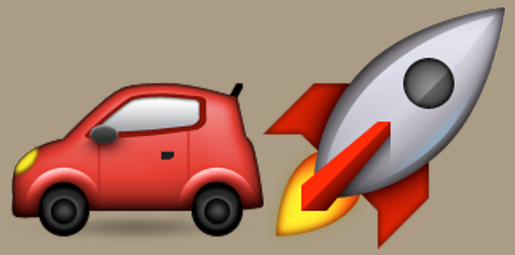
# DDD-rail Your Monorail

*Breaking up the Rails monolith with domain driven design*

*Hi, I'm Andrew*

Carbon Five



# Meet Delorean

*"It's like Uber, for time travel!"*

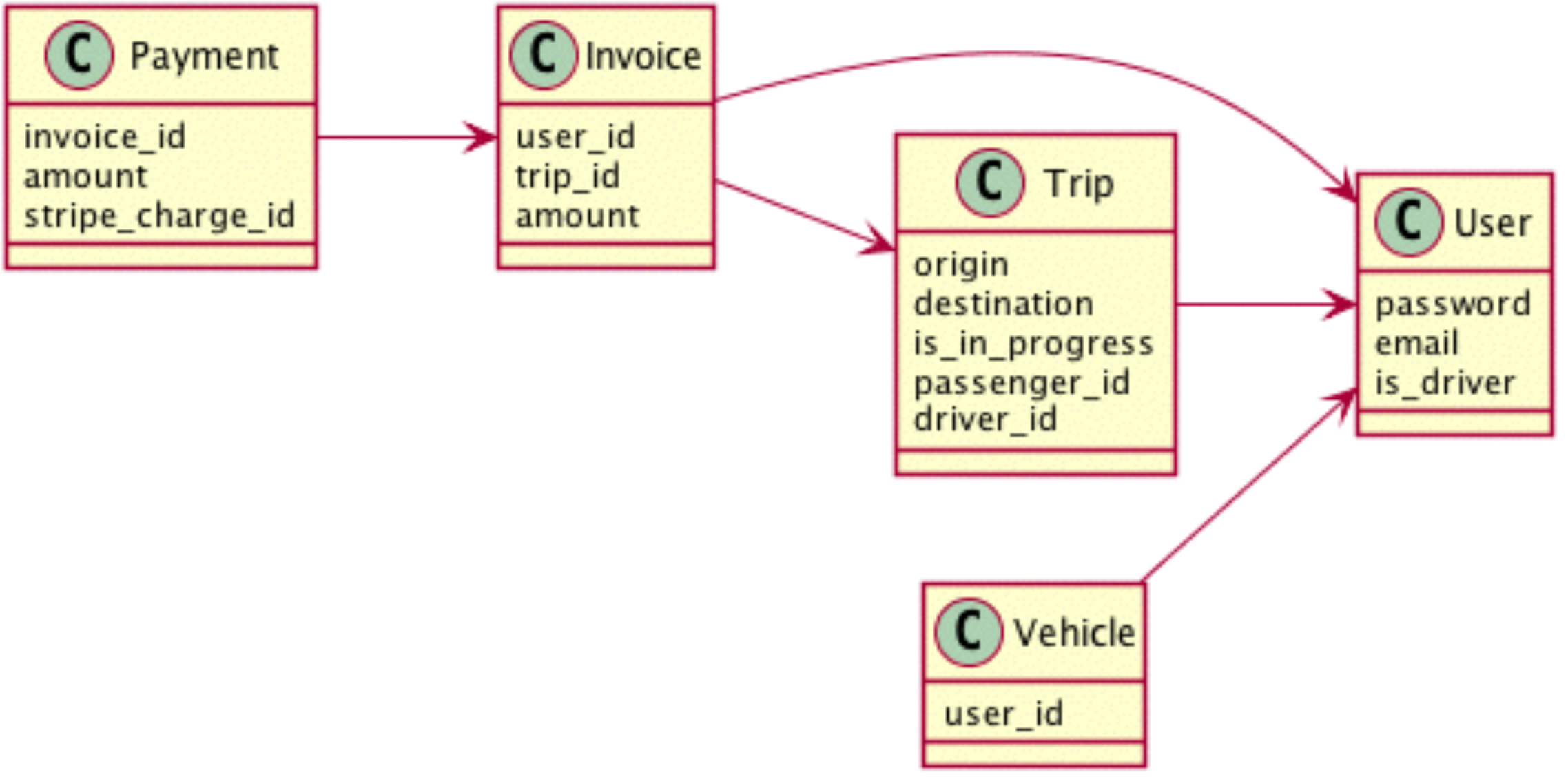

The situation

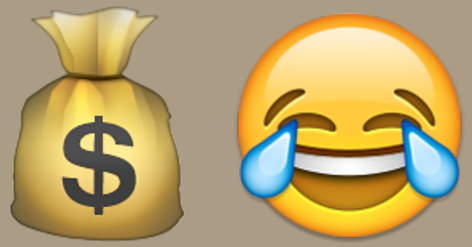
*What happened?*

In the beginning, there was an app.

# *The business decides...*

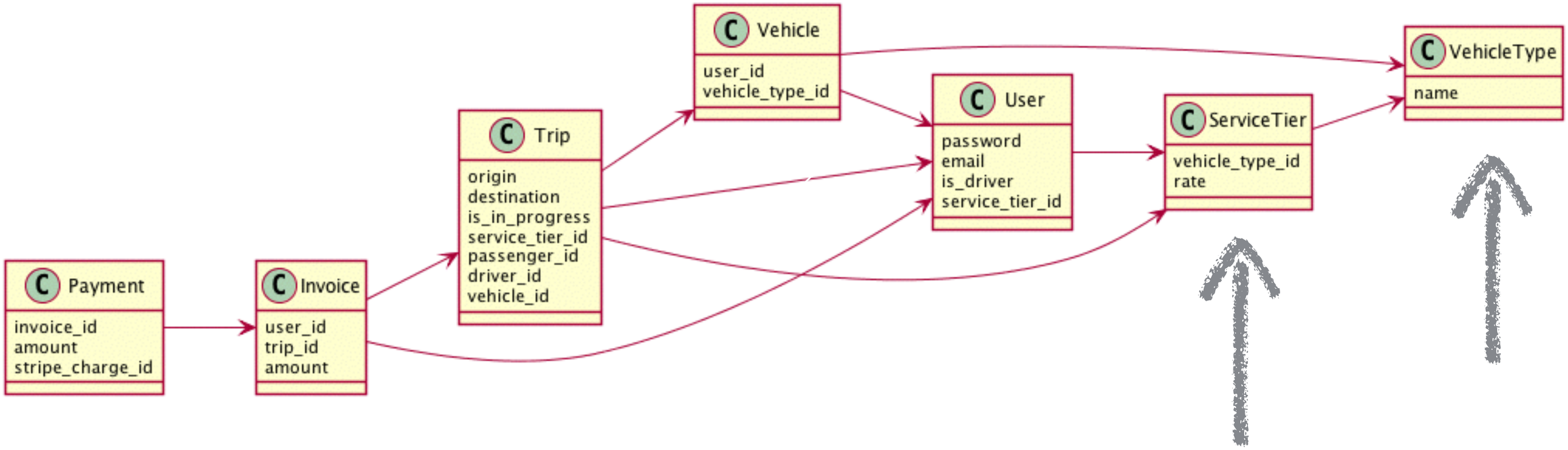
- Drivers deliver passengers from year A to year B. 🚕⏳

