

# Domain Driven Design Hexagonal Architecture Rails

**Declan Whelan**

@dwhelan



**Eric Roberts**

@eroberts



**boltmade**

# 1. Embrace complexity



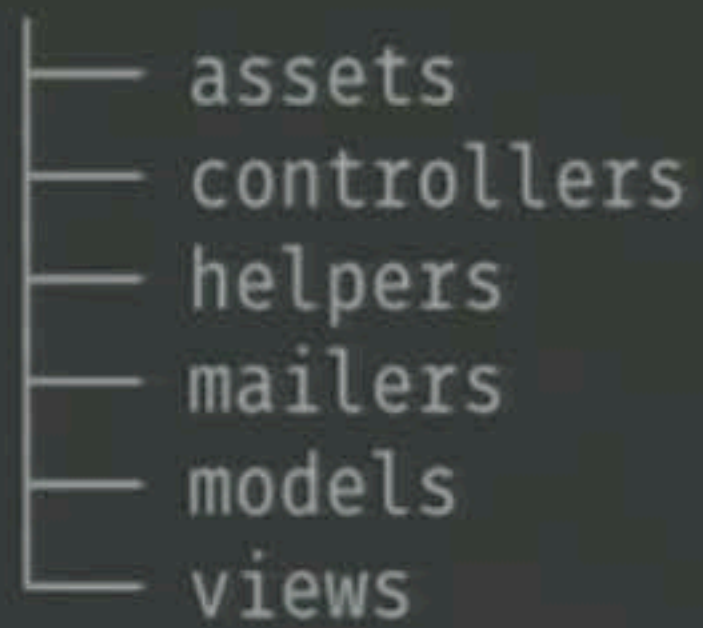
## 2. Know where you're going



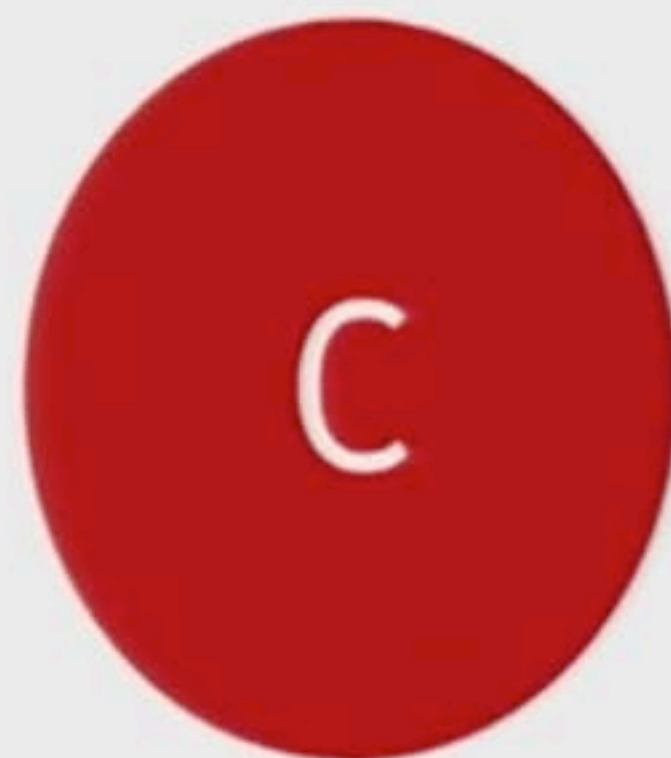
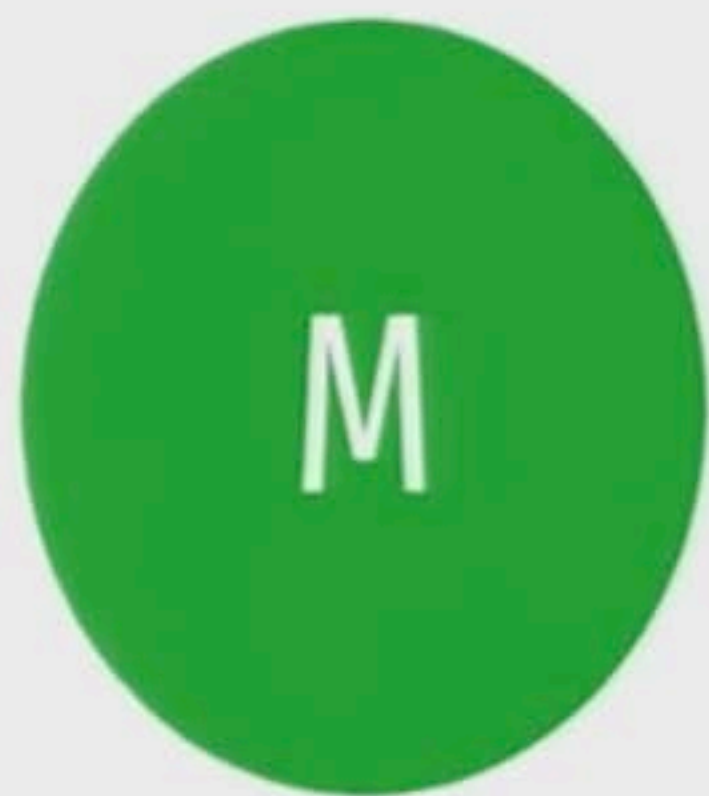
3. Be more than just a  
“Rails Developer”



