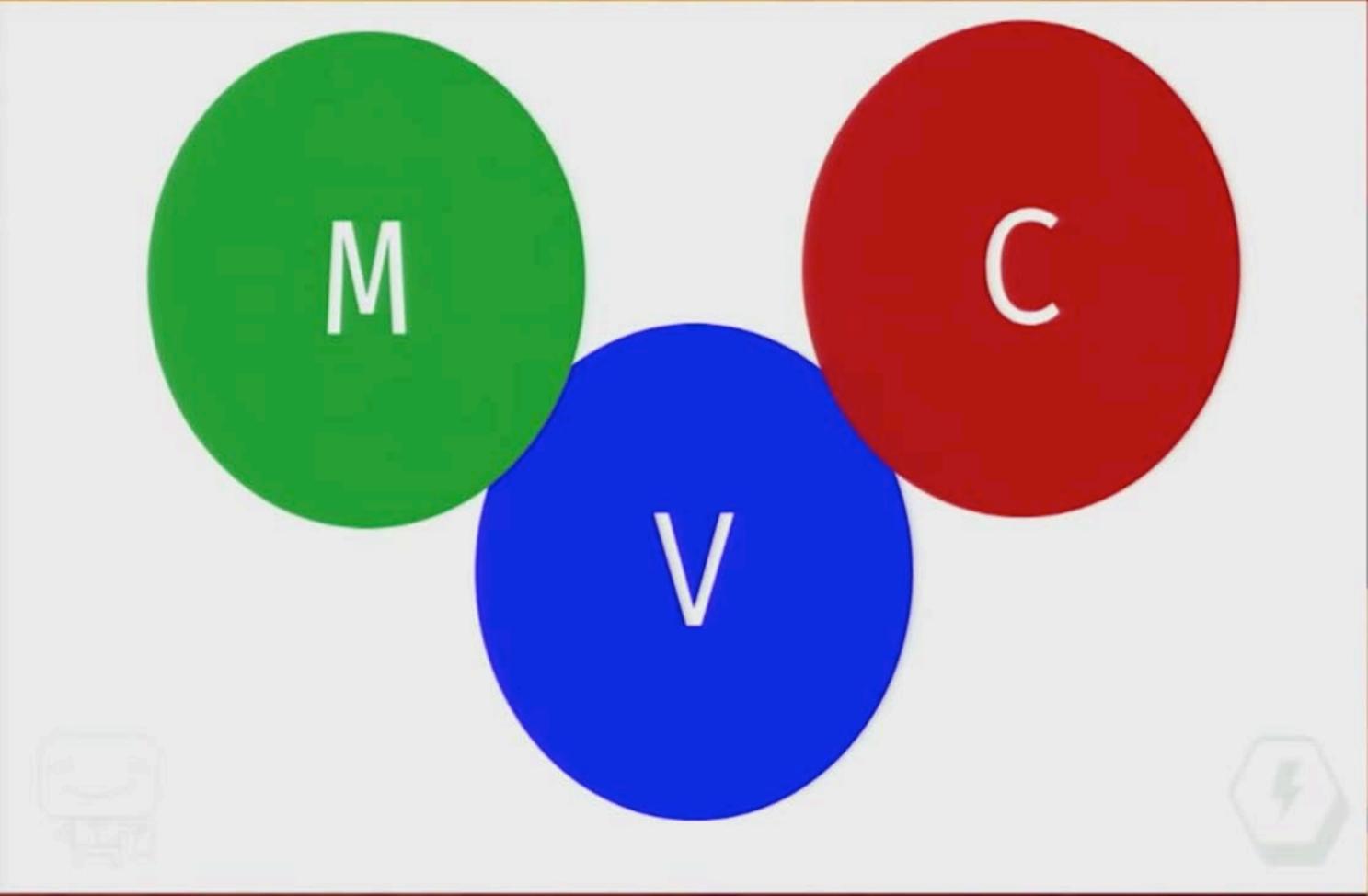












```
oricings each with index on (e) 11
     while pricings[1] present? be now range and date to pricings[1] and date
  PIRIT New range start date <> e.ond date 55 new range start date >> e.start date
     If new range start date += e.start date
```

```
pricings[1] end date - new range start date-1.day
  if i to # AN pricing and date ++ pricings(i-1) and date
pricings[pos].save
```

100

# Pencil Sharpener





# 7 Patterns to Refactor Fat ActiveRecord Models

Posted by Ribranary on Oct 17th, 2012.

