I ♥ Complexity

Adam Keys

http://therealadam.com

Manifesto

The first large Rails app I built was in search. Search is relatively straight-forward. Queries in, results out. Eventually I found it quite boring.

When I got into search, I thought I was in the perfect domain. Dealing with money, regulations and people must be the root of all complexity. Search has none of them, so I could operate on pure abstract goodness!

As I said earlier, I found this ascetic existence lacking.

So I started looking for ideas that make complex problems more tractable.

My last non-Rails job and my current job both deal with somewhat involved systems. There's lots to know, both about the problem and to understand the solution. I find this exciting.

All the simple ideas are done.

Let's try something harder.

However, let's add complexity carefully. The goal is to maximize the potential while minimizing mental effort.

We're going to talk about some concepts that can help us make larger, more involved apps. They're all easy to understand in isolation. They also compliment each other nicely.

The concepts are domain modeling, stateful logic, first-class currency, time travel and asynchronous processing.

Let's dive in.

At the beginning of a project, we speak in different terms than our customers. They may speak in terms of accounts, debits and credits while we speak in terms of users, additions and subtractions.

The crux of domain domain driven design is creating one jargon. You speak to the customer using terms that appear as actual objects and methods in your software.

Domain Modeling

The ubiquitous language is an abstract goal of domain driven design. You iterate on a vocabulary that allows you to speak in terms your customer understands.

As you iterate on this language, the models and objects in your system mature. Over time, the system becomes a better reflection of the problem you're trying to solve.

In this way, you need to maintain fewer documents and its easier to look at the system and describe what it does for the user.

Ubiquitous Language

Entities, Values, Services

Entities are like ActiveRecord objects. They are unique and special. They have behavior and interact with the rest of the system to accomplish interesting things.

Value objects are often the glue between entities. They are transient and non-unique. Instead, they promote self-describing classes to first-class objects, like Money.

Services make things happen. They often operate on aggregation of entities. Other times, they handle moving entities between unrelated subsystems.

We take the ubiquitous language and we map that into our models.

We use entities, values and services to express what's going on through naming objects and methods based on their intent, rather than their mechanics.

In doing so, we've represented the pure, abstract essence of the application. This way we can tackle larger problems where dealing with ceremony and complexity may have thwarted us.

Domain + Intention = Essence

Anemic domain

```
create_table :orders do |t|
  t.string :customer_name
  t.string :customer_address
  t.float :amount
end

create_table :line_items do |t|
  t.references :order
  t.references :product
end

create_table :products do |t|
  t.string :name
  t.float :price
end
```

This isn't far from a typical starting point. No shame in that, this is a great place to start, for example, if you're using scaffolding to get down to details with your customer.

That said, if we're serious about orders and products, we don't want to stay this way for too long.

Anemic domain

The problems will arise because our intention is not apparent from these model classes.

```
class Order < ActiveRecord::Base
  has_many :line_items
  has_many :products, :through => :line_items
end

class Product < ActiveRecord::Base
end
```

Anemic domain

The first bit here is a little ugly. Its also doesn't make for a nice REST model.

```
assert_equal @order,
             Order.find_by_customer_name(
              'Ulysses Arthur')
```

The second bit shows where our anemic domain starts to fall down. We have to update the value of the order any time a product is added. Which we will inevitably forget.

Anemic domain

The first bit here is a little ugly. Its also doesn't make for a nice REST model.

```
assert_equal @order,
             Order.find_by_customer_name(
              'Ulysses Arthur')
products = [Product.create(:name => 'Gizmo',
                           :price => '1.23'),
            Product.create(:name => 'Frobber',
                           :price => '2.34')]
@order.products = products
assert_equal @order.amount, 3.57
```

The second bit shows where our anemic domain starts to fall down. We have to update the value of the order any time a product is added. Which we will inevitably forget.