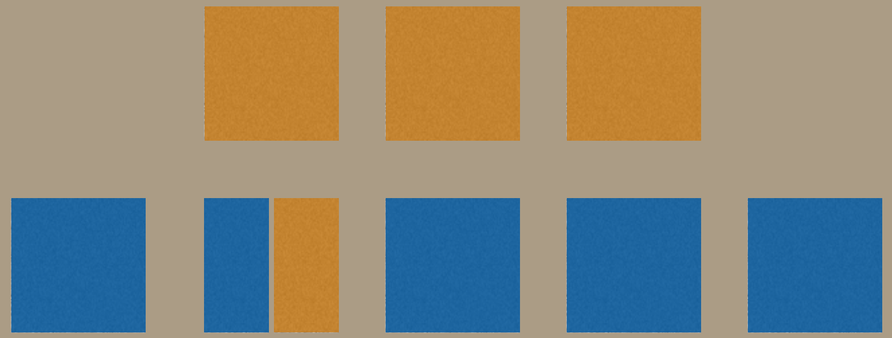

Payment
invoice_id
amount
stripe_charge_id
Invoice
user_id
trip_id
amount
Trip
origin
destination
is_in_progress
passenger_id
driver_id
User
password
email
is_driver
Vehicle
user_id



# *The business decides...*

- Drivers deliver passengers from year A to year B. 🚕⏳
- **Passengers choose the type of Delorean service they want.** 🚕🚗🚐

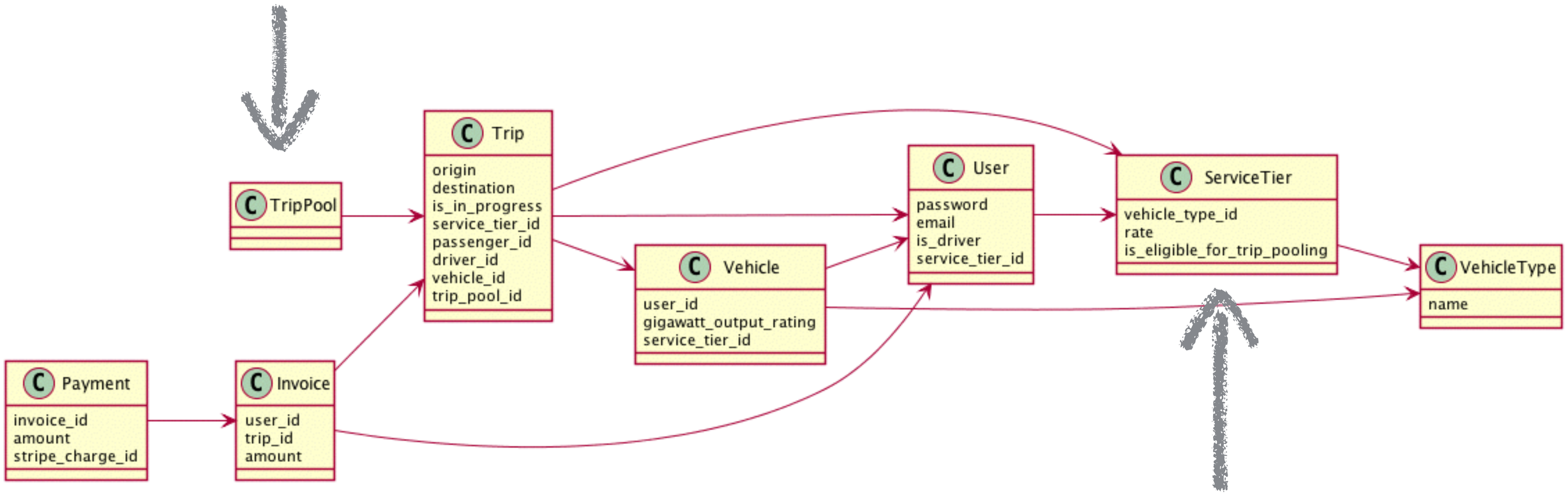

Payment
invoice_id
amount
stripe_charge_id
Invoice
user_id
trip_id
amount
Trip
origin
destination
is_in_progress
service_tier_id
passenger_id
driver_id
vehicle_id
Vehicle
user_id
vehicle_type_id
User
password
email
is_driver
service_tier_id
ServiceTier
vehicle_type_id
rate
VehicleType
name

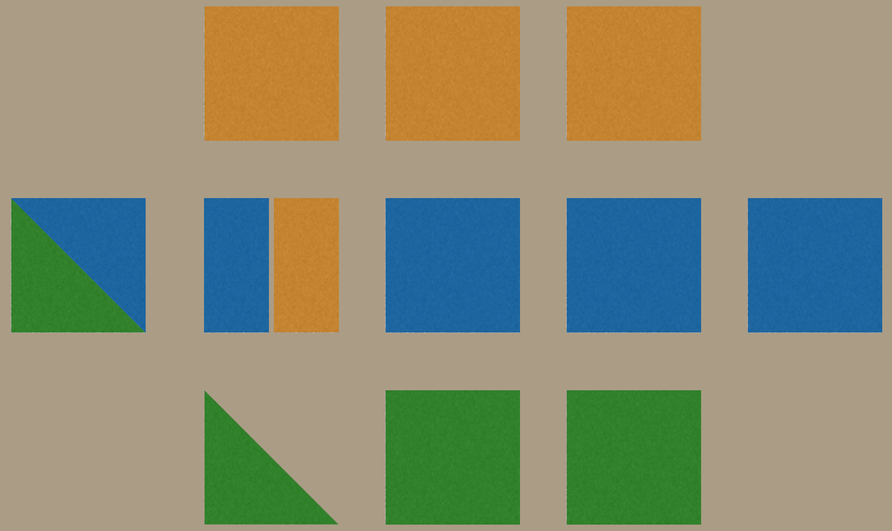



The Apple Times

# *The business decides...*

- Drivers deliver passengers from year A to year B. 🚕⏳
- Passengers choose the type of Delorean service they want. 🚕🚗🚐
- **Time travel carpools!** 🚕👨🏽👩🏻