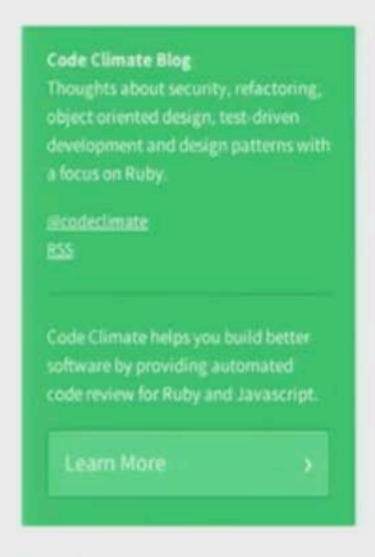


When teams use <u>Code Climate</u> to improve the quality of their Rails applications, they learn to break the habit of allowing models to get fat. "Fat models" cause maintenance issues in large apps. Only incrementally better than cluttering controllers with domain logic, they usually represent a failure to apply the <u>Single Responsibility Principle</u> (SRP). "Anything related to what a user does" is not a <u>single</u> responsibility.

Early on, SRP is easier to apply. ActiveRecord classes handle persistence, associations and not much else. But bit-by-bit, they grow. Objects that are inherently responsible for persistence become the *de facto* owner of all business logic as well. And a year or two later you have a User class with over 500 lines of code, and hundreds of methods in it's *public* interface. Callback hell ensues.

As you add more intrinsic complexity (read: features!) to your application, the goal is to spread it across a coordinated set of small, encapsulated objects (and, at a higher level, modules) just as you might spread cake batter across the bottom of a pan. Fat models are like the big clumps you get when you

first name the better in Defector to break them down and careed out the logic ownly. Deposit this