Strong domain

```
create_table :customers, :force => true do |t|
  t.string :name
  t.string :currency
  t.string :address
end

create_table :orders, :force => true do |t|
  t.references :customer
end

create_table :line_items, :force => true do |t|
  t.references :product
  t.references :order
  t.timestamps
end
```

Strong domain

```
create_table :customers, :force => true do |t|
  t.string :name
  t.string :currency
  t.string :address
end

create_table :orders, :force => true do |t|
  t.references :customer
end

create_table :line_items, :force => true do |t|
  t.references :product
  t.references :order
  t.timestamps
end
```

Strong domain

```
class Order < ActiveRecord::Base
  def amount
    sum = products.inject(0.to_money) do |sum, product|
      sum += product.price
    end

    if sum.currency == customer.currency
      sum
    else
      sum.exchange_to(customer.currency)
    end
  end
end
```

Computers are inherently stateful. Its what makes them useful. Despite the efforts of REST and functional languages to push state to the side, state is the most important part of our systems.

We’ve spent a lot of mental effort trying to isolate state. Too much state is hard to understand and becomes unwieldy. But discarding it leaves us equally complex, unwieldy structures.

The happy middle is a way to encode state within our entities. That delineation in hand, we can then describe common and unique behavior for each state.

Stateful Logic

Dog rescue

```
create_table :dogs, :force => true do |t|
  t.string :name
  t.integer :age
  t.integer :at_vet, :at_foster,
            :at_hospice, :at_forever_home
  t.timestamps
end
```

Dog rescue

```
create_table :dogs, :force => true do |t|
  t.string :name
  t.integer :age
  t.integer :at_vet, :at_foster,
            :at_hospice, :at_forever_home
  t.timestamps
end
```

Dog rescue

```
class Dog < ActiveRecord::Base

  has_many :vettings
  belongs_to :foster_parent
  belongs_to :hospice_provider
  has_one :adoptive_parent
```

Dog rescue

```
@dog = Dog.new(:name => 'Cooper', :age => 2)
vetting = Vetting.new(:heartworms => false,
                      :fixed => true)
@dog.vettings << vetting
@dog.at_vet = true
```

Dog rescue

```
@dog = Dog.new(:name => 'Cooper', :age => 2)
vetting = Vetting.new(:heartworms => false,
                      :fixed => true)
@dog.vettings << vetting
@dog.at_vet = true
```

Dog rescue

```
def rescued?
  !at_vet? && !at_foster? &&
  !at_hospice? && !adopted?
end

def vetted?
  at_vet? && !at_foster? &&
  !at_hospice? && !adopted?
end
```

Dog rescue

```
def rescued?
  !at_vet? && !at_foster? &&
  !at_hospice? && !adopted?
end

def vetted?
  at_vet? && !at_foster? &&
  !at_hospice? && !adopted?
end
```

Dog rescue

```
def rescued?
  !at_vet? && !at_foster? &&
  !at_hospice? && !adopted?
end

def vetted?
  at_vet? && !at_foster? &&
  !at_hospice? && !adopted?
end
```

Where's the bug??!

Dog rescue

```
def rescued?
  !at_vet? && !at_foster? &&
  !at_hospice? && !adopted?
end

def vetted?
  at_vet? && !at_foster? &&
  !at_hospice? && !adopted?
end
```

Where’s the bug??!

What do these states **mean**?

Dog Rescue, with state

```
class Dog < ActiveRecord::Base

  has_many :vettings
  belongs_to :foster_parent
  belongs_to :hospice_provider
  has_one :adoptive_parent

  include AASM

  aasm_initial_state :sheltered
```

Dog Rescue, with state

```
class Dog < ActiveRecord::Base

  has_many :vettings
  belongs_to :foster_parent
  belongs_to :hospice_provider
  has_one :adoptive_parent

  include AASM

  aasm_initial_state :sheltered
```

Dog Rescue, with state

```
create_table :dogs, :force => true do |t|
  t.string :name
  t.integer :age
  t.string :aasm_state
  t.timestamps
end
```

Dog Rescue, with state

```
create_table :dogs, :force => true do |t|
  t.string :name
  t.integer :age
  t.string :aasm_state
  t.timestamps
end
```