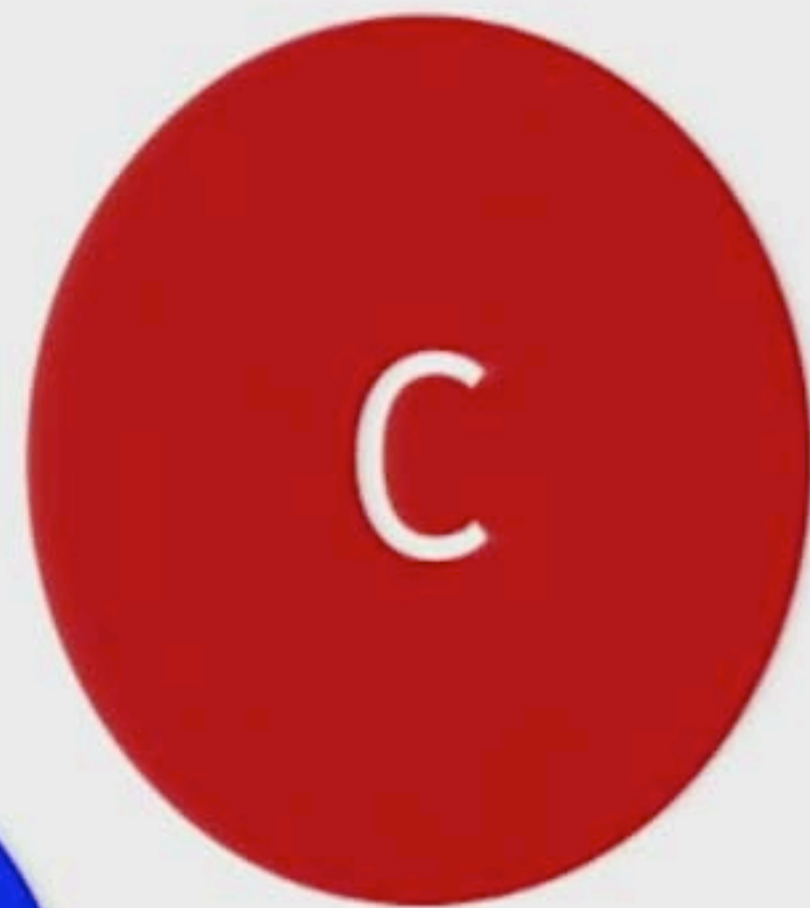
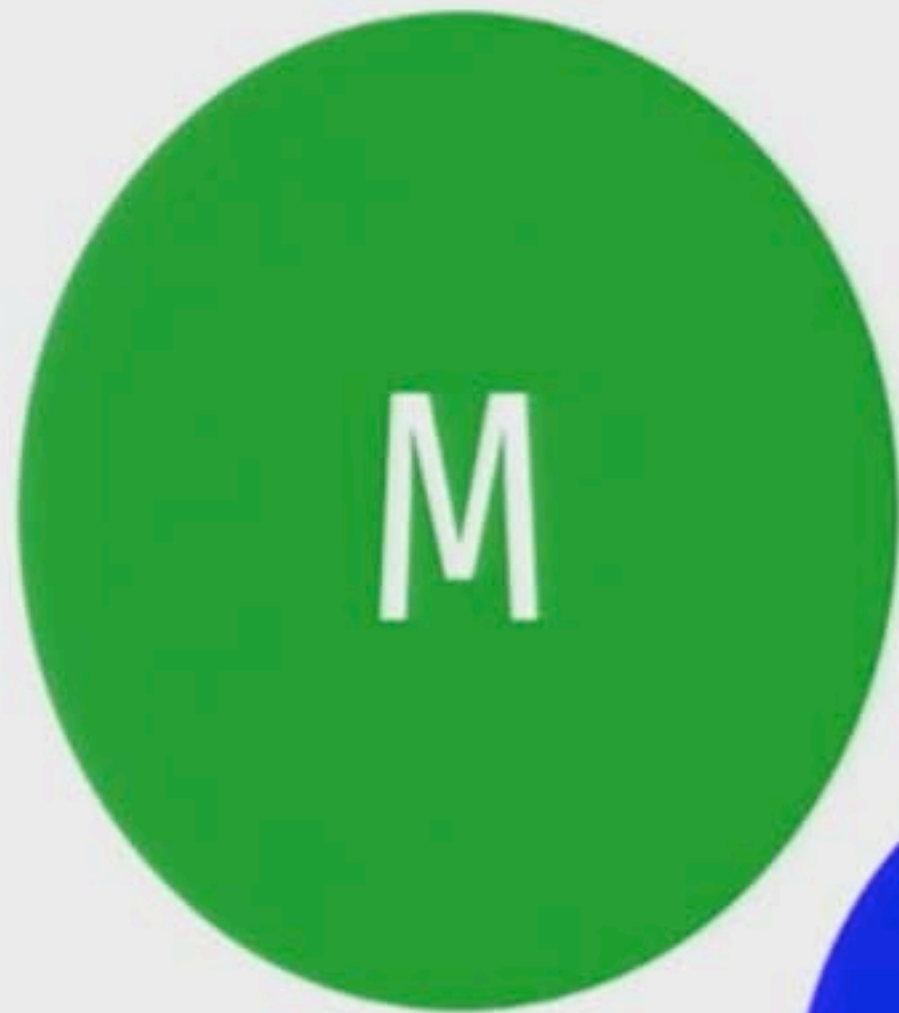
app





















# 7 Patterns to Refactor Fat ActiveRecord Models

Posted by [@primary](#) on Oct 17th, 2012

[Tweet](#) 1,237

When teams use [Code Climate](#) 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 [Single Responsibility Principle](#) (SRP). “Anything related to what a user does” is not a *single* 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 its *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 pour the batter in. Refactor to break them down and spread out the logic evenly. Repeat this

## Code Climate Blog

Thoughts about security, refactoring, object oriented design, test-driven development and design patterns with a focus on Ruby.

[@codeclimate](#)  
[RSS](#)

Code Climate helps you build better software by providing automated code review for Ruby and Javascript.