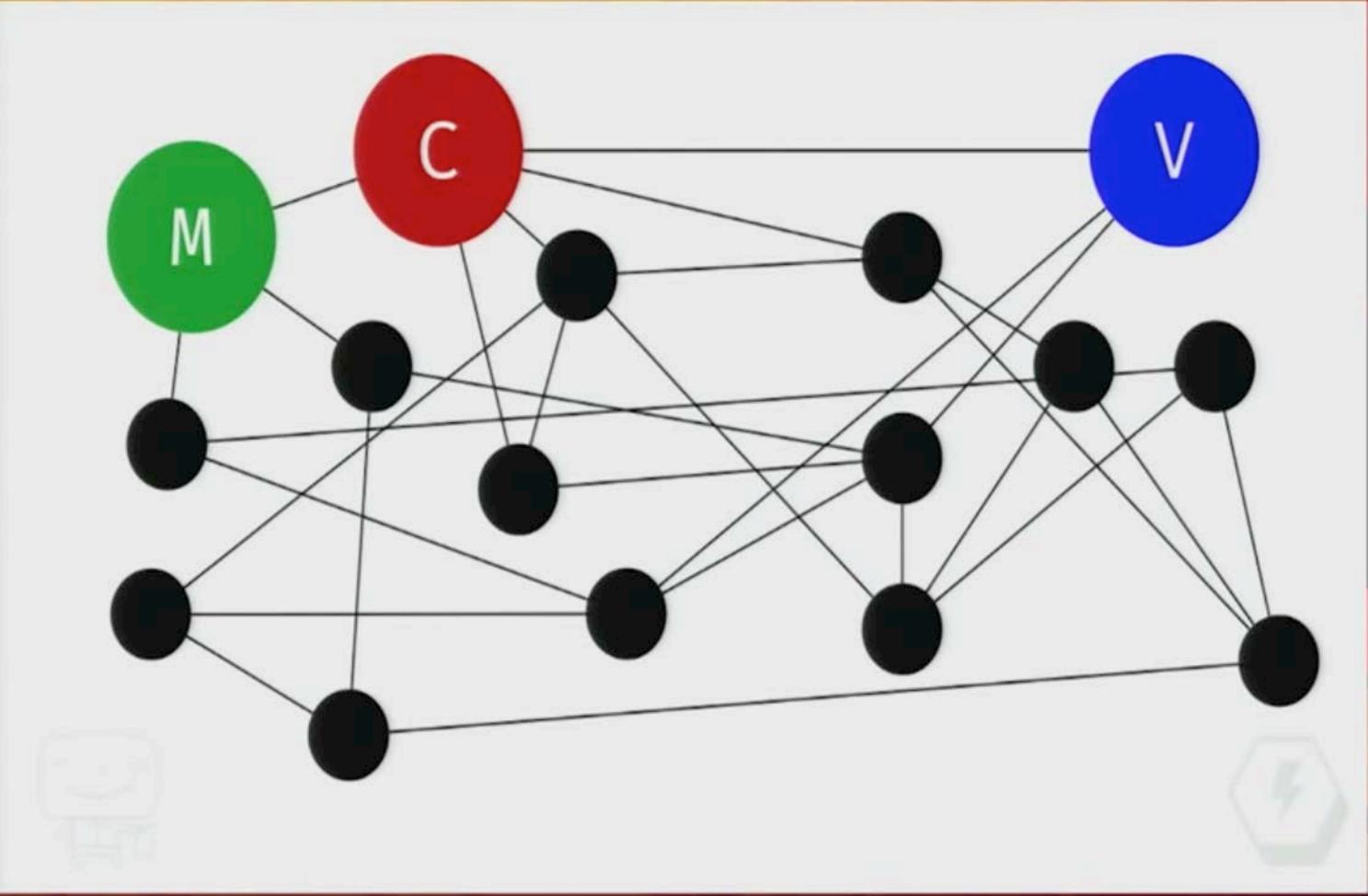
#### **Popular Posts**

7 Patterns to Refactor Fat ActiveRecord Models

Rails' Insecure Defaults

Your Objects, the Unix Way

Sublime Text 2 for Ruby



# Application Domain

# Domain-Driven DRSIGN