Dog Rescue, with state

```
aasm_state :sheltered
aasm_state :rescued
aasm_state :vetted
aasm_state :fostered
aasm_state :hospiced
aasm_state :adopted
```

Dog Rescue, with state

```
aasm_event :rescue do
  transitions :to => :rescued,
              :from => [:sheltered]
end

aasm_event :vet do
  transitions :to => :vetted,
              :from => [:rescued, :fostered],
              :guard => lambda { |dog|
                dog.vettings.length > 0
              }
end
```

Dog Rescue, with state

```
aasm_event :rescue do
  transitions :to => :rescued,
              :from => [:sheltered]
end

aasm_event :vet do
  transitions :to => :vetted,
              :from => [:rescued, :fostered],
              :guard => lambda { |dog|
                dog.vettings.length > 0
              }
end
```

Dog Rescue, with state

```
@dog = Dog.new(:name => 'Cooper', :age => 2)
@dog.rescue
@dog.vettings << Vetting.new(:heartworms => false,
                             :fixed => true)
@dog.vet
```

Dog Rescue, with state

```
@dog = Dog.new(:name => 'Cooper', :age => 2)
@dog.rescue
@dog.vettings << Vetting.new(:heartworms => false,
                             :fixed => true)
@dog.vet
```

Dog Rescue, with state

```
@dog = Dog.new(:name => 'Cooper', :age => 2)
@dog.rescue
@dog.vettings << Vetting.new(:heartworms => false,
                             :fixed => true)
@dog.vet
```

Monies

Its true what they say about money.

Not that its the root of all evil. That's false.

More money, more problems - it's true!

The trouble with money and software is that otherwise rational and kind people will argue with you about money down to the fraction of a fraction of a cent.

Further, programming language designers are way too cool to include money in their standard libraries. It seems reasonable one could just use floating point numbers for this.

Therein lies the rub. Floating point numbers are imprecise in devilish ways. So we need to promote money to a first-class object.

Survey: who has an application that doesn't deal with currency?

The must-have value object for every application!

Troubled money

This won't fly for two reasons.

* Floats will bite us in the weirdest possible ways
* Using a column to store the total is not particularly friendly

```
create_table :orders do |t|
  t.string :customer_name
  t.string :customer_address
  t.float :amount
end
```

Troubled money

This won't fly for two reasons.

* Floats will bite us in the weirdest possible ways
* Using a column to store the total is not particularly friendly

```
create_table :orders do |t|
  t.string :customer_name
  t.string :customer_address
  t.float :amount
end
```

Better money

Here's a better way to represent money. We're going to use Tobias Luetke's money gem. It gives us a Money value object that represents money as cents and the currency.

```
create_table :products, :force => true do |t|
  t.string :name
  t.integer :cents, :default => 0
  t.string :currency, :default => 'USD'
end
```

Better money

Here's a better way to represent money. We're going to use Tobias Luetke's money gem. It gives us a Money value object that represents money as cents and the currency.

```
create_table :products, :force => true do |t|
  t.string :name
  t.integer :cents, :default => 0
  t.string :currency, :default => 'USD'
end
```

Better money

```
class Product < ActiveRecord::Base
  validates_presence_of :name
  composed_of :price,
              :class_name => 'Money',
              :mapping => [%w(cents cents),
                           %w(currency currency)]

  validate :price_greater_than_zero

  def price_greater_than_zero
    unless cents > 0
      errors.add('cents',
                 'cannot be less than zero')
    end
  end
end
```

And here's our model. Per the Money gem readme, we use composed_of to put a pseudo-attribute on our class that maps to the cents and currency fields of the Money class.

Better money

```
class Product < ActiveRecord::Base
  validates_presence_of :name
  composed_of :price,
              :class_name => 'Money',
              :mapping => [%w(cents cents),
                           %w(currency currency)]

  validate :price_greater_than_zero

  def price_greater_than_zero
    unless cents > 0
      errors.add('cents',
                 'cannot be less than zero')
    end
  end
end
```

And here's our model. Per the Money gem readme, we use composed_of to put a pseudo-attribute on our class that maps to the cents and currency fields of the Money class.