[Learn More](#)

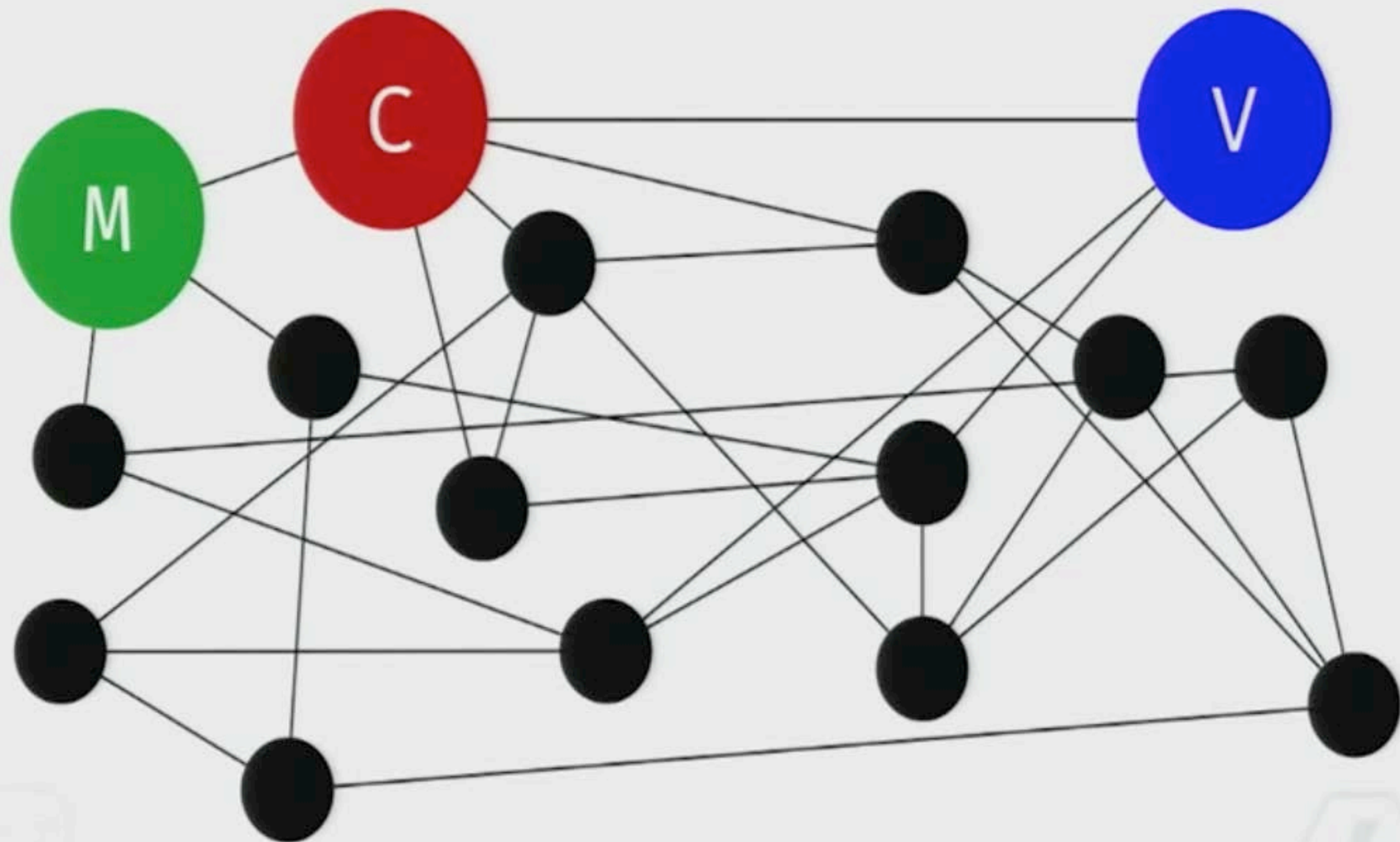
## Popular Posts

[7 Patterns to Refactor Fat ActiveRecord Models](#)

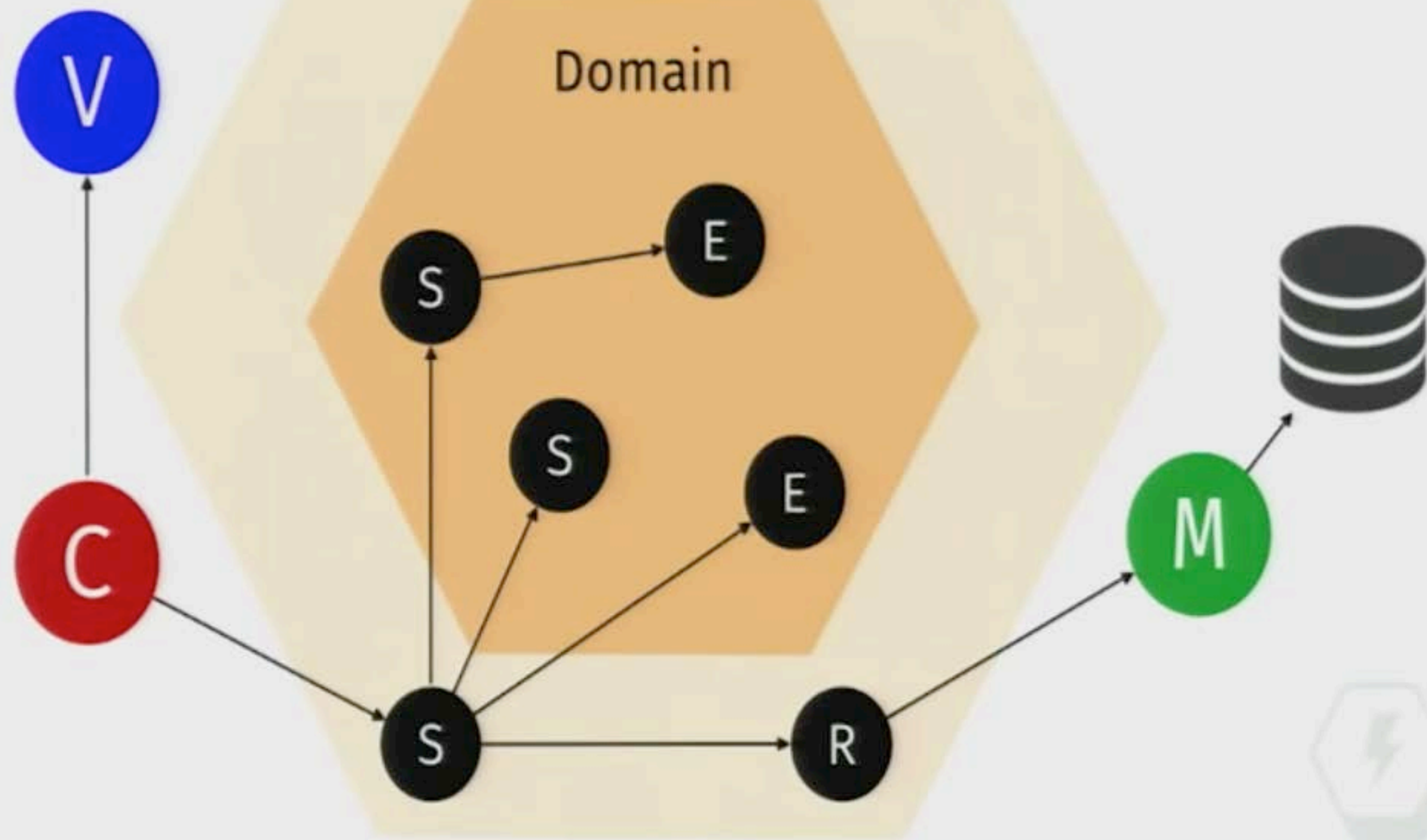
[Rails' Insecure Defaults](#)