Tackling Complexity in the Heart of Software



Eric Evans

Foreword by Martin Fowler

# Complexity

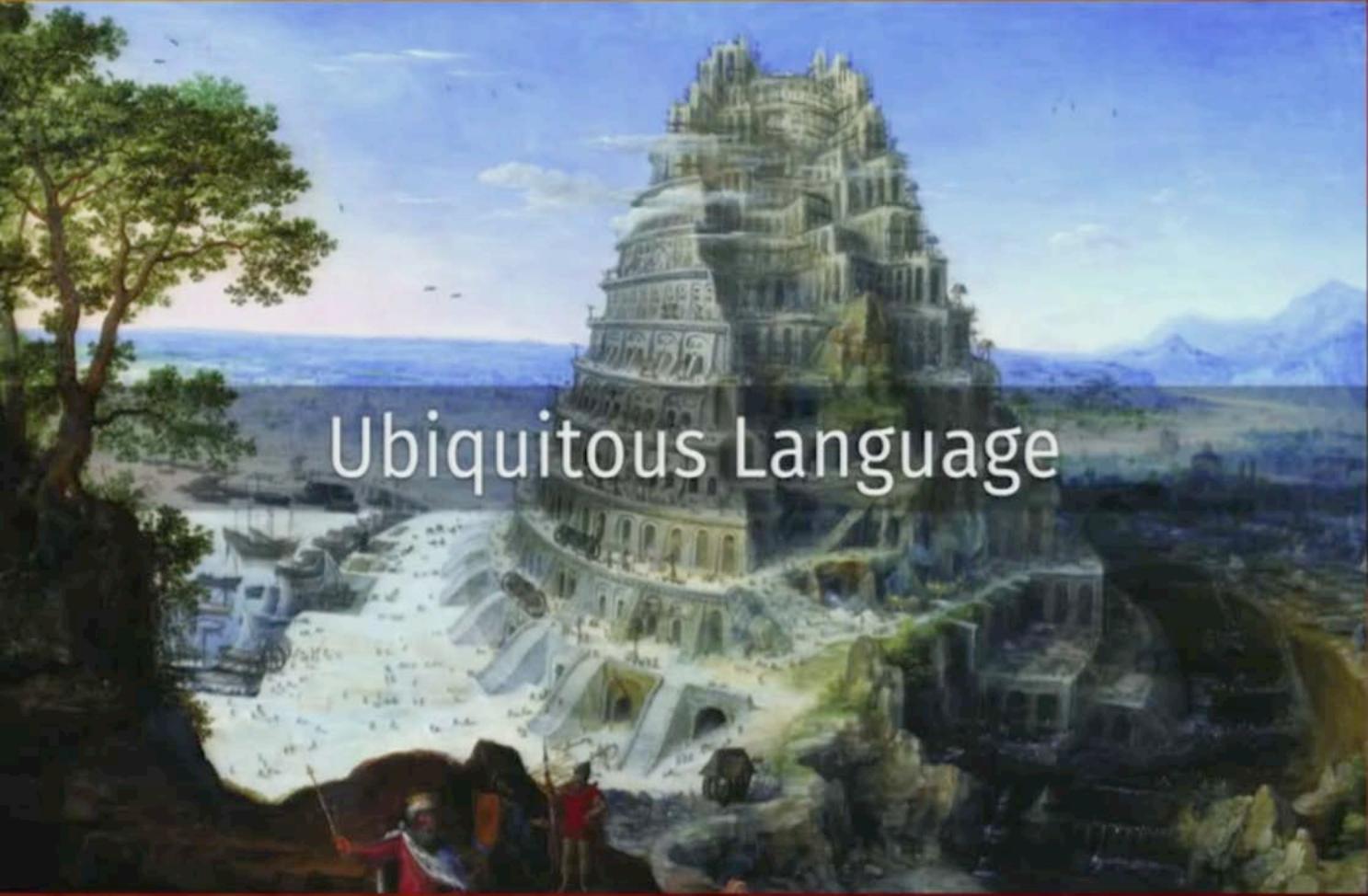


The critical complexity of most software projects is in understanding the domain itself.