Better money

```
class Order < ActiveRecord::Base
  def amount
    # Note that products.inject(0)
    # won't work because its _not_ Money.
    sum = products.inject(0.to_money) do |sum, p|
      sum += p.price
    end

    if sum.currency == customer.currency
      sum
    else
      sum.exchange_to(customer.currency)
    end
  end
```

So now we can implement the amount message as a real method and just sum it up.

Also note that we can convert between different currencies (which is stored, obviously, on the customer model).

Exchanging

So somewhere in your app, you'll want to set up an exchange like so.

In reality, you'll probably want to fetch these numbers from a real exchange web service.

```
def setup_exchanges!
  Money.bank = VariableExchangeBank.new
  Money.bank.add_rate('USD', 'EUR', 0.67648)
  Money.bank.add_rate('EUR', 'USD', 1.47823)
end
```

Exchanging

With all that set up, you can painlessly sum up orders and then convert it back to another currency at your whimsy.

As far as I can tell, calculations occur in the first currency used.

```
@frobulator = Product.create(
  :name => 'Frobulator (US)',
  :price => Money.us_dollar(10))
@grokulator = Product.create(
  :name => 'Grokulator (EU)',
  :price => Money.euro(100))

@order.products = [@frobulator, @grokulator]
@order.amount
```

I worked on an application that was fundamentally built on top of time travel. Almost every entity in the system could go forward or backward in time.

Initially, I was completely terrified. Certainly this was ceremonial complexity and not essential complexity.

Then I found Martin Fowler's writings on the topic and I felt much better. Turns out lots of people need to move through the space/time contiuum effortlessly.

It also turns out that time travel needn't be that difficult. The important part is to figure what really matters to you. Time travel is a Bohemian existence, it turns out.

Time Travel

Versioned Products

Luckily Rick Olson has our back here. acts_as_versioned is a simple way to add time travel to your application.

First, we add a version column to our model.

```
create_table :products, :force => true do |t|
  t.string :name
  t.text :description
  t.integer :cents, :default => 0
  t.string :currency, :default => 'USD'
  t.integer :version, :null => false
end
```

Versioned Products

Luckily Rick Olson has our back here. acts_as_versioned is a simple way to add time travel to your application.

First, we add a version column to our model.

```
create_table :products, :force => true do |t|
  t.string :name
  t.text :description
  t.integer :cents, :default => 0
  t.string :currency, :default => 'USD'
  t.integer :version, :null => false
end
```

Versioned Products

Next, we need to generate a table that stores all the previous versions of our class.

So we'll create a Product class in our migration and then call created_versioned_table on it.

```
# In our migration...
class Product < ActiveRecord::Base
  acts_as_versioned
end

Product.create_versioned_table
```

Versioned Products

Next, we need to generate a table that stores all the previous versions of our class.

So we'll create a Product class in our migration and then call created_versioned_table on it.

```
# In our migration...
class Product < ActiveRecord::Base
  acts_as_versioned
end

Product.create_versioned_table
```

Versioned Products

Then, in our real model (not the migration one) we call the acts_as_versioned method again.

And that's it.

```
class Product < ActiveRecord::Base
  acts_as_versioned
```

Versioned Products

Now we can do cool stuff like get at the previous version of a model attribute or set the current version of a model.

```
@product = Product.create(
  :name => 'iPhone',
  :description => 'The phone with web apps!',
  :price => Money.new(599.99, 'USD'))

@product.description =
  'The phone with native apps!'
@product.save!

previous_version =
  @product.versions.latest.previous
assert_equal 'The phone with web apps!',
  previous_version.description
```

acts_as_versioned is a solid foundation for your time-travel needs.

That said, as with most time machines, a little customization is needed to suit. Don't fear peaking behind the curtain.