[Your Objects, the Unix Way](#)

[Sublime Text 2 for Ruby](#)







Domain-Driven

DESIGN

Tackling Complexity in the Heart of Software



Eric Evans

Foreword by Martin Fowler



# Complexity

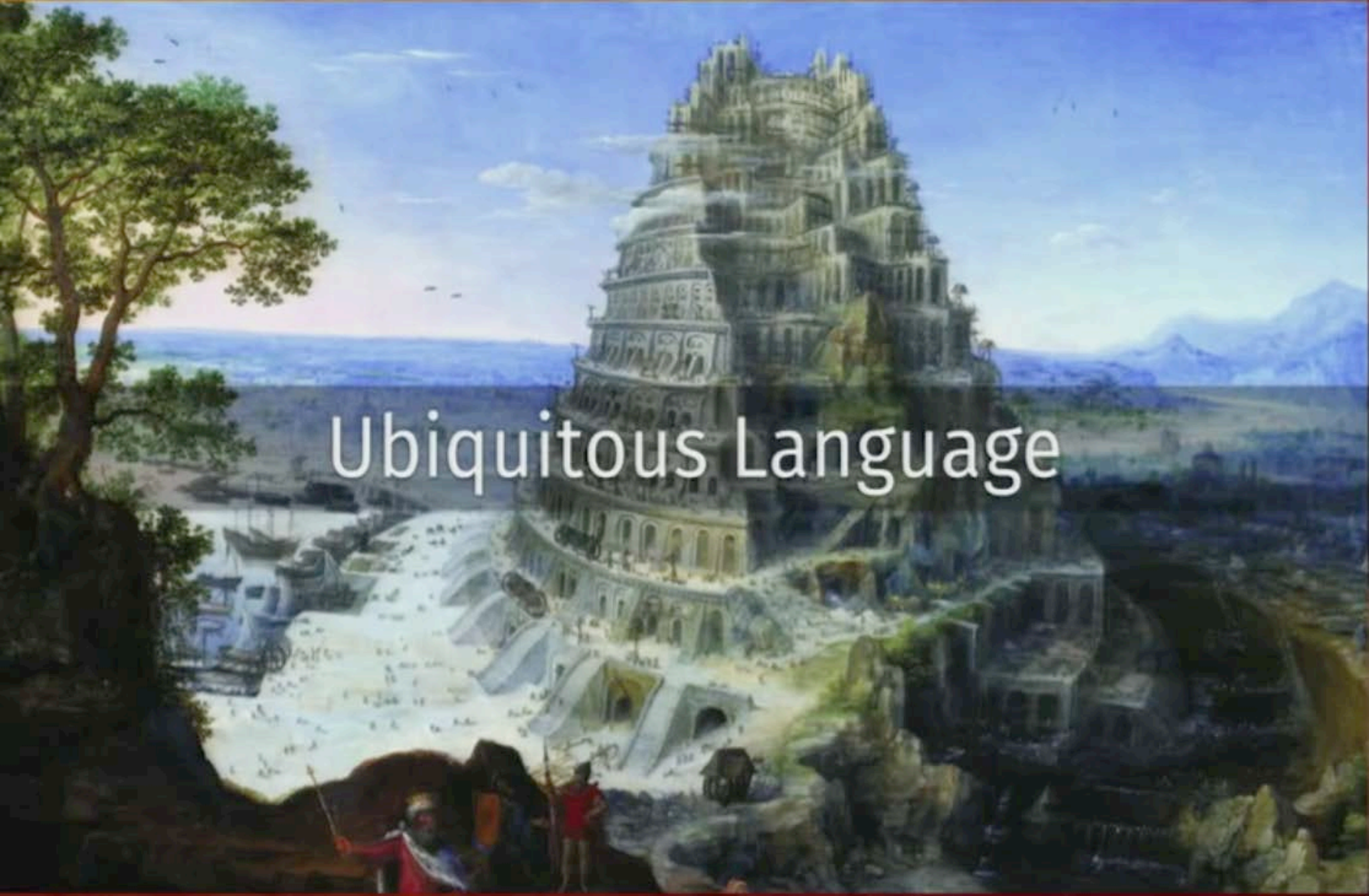


*The critical complexity of most software projects is in understanding the domain itself.*