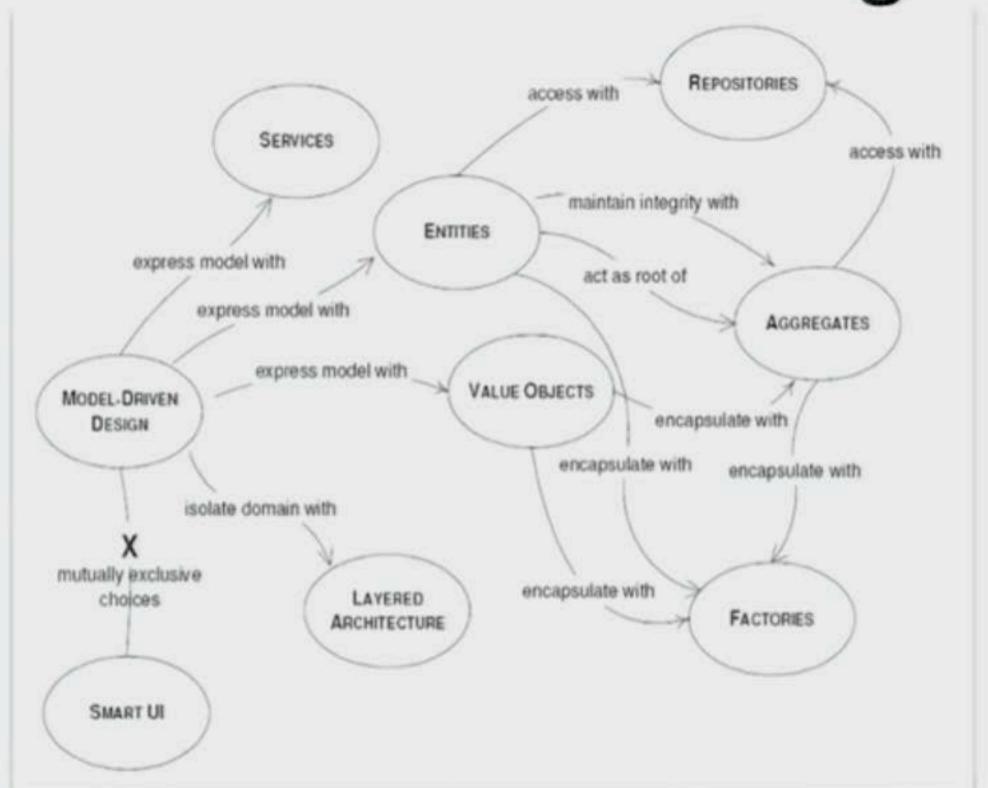
Eric Evans



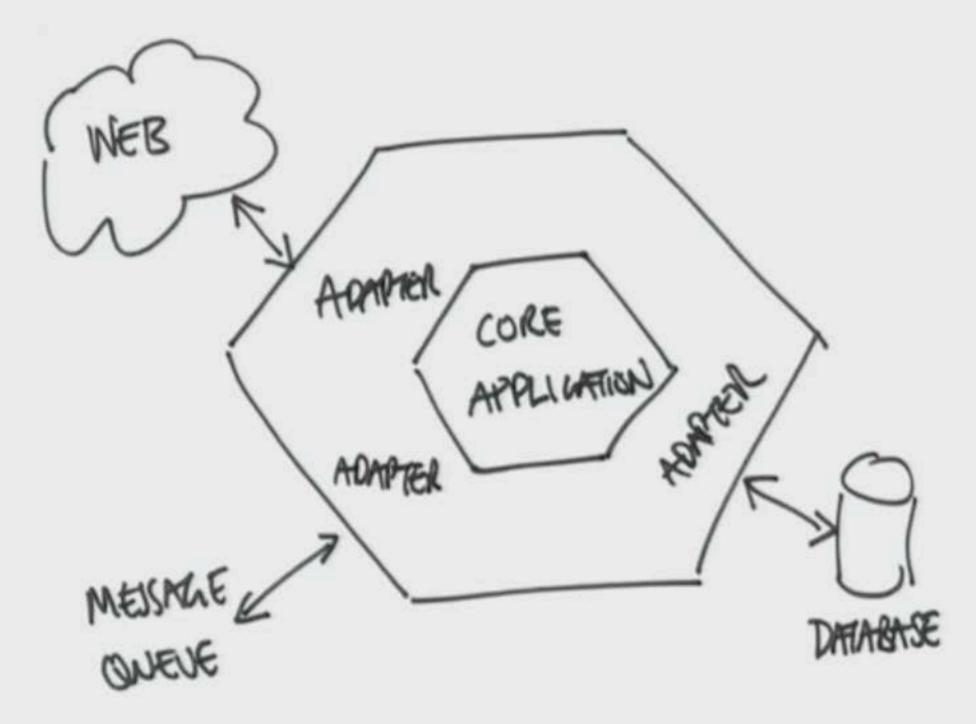


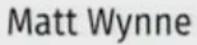


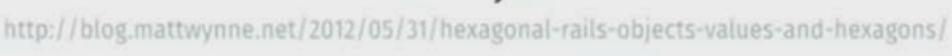
# Domain Driven Design





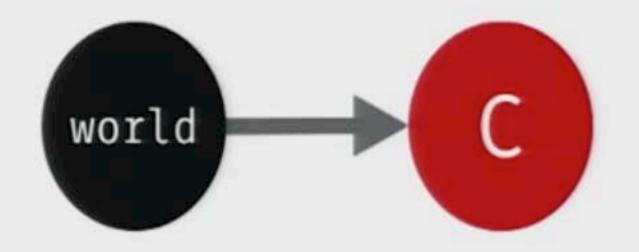






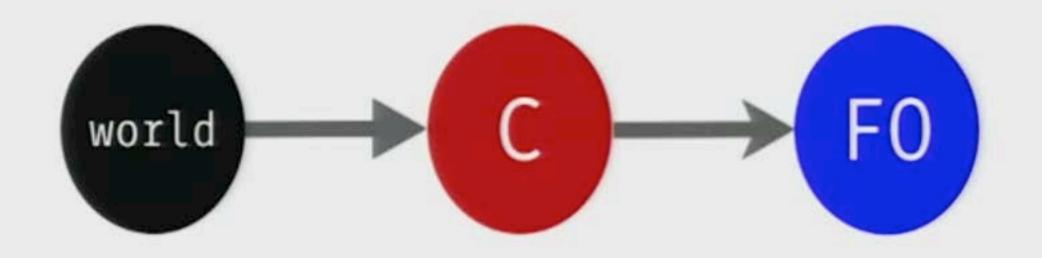


# Form Object





# Form Object





#### Select your ticket

All tickets include first class travel from Toronto to Montreal and full access to the International Startup Festival