Versioned Products

Now we can do cool stuff like get at the previous version of a model attribute or set the current version of a model.

```
@product = Product.create(
  :name => 'iPhone',
  :description => 'The phone with web apps!',
  :price => Money.new(599.99, 'USD'))

@product.description =
  'The phone with native apps!'
@product.save!

previous_version =
  @product.versions.latest.previous
assert_equal 'The phone with web apps!',
  previous_version.description
```

acts_as_versioned is a solid foundation for your time-travel needs.

That said, as with most time machines, a little customization is needed to suit. Don’t fear peaking behind the curtain.

Most of us live in a transactional world. A user requests a page, data or change. We labor to produce it as quickly as possible. The fat is trimmed as much as possible in the name of transaction rates.

However, there are lots of interesting things we can do that take more than a few seconds. Further, most people realize the value of these things and just want to know when to refresh their page.

Right now, spinning off threads isn't the best of ideas in most Rails apps. So, we're left with queues. Which is fine, because queues are neat.

At their simplest, queues just say "hey, do this for me when you get the chance." But if we promote the queue to a first-class member of our domain model, we get neat things.

Asynchronous Processing

Pretty cool service, but its part of your domain too!

Moderating through time

```
create_table :moderations, :force => true do |t|
  t.references :product
  t.string :aasm_state, :null => false
  t.integer :version
  t.timestamps
end
```

So let's suppose that we discover that, in reality, products go through a sort of editorial process.

We need to provide a verification that a product's title, description and price have been vetted.

Moderating through time

```
create_table :products, :force => true do |t|
  t.string :name
  t.text :description
  t.integer :cents, :default => 0
  t.string :currency, :default => 'USD'
  t.integer :version, :null => false
  t.integer :display_version, :default => 0
end
```

However, we don't want to show changes to products that have been edited immediately. Those changes need vetting too.

So we need to customize our time machine

Moderating through time

```
create_table :products, :force => true do |t|
  t.string :name
  t.text :description
  t.integer :cents, :default => 0
  t.string :currency, :default => 'USD'
  t.integer :version, :null => false
  t.integer :display_version, :default => 0
end
```

However, we don't want to show changes to products that have been edited immediately. Those changes need vetting too.

So we need to customize our time machine

Moderating through time

```
class Product < ActiveRecord::Base
  has_many :moderations do
    def current
      last
    end
  end

  after_save :create_moderation_entry

  def display?
    display_version > 0
  end

  private

    def create_moderation_entry
      moderations.create!(:version => version) if save_version?
    end
```

Here again is our Product class. We've add all the moderations for this product, including an accessor for the most recent moderation.

Next we've got a callback that will create a new moderation entry every time out product is updated.

In this way, we get a queue built into our application.

The other bit worth noting is that we've got this display? flag that indicates whether a product should be shown. This hides new products that have yet to be vetted.

Moderating through time

```
class Product < ActiveRecord::Base
  has_many :moderations do
    def current
      last
    end
  end

  after_save :create_moderation_entry

  def display?
    display_version > 0
  end

  private

    def create_moderation_entry
      moderations.create!(:version => version) if save_version?
    end
```

Here again is our Product class. We've add all the moderations for this product, including an accessor for the most recent moderation.

Next we've got a callback that will create a new moderation entry every time out product is updated.

In this way, we get a queue built into our application.

The other bit worth noting is that we've got this display? flag that indicates whether a product should be shown. This hides new products that have yet to be vetted.

Moderating through time

```
class Product < ActiveRecord::Base
  has_many :moderations do
    def current
      last
    end
  end

  after_save :create_moderation_entry

  def display?
    display_version > 0
  end

  private

    def create_moderation_entry
      moderations.create!(:version => version) if save_version?
    end
```

Here again is our Product class. We've add all the moderations for this product, including an accessor for the most recent moderation.

Next we've got a callback that will create a new moderation entry every time out product is updated.

In this way, we get a queue built into our application.