Eric Evans



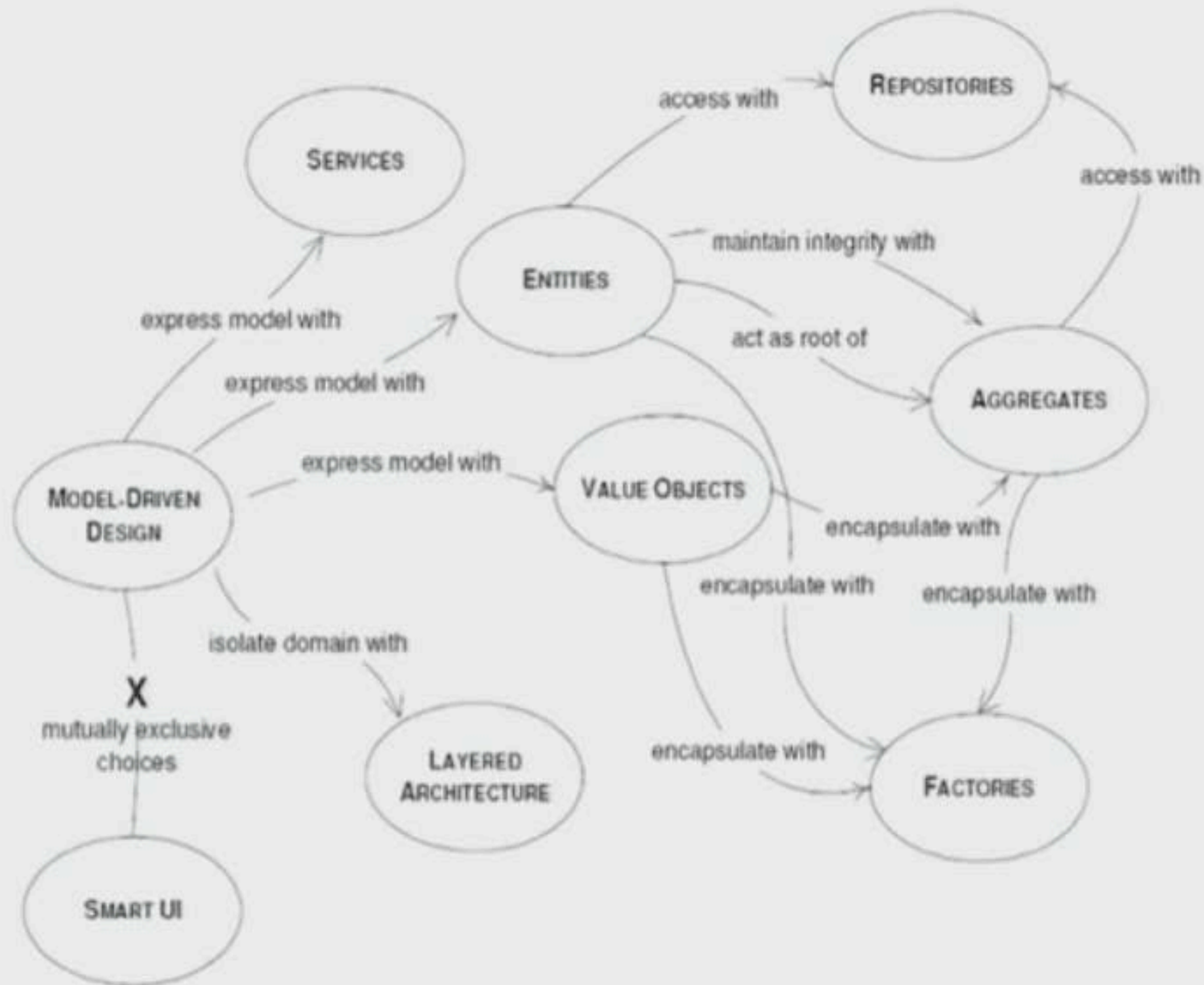




# Ubiquitous Language

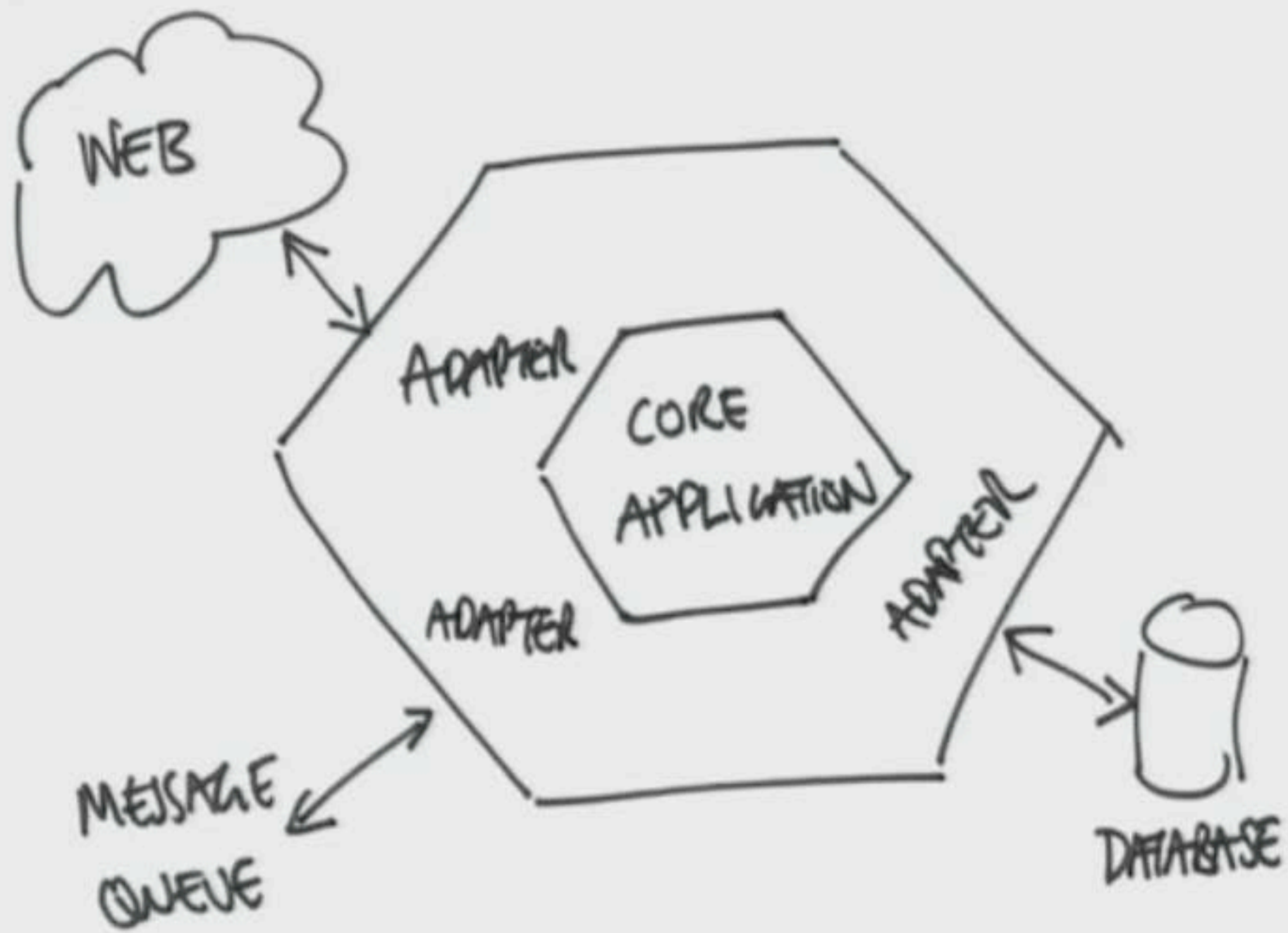


# Domain Driven Design



# Hexagonal Architecture





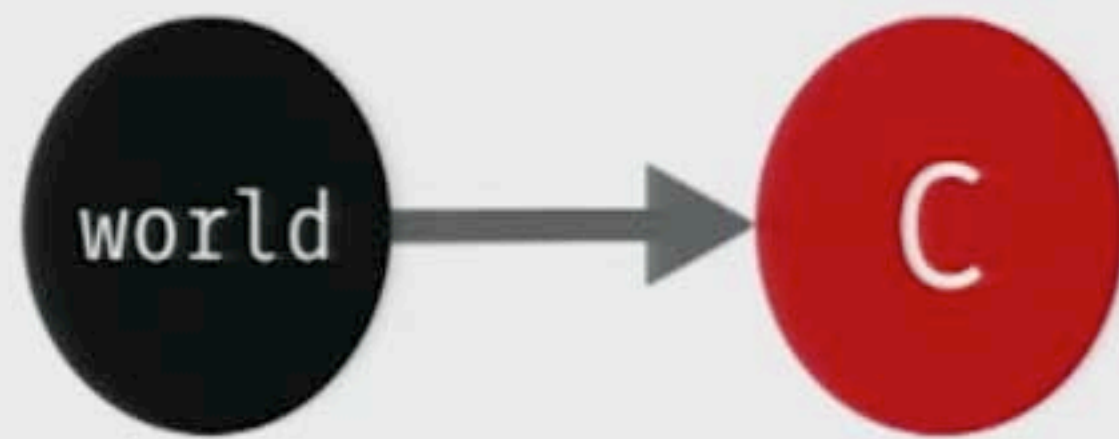
Matt Wynne