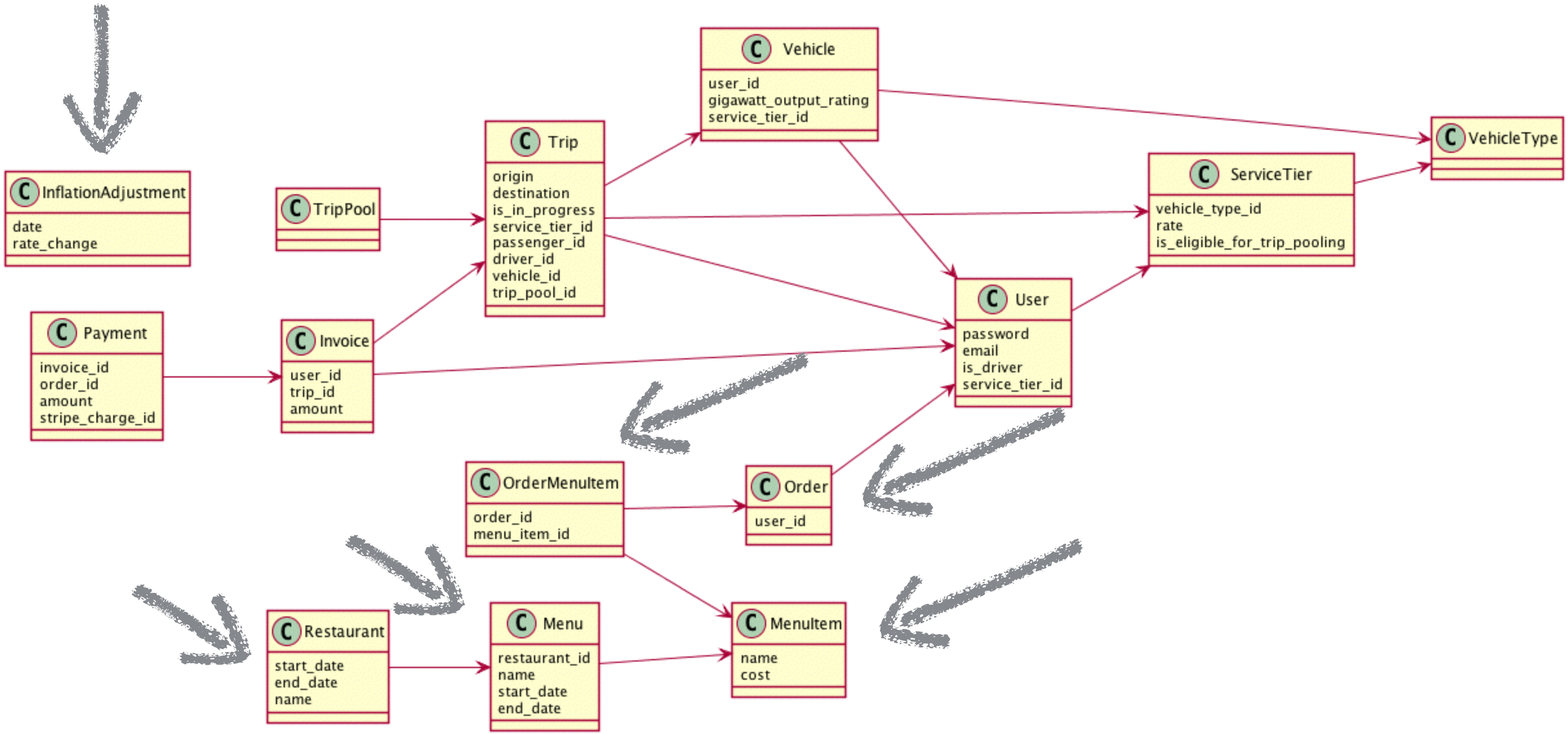

Trip
origin
destination
is_in_progress
service_tier_id
passenger_id
driver_id
vehicle_id
trip_pool_id
TripPool
User
password
email
is_driver
service_tier_id
ServiceTier
vehicle_type_id
rate
is_eligible_for_trip_pooling
Vehicle
user_id
gigawatt_output_rating
service_tier_id
VehicleType
name
Payment
invoice_id
amount
stripe_charge_id
Invoice
user_id
trip_id
amount

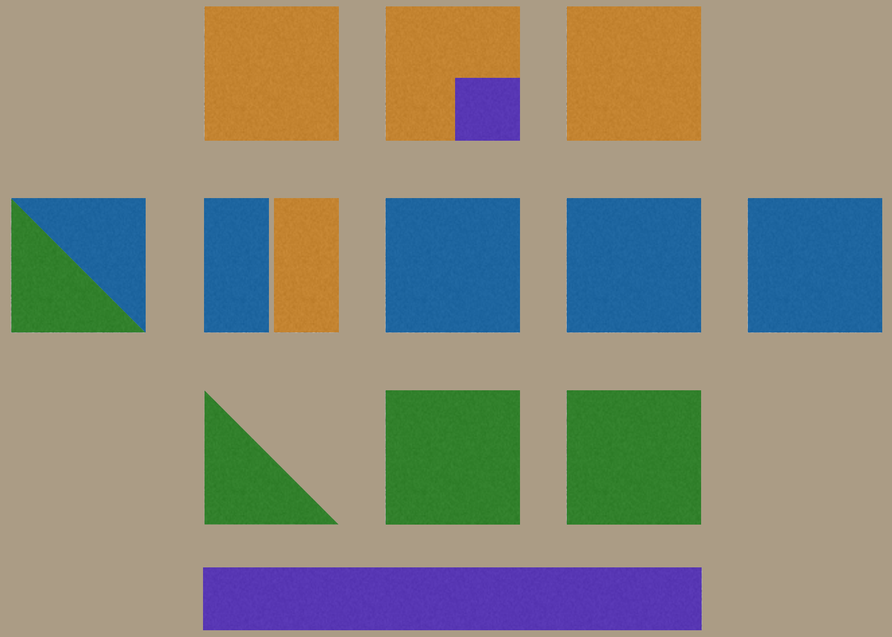



The Apple Times

# *The business decides...*

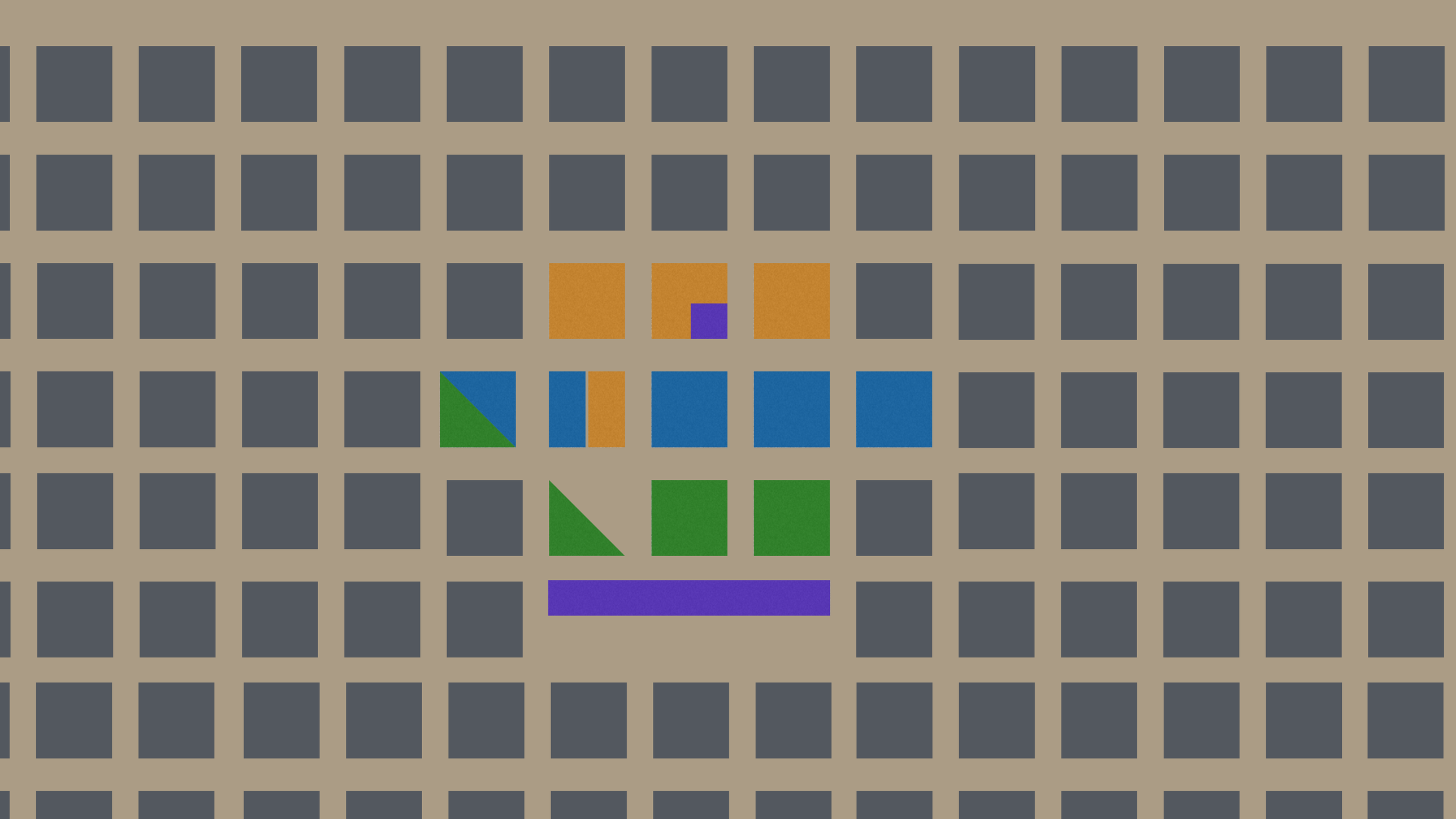
- Drivers deliver passengers from year A to year B. 🚕⏳
- Passengers choose the type of Delorean service they want. 🚕🚗🚐
- Time travel carpools! 🚕👨🏽👩🏻
- **DeloreanEATS: Customers order food, drivers pick up from time period and deliver to customer time period!** 🚕🍔