The other bit worth noting is that we've got this display? flag that indicates whether a product should be shown. This hides new products that have yet to be vetted.

Moderating through time

```
class Product < ActiveRecord::Base
 has_many :moderations do
  def current
   last
  end
 end

 after_save :create_moderation_entry

 def display?
  display_version > 0
 end

 private

  def create_moderation_entry
   moderations.create!(:version => version) if save_version?
  end
```

Here again is our Product class. We've add all the moderations for this product, including an accessor for the most recent moderation.

Next we've got a callback that will create a new moderation entry every time out product is updated.

In this way, we get a queue built into our application.

The other bit worth noting is that we've got this display? flag that indicates whether a product should be shown. This hides new products that have yet to be vetted.

Moderating through time

```
class Moderation < ActiveRecord::Base
  belongs_to :product

  include AASM

  aasm_initial_state :pending

  aasm_state :pending
  aasm_state :approved
  aasm_state :rejected
```

The beginnings of our actual moderation class are straight-forward AASM bits.

The cool thing is that AASM automatically creates accessors for each state. So to get our "queue" of pending moderations, we'll just call Moderation.pending.

Moderating through time

```
class Moderation < ActiveRecord::Base
  belongs_to :product

  include AASM

  aasm_initial_state :pending

  aasm_state :pending
  aasm_state :approved
  aasm_state :rejected
```

The beginnings of our actual moderation class are straight-forward AASM bits.

The cool thing is that AASM automatically creates accessors for each state. So to get our "queue" of pending moderations, we'll just call Moderation.pending.

Moderating through time

```
aasm_event :reject do
  transitions :from => :pending,
              :to => :rejected
end

aasm_event :approve do
  transitions :from => :pending,
              :to => :approved,
              :on_transition =>
                :update_product_display_version
end

private

  def update_product_display_version
    product.display_version = version
    product.save_without_revision
  end
```

To move a moderation through our queue, we'll call accept or reject on it.

When we call accept, we'll update the display_version (our customized time machine) and then save _without_ a new revision. That way, we don't get an endless moderation loop.

Moderating through time

```
aasm_event :reject do
  transitions :from => :pending,
              :to => :rejected
end

aasm_event :approve do
  transitions :from => :pending,
              :to => :approved,
              :on_transition =>
                :update_product_display_version
end

private

  def update_product_display_version
    product.display_version = version
    product.save_without_revision
  end
```

To move a moderation through our queue, we'll call accept or reject on it.

When we call accept, we'll update the display_version (our customized time machine) and then save _without_ a new revision. That way, we don't get an endless moderation loop.

Moderating through time

```
aasm_event :reject do
  transitions :from => :pending,
              :to => :rejected
end

aasm_event :approve do
  transitions :from => :pending,
              :to => :approved,
              :on_transition =>
                :update_product_display_version
end

private

  def update_product_display_version
    product.display_version = version
    product.save_without_revision
  end
```

To move a moderation through our queue, we'll call accept or reject on it.

When we call accept, we'll update the display_version (our customized time machine) and then save _without_ a new revision. That way, we don't get an endless moderation loop.

Moderating through time

Our controller code will end up having snippets like this. Easy to write and understand later.

