

Common And Useful Software Learnings From a Long Living Rails Application

Derrek Long

STI: Single Table Inheritance

“STI is not used anywhere. At all. ...I have never seen a Rails app where using STI was the correct design decision...”

- www.matthewpaulmoore.com/ruby-on-rails-code-quality-checklist

```
financial_transactions
```

Field	Type	Null	Key	Default	Extra
type	varchar(128)	NO		NULL	
amount	decimal(12,2)	NO		NULL	
currency_type	varchar(3)	NO		USD	
...					

```
class FinancialTransaction < ActiveRecord::Base
  def void; end
end
```

```
class Charge < FinancialTransaction
  validates_presence_of :authorization
  def refund; end
end
```

```
class Refund < FinancialTransaction
  after_create :refund_hooks
end
```

When Can You Use STI?

- Many shared data elements
- Shared “domain”
- E.G.:
 - Charges and Refunds are both Financial Transactions, share similar data elements, and fall in the same space for which you’re modeling
 - Cars and Airplanes are both Vehicles, both have some similar data (engines, drivers, wheels), but aren’t likely in the same domain unless you’re modeling at a very high level

How About Polymorph Associations?

- Payment Methods
 - Credit Card
 - Cash
 - House Account
 - Pay Pal
 - University Card

```
payment_methods
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
user_id	int(11)	NO		NULL	
chargeable_id	int(11)	NO	MUL	NULL	
chargeable_type	varchar(255)	NO	MUL	NULL	

```
class PaymentMethod < ActiveRecord::Base
  belongs_to :chargeable, :polymorphic => true
  belongs_to :user
end
```

```
credit_cards
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
resource_id	int(11)	NO		NULL	
last_digits	varchar(6)	NO		NULL	
expires	datetime	NO		NULL	
name_on_card	varchar(255)	NO		NULL	

```
class CreditCard < ActiveRecord::Base
  has_one :payment_method, :as => :chargeable
end
```

```
pm = User.first.payment_methods.first.chargeable
```

```
pm.class # CreditCard
```

```
pm.last_digits # 1111
```

```
pm = User.first.payment_methods.last.chargeable
```

```
pm.class # HouseAccount
```

```
pm.account_number # FX67-45
```

- When can you use Polymorphic Relationships?
 - Associate the same behavior to multiple different objects
 - Group together objects of a similar domain but with different elements
- E.G:
 - `user.payment_methods`
 - `user.default_payment_method.charge(20)`
 - `brand.artist`
 - `brand.venue`

```
class PaymentMethod < ActiveRecord::Base
  belongs_to :chargeable, :polymorphic => true
  belongs_to :user

  def charge(amount)
    chargeable.charge(amount)
  end

  def credit_card?
    chargeable_type == "CreditCard"
  end
end
```

Drivers

- Ticketing System: many venues each has one of 6 ticketing system implementations
- Taxi Ordering System: each taxi company has one of 5 dispatch systems

Constantize

venues

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
name	varchar(NO		NULL	
ticketing_driver	varchar(255)	NO		NULL	

...

```
class TicketingSystem::FlyTicket
  def reserve_seat(venue, user, seat, event); end
  def available_seats(venue, event); end
end
```

```
class TicketingSystem::Paciolin
  def reserve_seat(venue, user, seat, event); end
  def available_seats(venue, event); end
end
```

```
class Venue < ActiveRecord::Base
  def ticket_system
    @system ||= ticketing_driver.constantize
  end

  def reserve_seat(user, seat, event)
    ticket_system.reserve_seat(self, user, seat, event)
  end
end
```

Time

- Imagine you are a ticketing company selling tickets on behalf of venues around the world
- Your system is in Washington DC.
- How do you figure out the current local time in Chicago?
- If you are scheduling emails or sms to be sent an hour before an events onsale how do you know when to send it?
- How do you add/subtract times

Time Zone Olson And TZinfo

- *nix systems use zoneinfo (Olson Time Zones) files
- Ruby has TZinfo which mirrors the Olson style

time_zone_olson

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
name	varchar(64)	NO	UNI	NULL	
id	name				
850176422	America/Anchorage				
8415585	America/Chicago				

TimeZoneOlson.rb

```
class TimeZoneOlson < ActiveRecord::Base
  validates_presence_of(
    :name
  )
  delegate :utc_to_local, :to => :tz

  def tz
    @tz ||= TZInfo::Timezone.get(name)
  end

  def local_to_utc(time)
    tz.local_to_utc(time, true)
  end

  def current_local_time
    utc_to_local(Time.now.utc)
  end

  def abbreviation_local(local_time)
    tz.period_for_local(local_time).zone_identifier.to_s
  end

  def abbreviation_utc(utc_time)
    tz.period_for_utc(utc_time).zone_identifier.to_s
  end
end
```

```
TimeZoneOlson.last.name # America/New_York
```

```
TimeZoneOlson.last.current_local_time # 2010-04-11 14:45:00
```

```
Fleet.last.time_zone_olson.name # America/Los_Angeles
```

```
Fleet.last.time_zone_olson.current_local_time # 2010-04-11 11:45:00
```

```
Fleet.last.time_zone_olson.local_to_utc(Time.parse("2010-04-11 14:50:00"))
```

Other Time Best Practices

- Have all your systems (physical machines) clock in UTC timezone
- Always store UTC time in the database to record “when something happened”
- Optionally store the local time. But don't compute/compare from it. Useful for showing ‘events’ on a calendar or for reports

When Not To Be DRY!

- Testing - if a change to one piece of code will force testing on many many clients
- Don't have control over release schedule of system that depends on the code (iphone app)

- Derrek Long
- derreklong.com
- taximagic.com / ridecharge.com