InflationAdjustment
date
rate_change
TripPool
Trip
origin
destination
is_in_progress
service_tier_id
passenger_id
driver_id
vehicle_id
trip_pool_id
Vehicle
user_id
gigawatt_output_rating
service_tier_id
VehicleType
ServiceTier
vehicle_type_id
rate
is_eligible_for_trip_pooling
Payment
invoice_id
order_id
amount
stripe_charge_id
Invoice
user_id
trip_id
amount
User
password
email
is_driver
service_tier_id
OrderMenuItem
order_id
menu_item_id
Order
user_id
Restaurant
start_date
end_date
name
Menu
restaurant_id
name
start_date
end_date
MenuItem
name
cost




The Apple Times

Hm.



# 💣 Regressions

*The Payments team regresses Trip while refactoring ServiceTier*

*The Restaurants team deploys a new pricing algorithm that regresses rideshare pricing*

💣 Responsibilities?

*The Mobile team requests a new mobile login API, but which team should implement it?*

💣 Business won’t stop

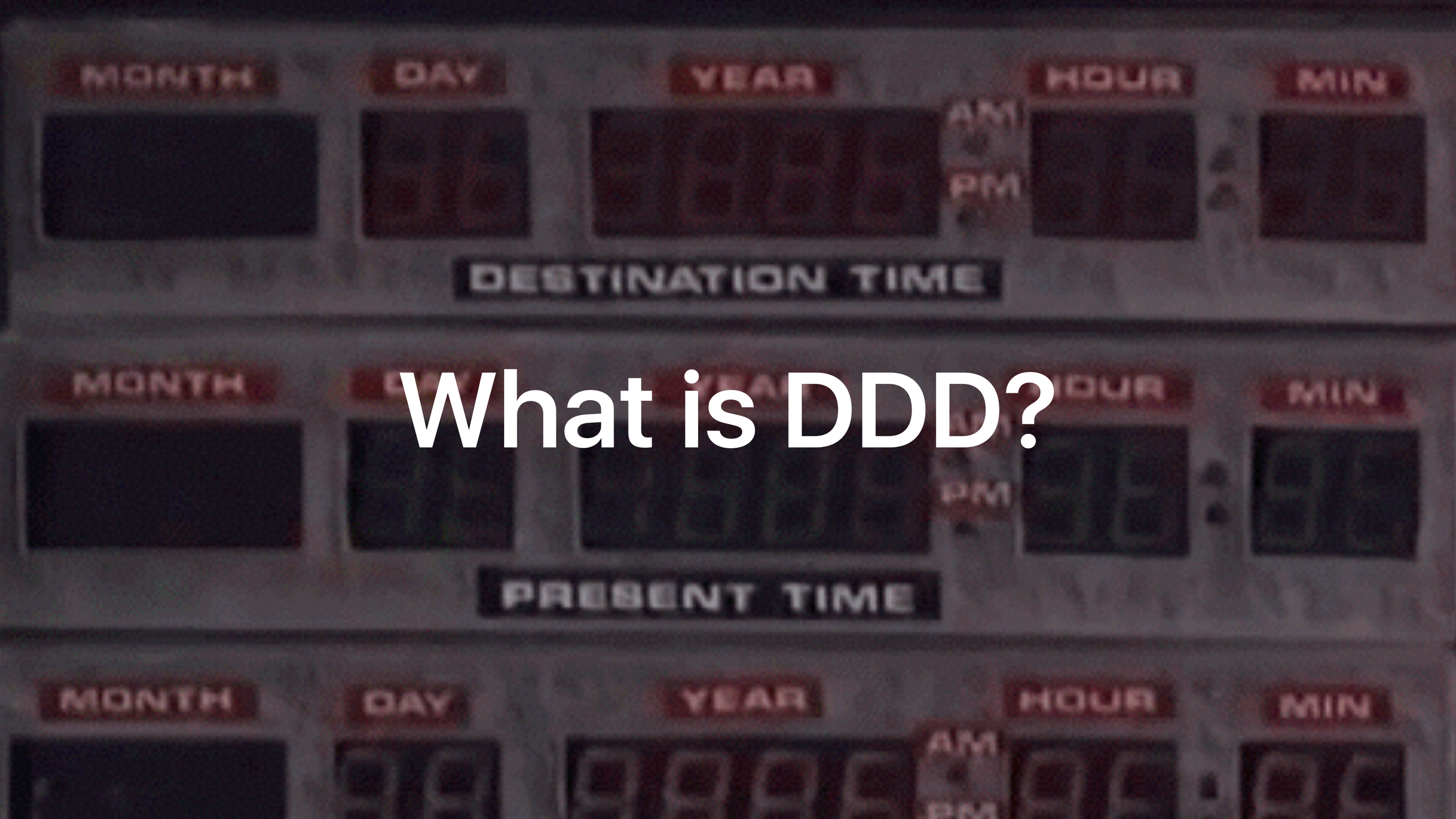
*Outsourced dev shop is rebuilding the marketing home page and needs a pricing API.*

*CEO wants to launch "Airbnb for time travel" feature in, let's say, 2 months!*

Does this sound familiar?

There are deeper insights to be had

# What is DDD?

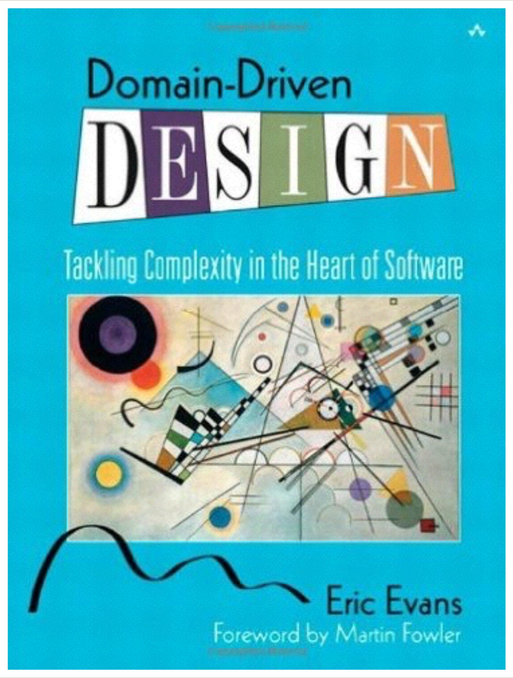

MONTH
DAY
YEAR
HOUR
MIN
AM
PM
DESTINATION TIME
MONTH
HOUR
MIN
PM
PRESENT TIME
MONTH
DAY
YEAR
HOUR
MIN
AM
PM

*A set of techniques to arrive at a flexible design that cleanly maps to the business model*

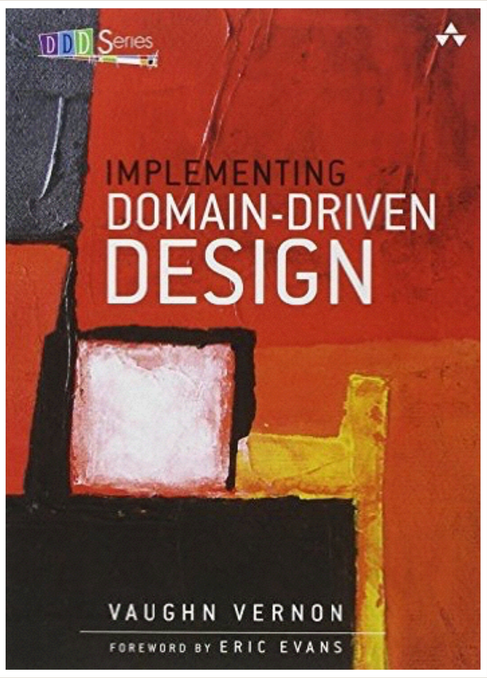
*Strong, expressive domain models*

*There is no such thing as a One True Perfect model*

*Embrace the chaos*




Domain-Driven Design: Eric Evans




Implementing Domain-Driven Design: Vaughn Vernon


💥 Let’s get designing!