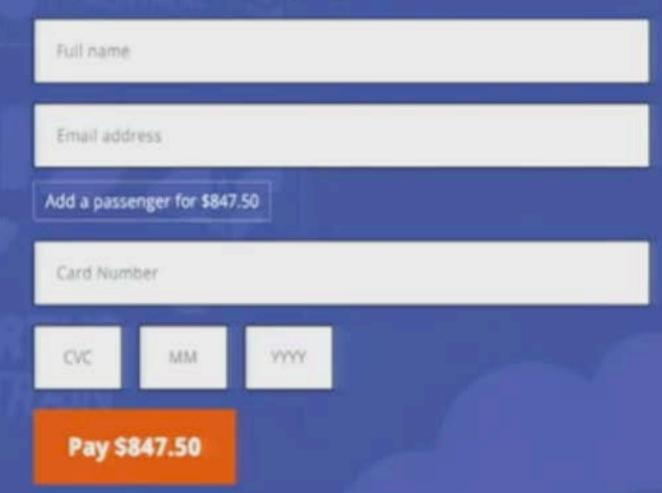
 For Investors and All Others (\$750 + \$97.50 HST)

This ticket grants you access to all Startup Festival activities; from the cocktail party on July 11th (Open House Day activities on July 12th also included). OFF EVENTS NOT INCLUDED.

 For Startups (\$600 + \$78.00 HST)

Must be a Startup to purchase this ticket. Your Startup must be 3 years or younger: 1 person per ticket. This ticket grants you access to all Startup Festival activities; from the cocktail party on July 9th, through to closing of the event on July 11th (Open House Day activities on July 12th also included). OFF EVENTS NOT INCLUDED.

#### Get on board July 9th



```
class TicketForm
  include ActiveModel::Model
  validates :trip, :price, :passengers, presence: true
  def passengers
    @passengers | = [Passenger.new]
  end
  def tickets
    @tickets | = self.passengers.map do | passenger
      Ticket.new(...)
   end
  end
end
```



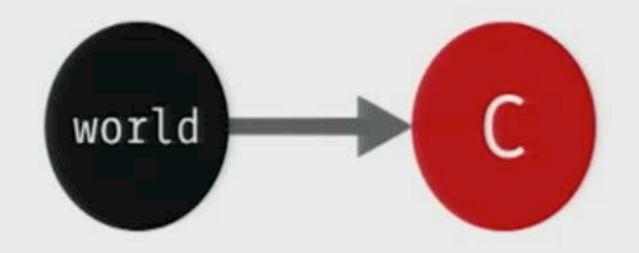
```
class TicketsController < ApplicationController
 def create
    @ticket form = TicketForm.new(params)
    tickets = @ticket form.tickets
    if @ticket form.valid? && TicketCharger.new(tickets).charge!
      redirect to success url
    else
      render 'new'
    end
  end
```



end

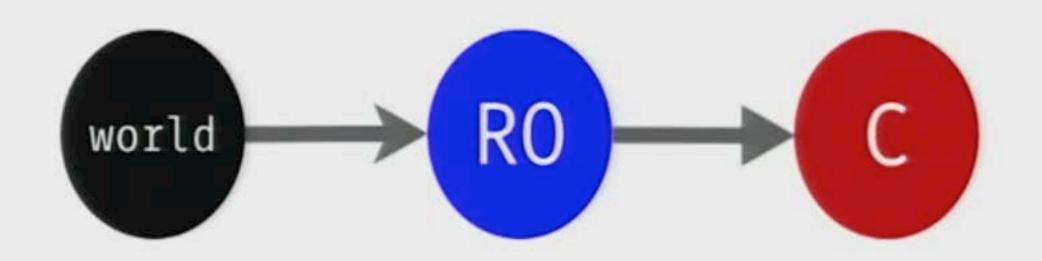


# Request Object





# Request Object



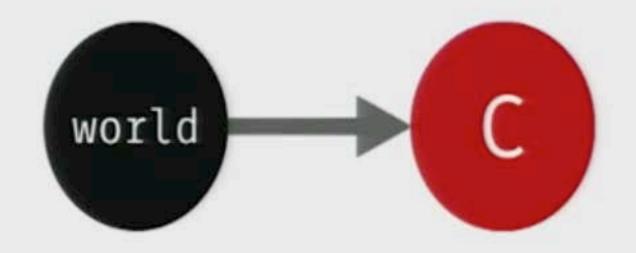
#### class CreateOrderRequest include Virtus.value\_object include ActiveModel::Validations attribute : customer, Customer validates :customer, nested: true, presence: true attribute :billing, Billing validates :billing, nested: true, presence: true attribute :shipping, Shipping validates :shipping, nested: true, presence: true