<http://blog.mattwynne.net/2012/05/31/hexagonal-rails-objects-values-and-hexagons/>

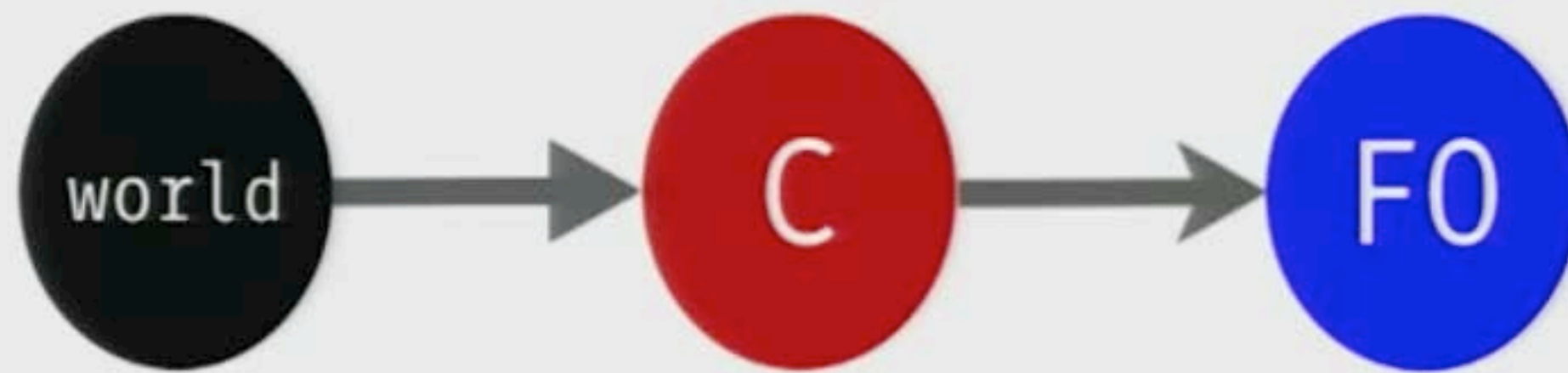




# Form Object



# 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 9th, through to closing of the event 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