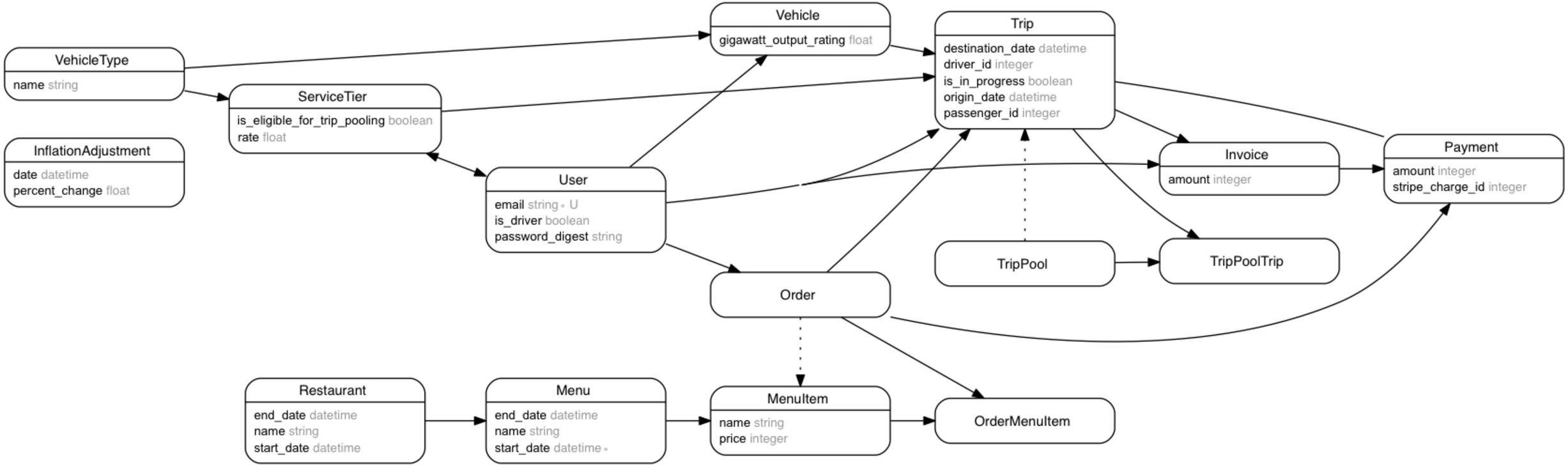
# Step 1: Visualize your domain models

# Rails ERD

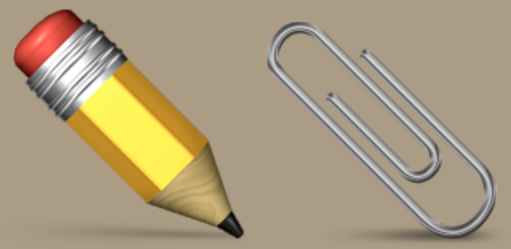
https://github.com/voormedia/rails-erd

Ruby gem to generate UML diagrams from your ActiveRecord models

Delorean domain model



This helps you get the entire system into your mind.



Print it out!

# Step 2: Find your core- and sub-domains

# Concept: Core Domain

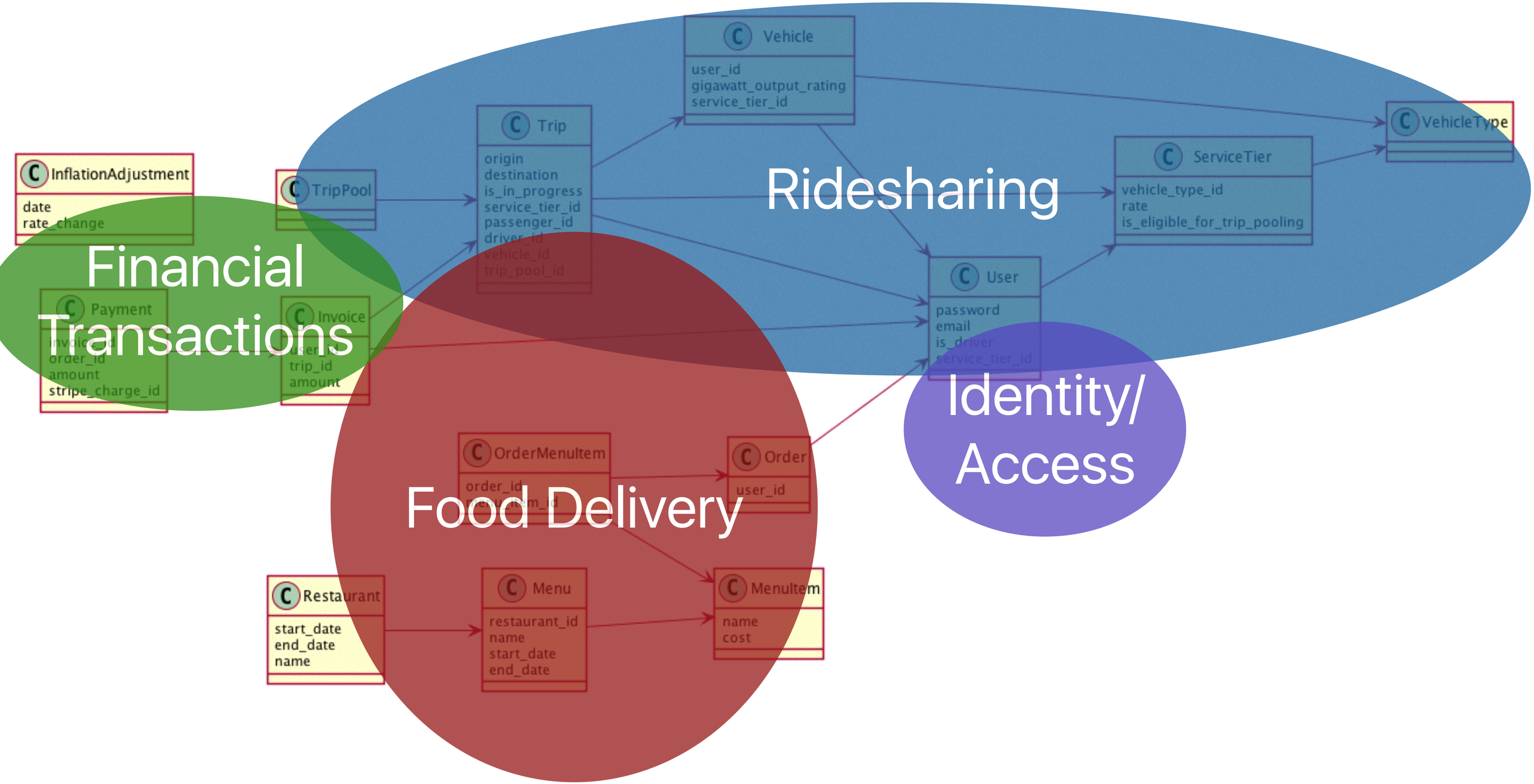
# *What a business does*

# Transportation Core Domain

# Concept: Subdomains

*A supporting unit within the business*

Rideshare Subdomain
Food Delivery Subdomain
Financial Transaction Subdomain
Identity and Access Subdomain