end

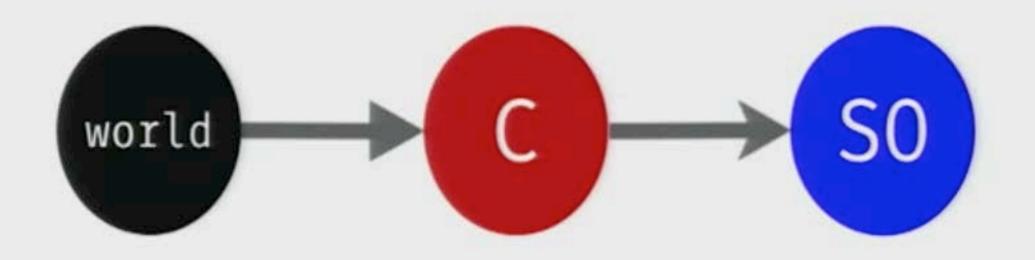


```
class ApplicationController < ActionController::Base
 before filter :validate_request
 def validate request
    handle_error(request_object) unless request_object.valid?
 end
 def request_object
   @request_object ||= request_class.new(request_parameters)
 end
 def request class
    "#{action_name}#{resource_name}Request".constantize
 end
 def handle_error(request_object)
    [...]
 end
end
```

# Service Object



# Service Object





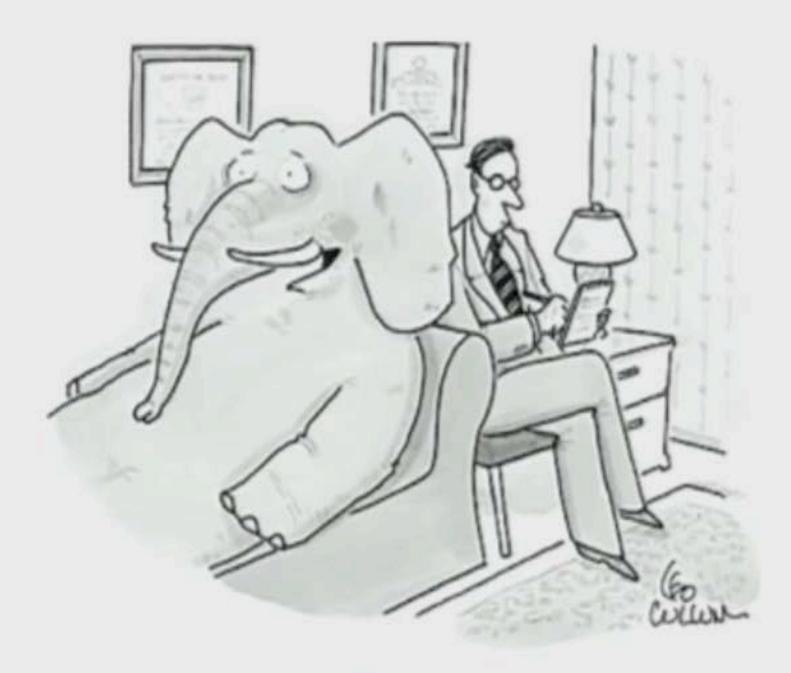


```
class OrderService
  def create(order)
    authorize!(order)
    repository.save!(order)
    purchase(order) do |transaction|
      repository.save!(order)
    end
  end
  def purchase(order, &block)
    PaymentService.new.purchase(order, &block)
  end
end
```