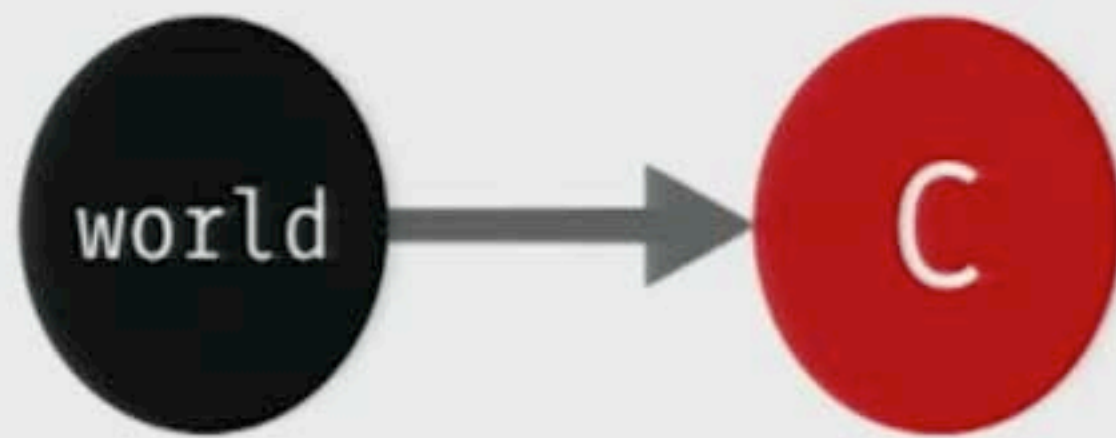
[Add a passenger for \\$847.50](#)

**Pay \$847.50**





# 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

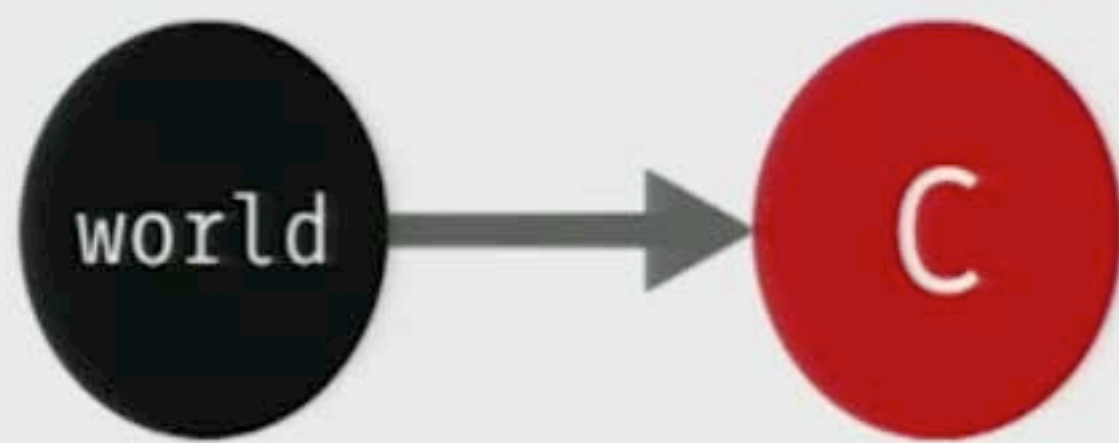
end
```







# 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
```