Ridesharing
Financial Transactions
Food Delivery
Identity/ Access
Vehicle
user_id
gigawatt_output_rating
service_tier_id
Trip
origin
destination
is_in_progress
service_tier_id
passenger_id
driver_id
vehicle_id
trip_pool_id
VehicleType
ServiceTier
vehicle_type_id
rate
is_eligible_for_trip_pooling
InflationAdjustment
date
rate_change
TripPool
User
password
email
is_driver
service_tier_id
Payment
order_id
amount
stripe_charge_id
Invoice
trip_id
amount
OrderMenuItem
order_id
Order
user_id
Restaurant
start_date
end_date
name
Menu
restaurant_id
name
start_date
end_date
MenuItem
name
cost

# Step 3: Get the team talking in the same language

# Concept: Ubiquitous Language

*A defined language that is used consistently across business and technical contexts*

Bring in the domain experts

A Driver picks up a Passenger

A Passenger requests a Pickup

An Owner drives a Vehicle

An Operator? drives a Vehicle

A User logs in and changes her Password

An Invoice is delivered to the Passenger

A Customer orders an item from a Menu, which is picked up and delivered by the Delivery Agent

# Step 4: Make a glossary

# Ridesharing

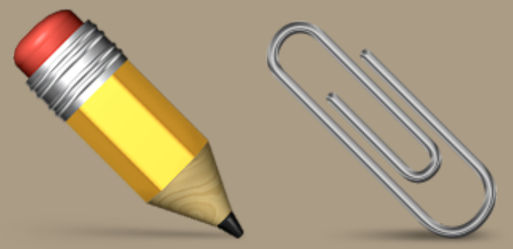
- Passenger: “...”
- Driver: “...”
- Trip: “...”
- Pickup: “...”
- Vehicle: “...”
- Vehicle Owner: “...”

# Financial Transaction

- Invoice: “...”
- Order: “...”
- Payment: “...”
- Royalty: “...”
- Salary: “...”

# Identity and Access

- User: “...”
- Secure password: “...”
- Role: “...”



Print ‘em out!