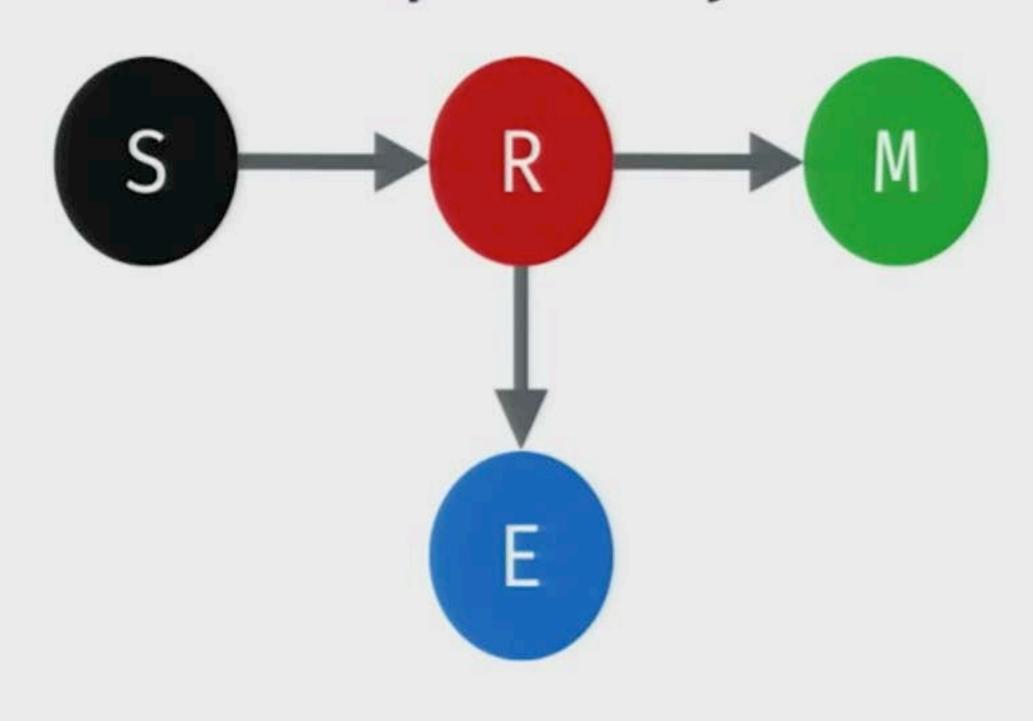
### OK, so now what?



"I'm right there in the room, and no one even acknowledges me."

# Application Domain

# Repository



```
class Repository
  class << self
    attr accessor :mapper
    def save! (domain)
      record = mapper.export(domain, record)
      response = record.save!
      domain.id = record.id
     response
    end
  end
  def self.method missing(method name, *args, &block)
    Scope.new(mapper).send(method name, *args, &block)
  end
end
```

```
class Scope
  attr accessor imapper, iscope
  def initialize(mapper)
   #mapper = mapper
   #scope = mapper.export class
  def method missing(method name, *args, &block)
   #scope = scope.send(method name, *args, & map block(block))
   scope, is a?(ActiveRecord::Relation) ? self : map(scope)
  def map block(block)
   Proc.new ( | *args | block.call( map(*args)) ) if block
  def map(object)
   if object.is a?(mapper.export class)
     mapper.map(object)
   elsif object.is a?(Enumerable)
     object.map { | e | e.is_a?(mapper.export_class) / mapper.map(e) : e }
   else
     object
  def respond to?(method_name, include_private = false)
    scope.respond to?(method name, include private) | super
```

```
class IdentityMap
 class << self
   def add(record)
      raise ArgumentError.new('Record cannot be added with a nil id') unless record.id
      repository[key(record.class)][record.id] = record
   def remove(record)
      repository[key(record.class)].delete(record.id)
   def get(klass, id)
     repository[key(klass)][id]
   def clear
      repository.clear
   def repository
     Thread.current[:identity_map] |= Hash.new { |h,k| h[k] = {} }
   def key(klass)
      klass
 end
```

So, what's the point again?

1. Embrace complexity

2. Know where you're going

3. Be more than just a "Rails Developer"





### Continue the discussion

https://github.com/dwhelan/hex-ddd

Declan Whelan @dwhelan



Eric Roberts

@eroberts



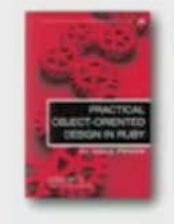
# Reading



Patterns of Enterprise Application Architecture Martin Fowler



Domain Driven Design Tackling Complexity in the Heart of Software Eric Evans



Practical Object Oriented Design in Ruby An Agile Primer Sandi Metz