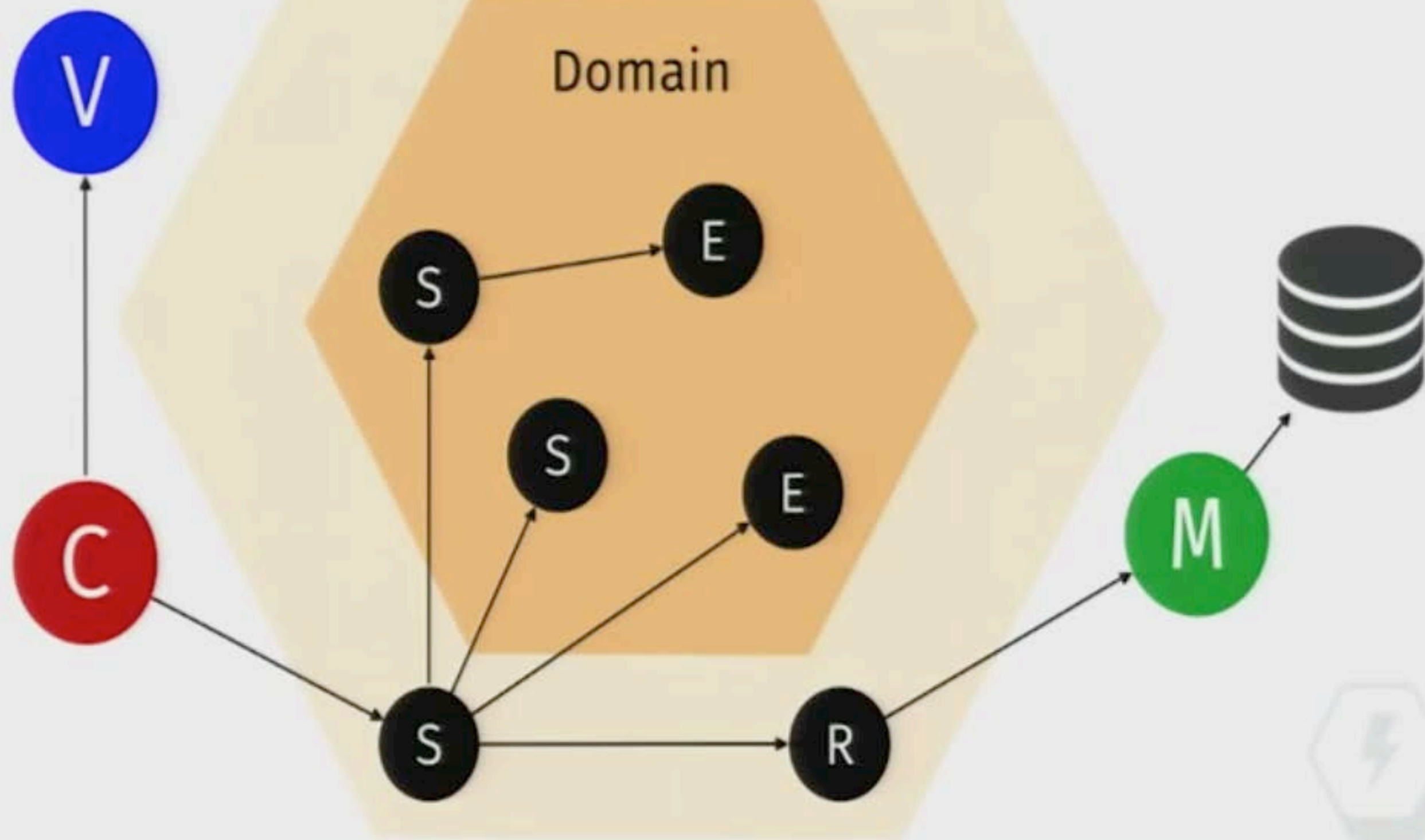


OK, so now what?

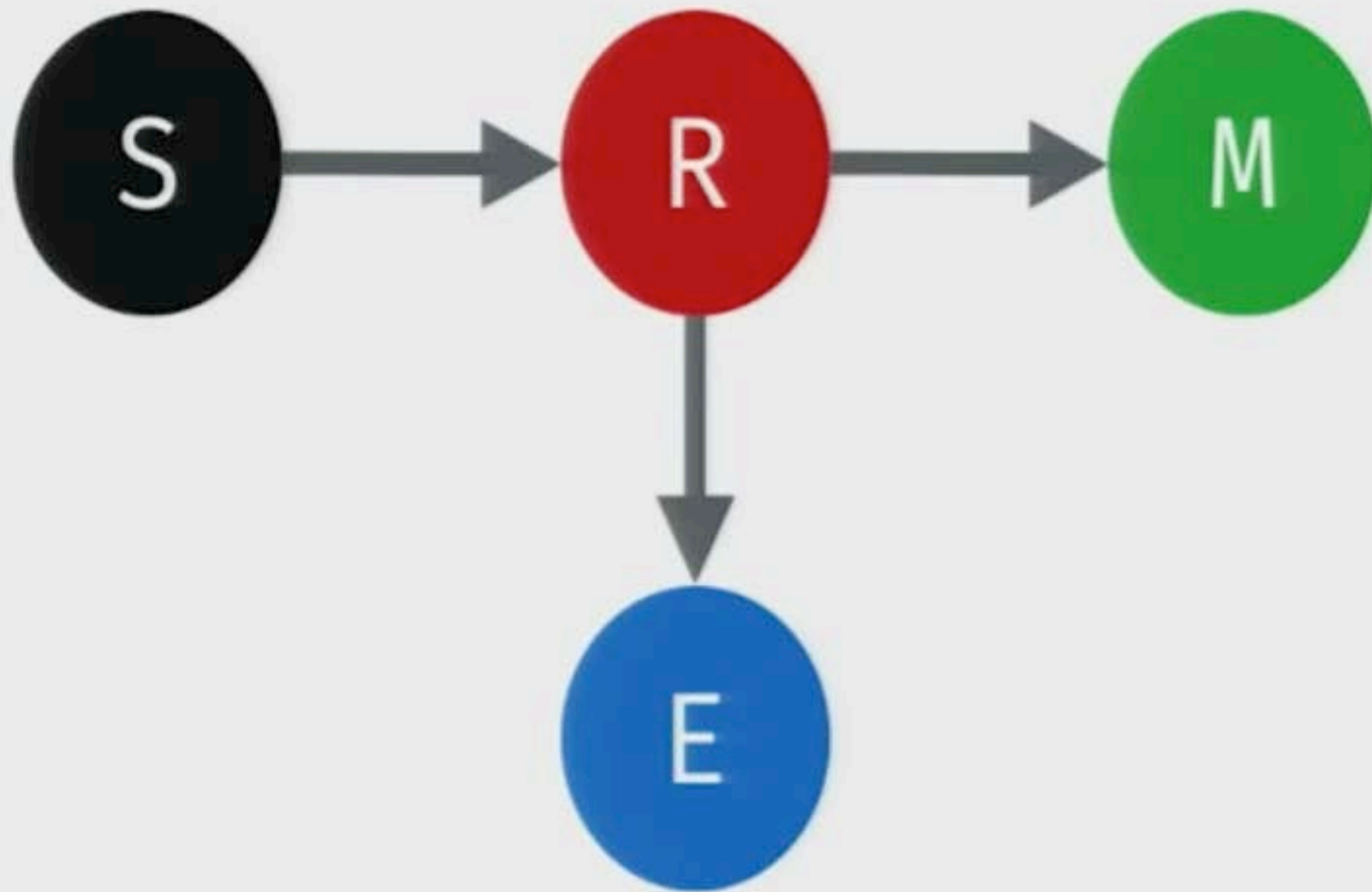


“I’m right there in the room, and no one even acknowledges me.”





# 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
```





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



**boltmade**

# 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