# Step 5: Draw out your software systems (bounded contexts)

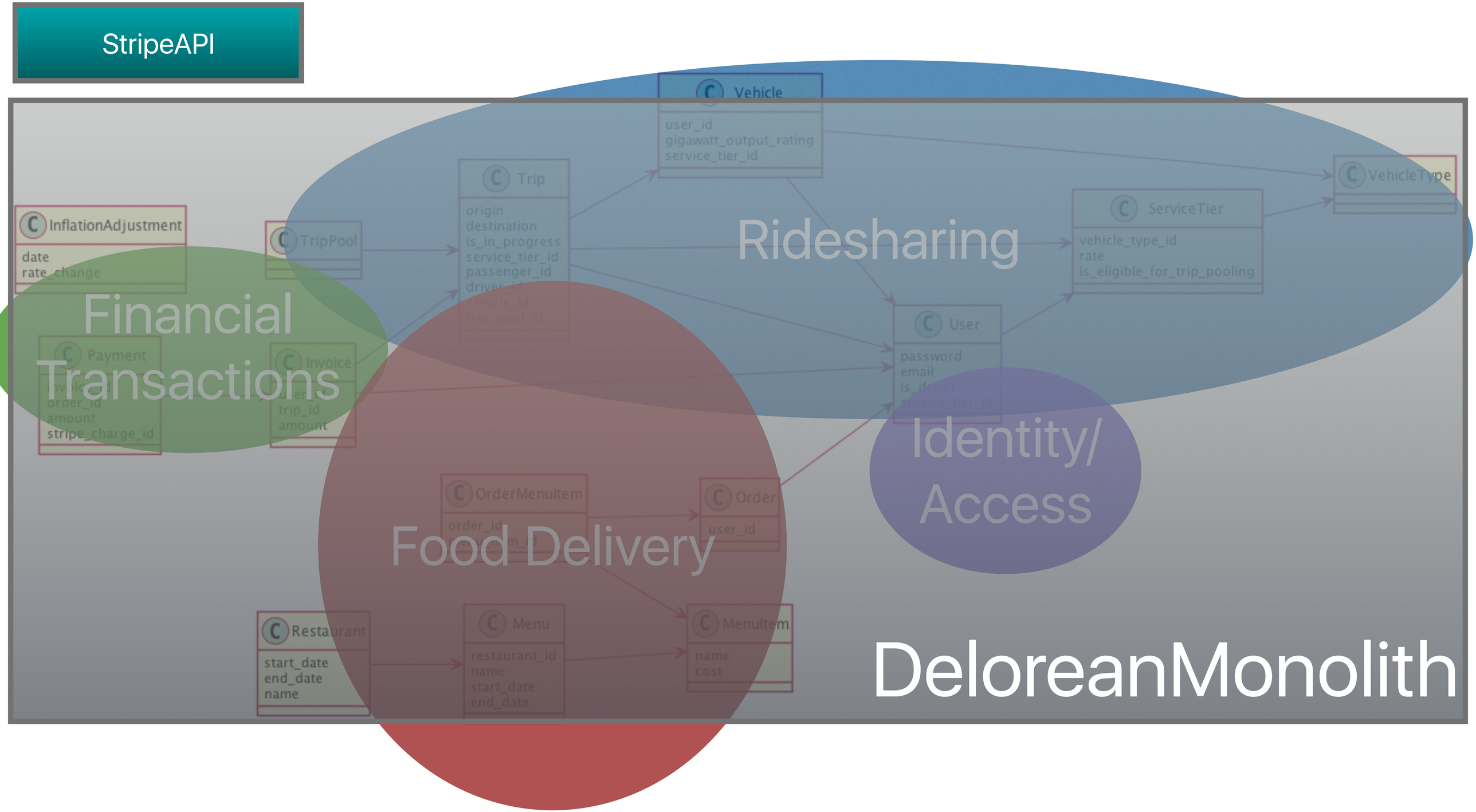
# Concept: Bounded Contexts

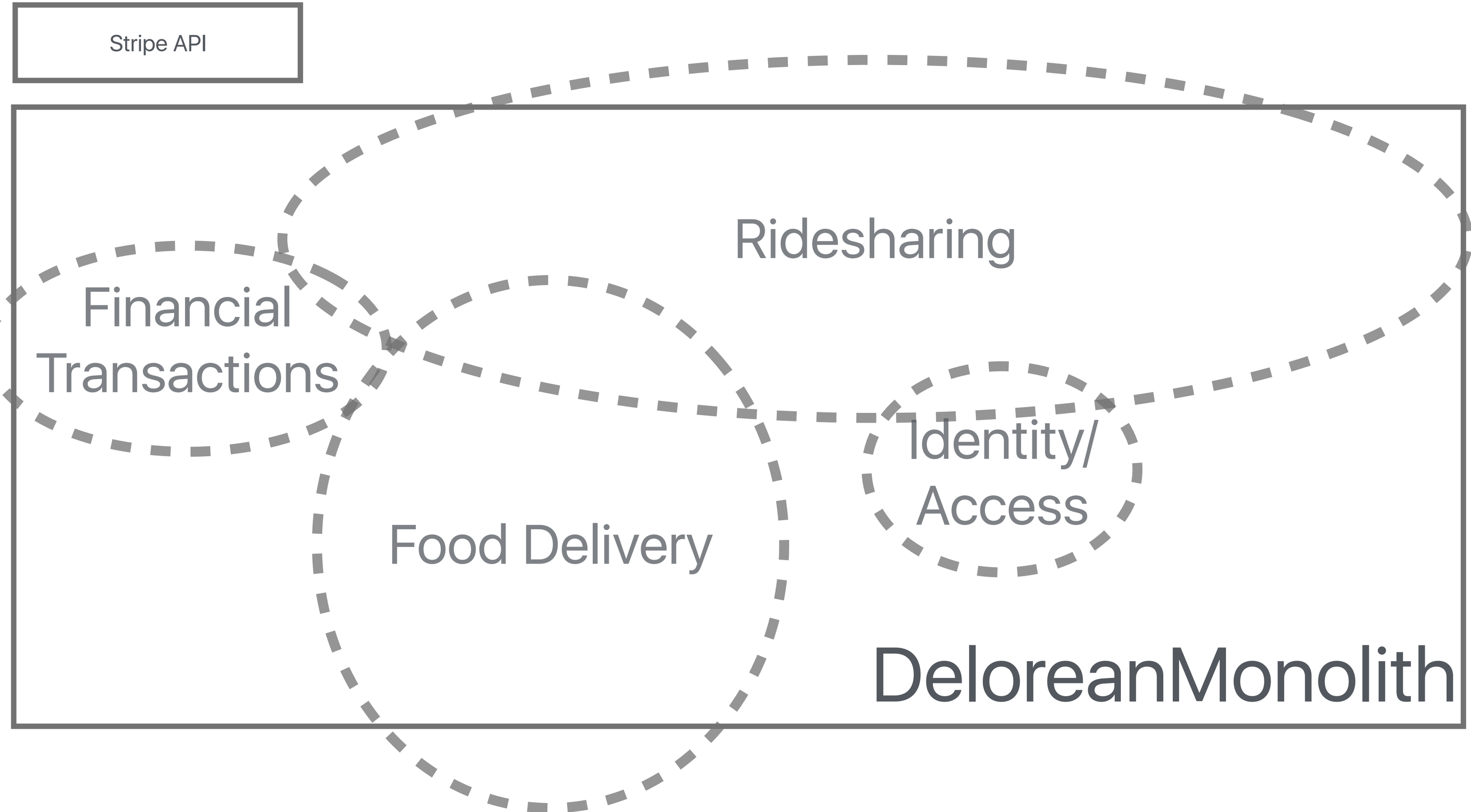
*A software system that defines the applicability of a ubiquitous language*

| FoodDeliveryService | TripRoutingEngine |
| --- | --- |
| “menu” | “trip” |
| “restaurant” | “plan” |
| “delivery” | “pickup” |
| “customer date” | “destination” |

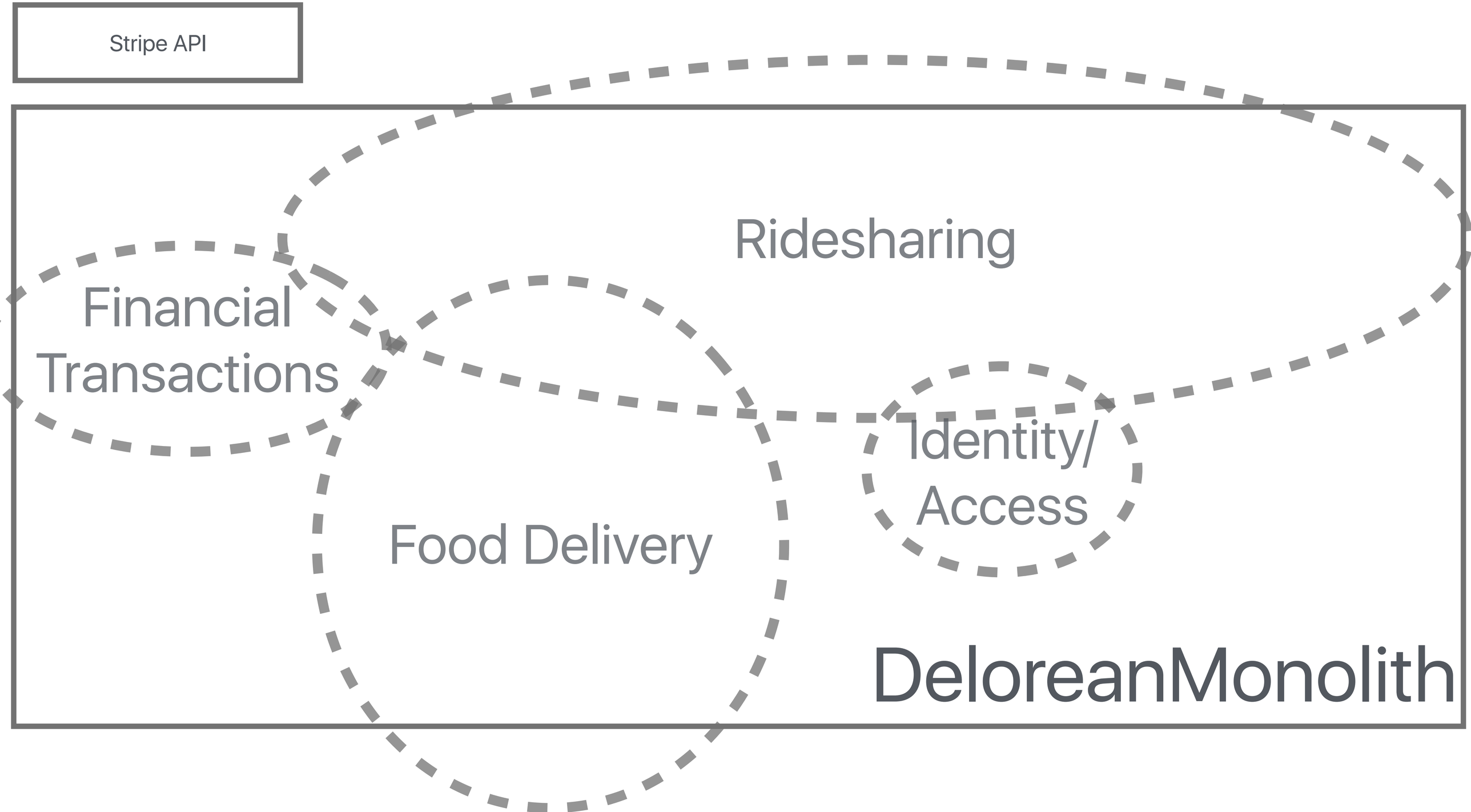
Software systems are natural boundaries for these linguistic terms

If a term leaks into a different software system/bounded context, you have a smell