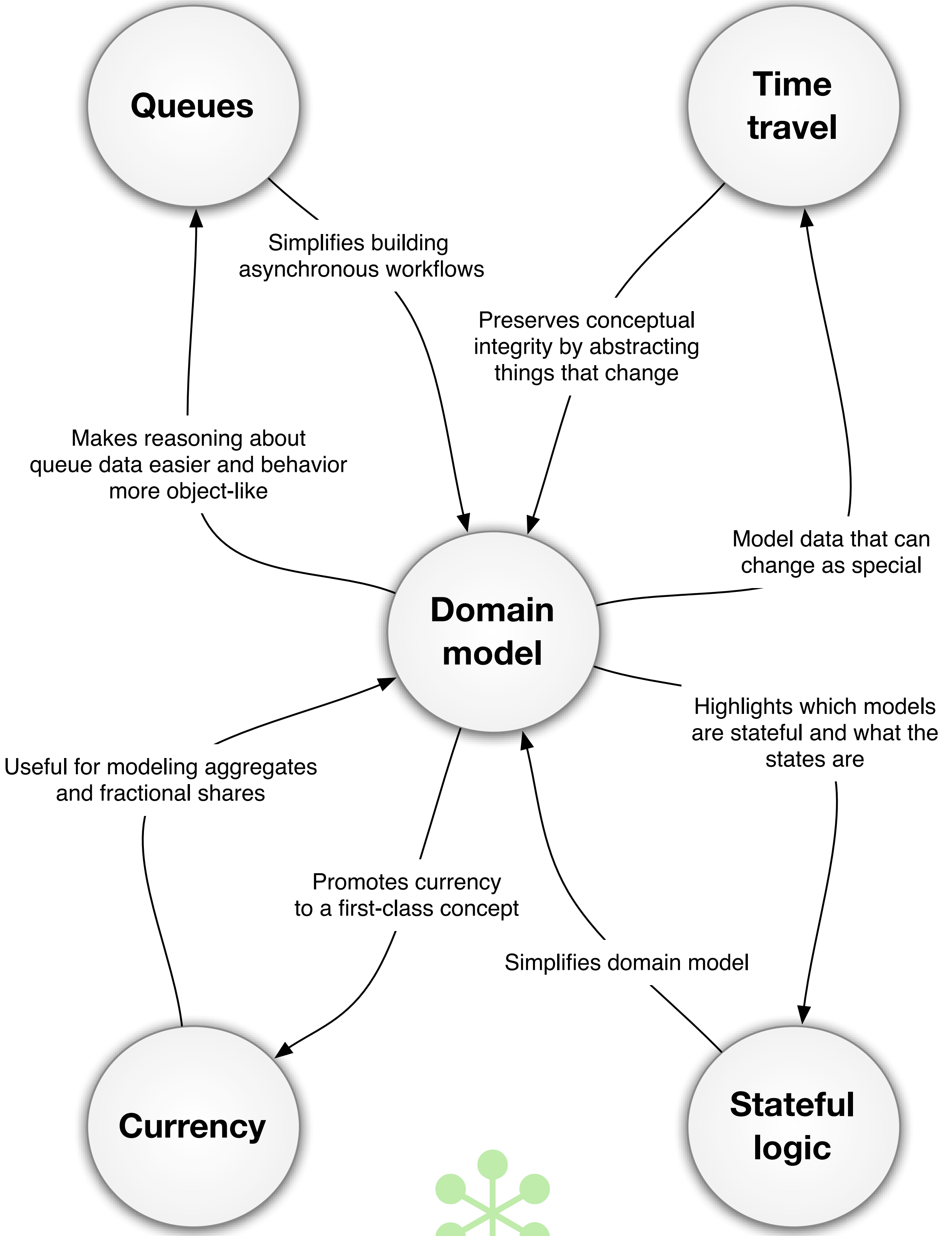
```
@product = Product.create!(
  :name => 'Boeing 777-200',
  :description => 'The wide-body with tons of leg room!',
  :price => Money.new(10_000_000_000, 'USD'))

@product.moderations.current.approve!
  @product.update_attribute(
    :description,
    'The wide-body with no leg room' +
    ' and horrible seats.')

@product.moderations.current.reject!
```

This is just one way to implement a queue in your app. Sometimes you'll want a dedicated queue like Starling or Beanstalk. But other times, putting it into your domain model makes a lot of sense.

Tackling Complexity



Queues

Time travel

Simplifies building asynchronous workflows

Preserves conceptual integrity by abstracting things that change

Makes reasoning about queue data easier and behavior more object-like

Model data that can change as special

Domain model

Highlights which models are stateful and what the states are

Useful for modeling aggregates and fractional shares

Promotes currency to a first-class concept

Simplifies domain model

Currency

Stateful logic

Thanks!

http://therealadam.com