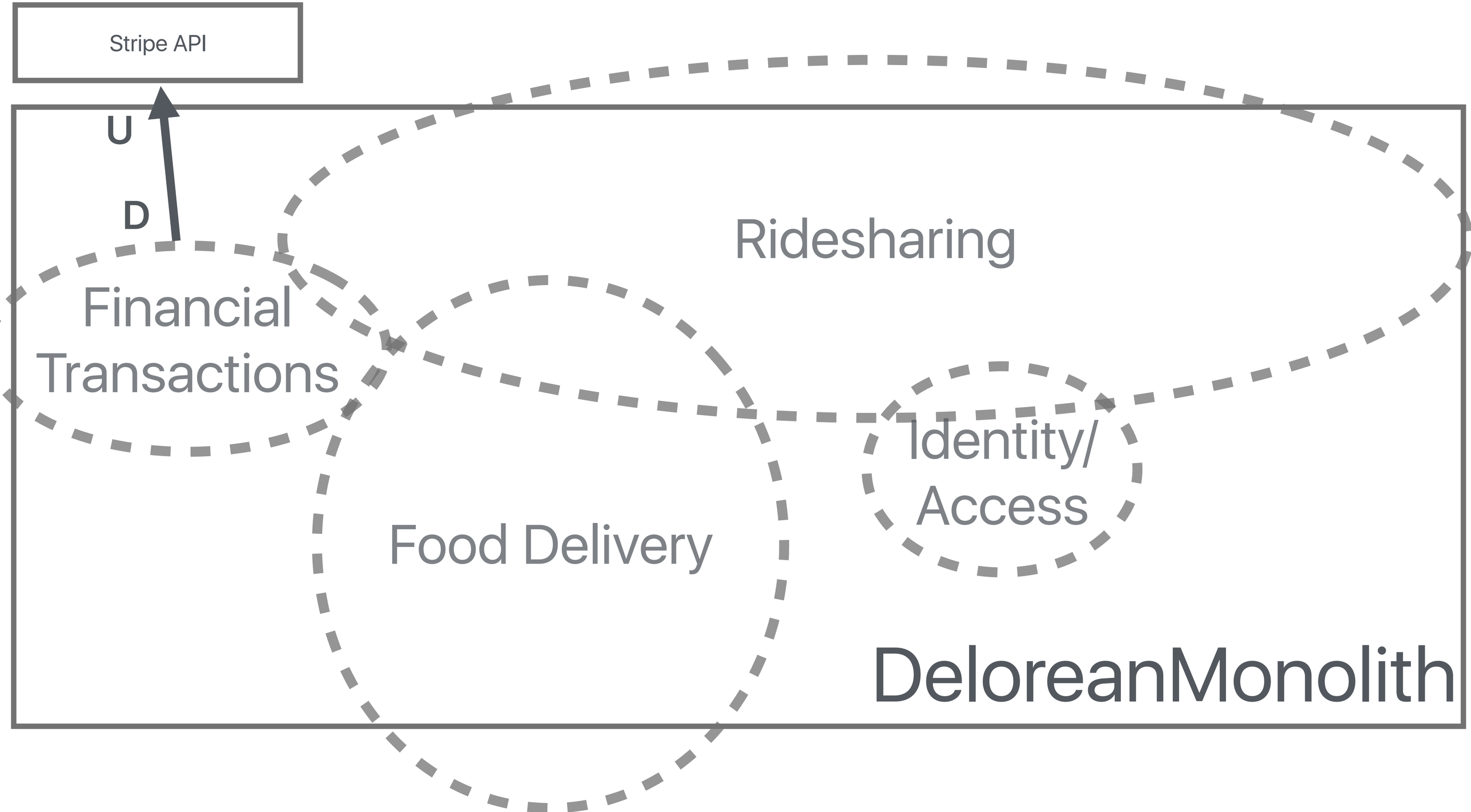

StripeAPI
Vehicle
user_id
gigawatt_output_rating
service_tier_id
Trip
origin
destination
is_in_progress
service_tier_id
passenger_id
vehicle_id
trip_pool_id
VehicleType
ServiceTier
vehicle_type_id
rate
is_eligible_for_trip_pooling
InflationAdjustment
date
rate_change
TripPool
Ridesharing
Financial
Transactions
Payment
order_id
amount
stripe_charge_id
Invoice
trip_id
amount
User
password
email
Identity/
Access
OrderMenuItem
order_id
Order
user_id
Food Delivery
Restaurant
start_date
end_date
name
Menu
restaurant_id
name
start_date
end_date
MenuItem
name
cost
DeloreanMonolith

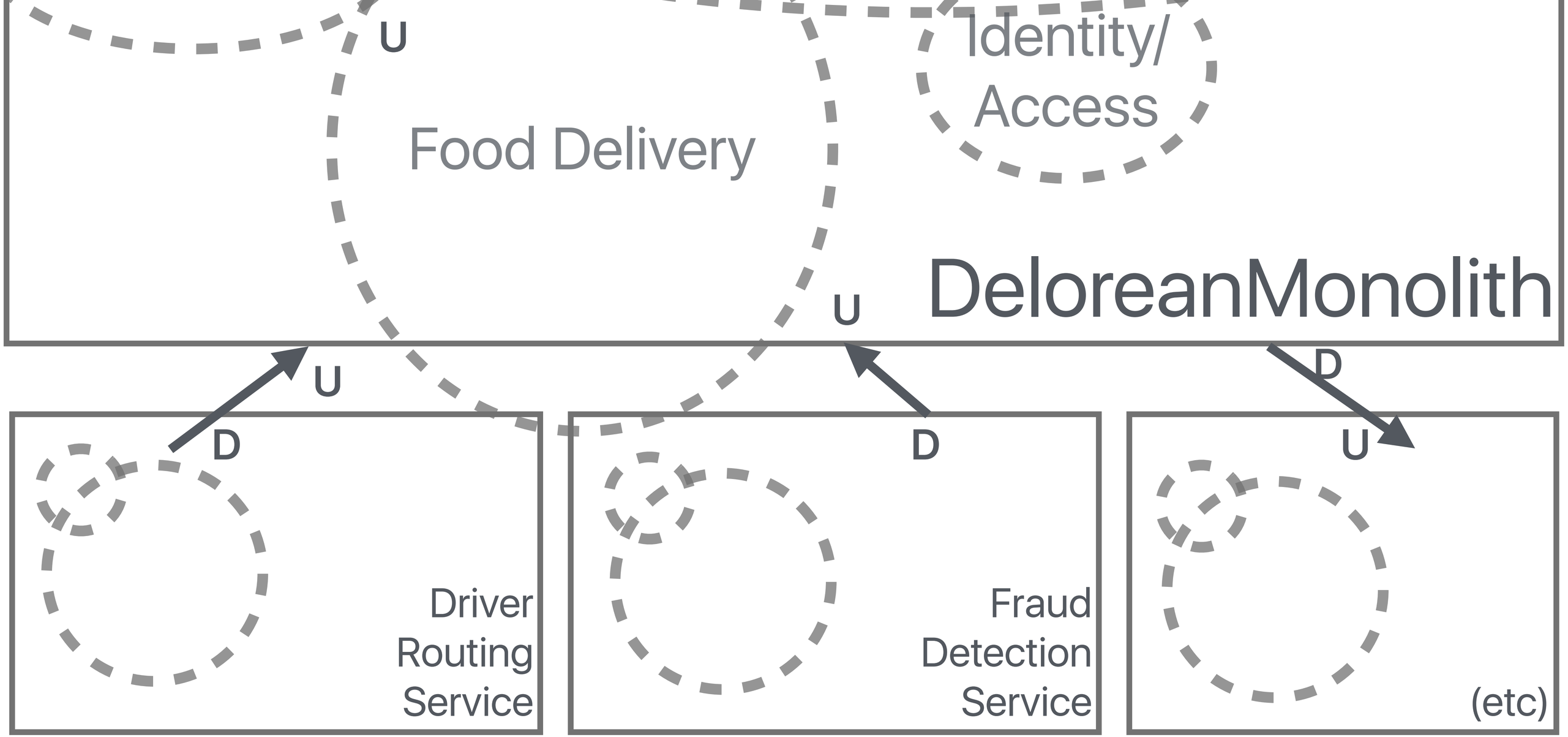

Stripe API
Ridesharing
Financial Transactions
Identity/ Access
Food Delivery
DeloreanMonolith


Identity/
Access
Food Delivery
DeloreanMonolith
Driver
Routing
Service
Fraud
Detection
Service
(etc)


Stripe API
Ridesharing
Financial Transactions
Identity/ Access
Food Delivery
DeloreanMonolith

# Step 6: Now add directional dependencies

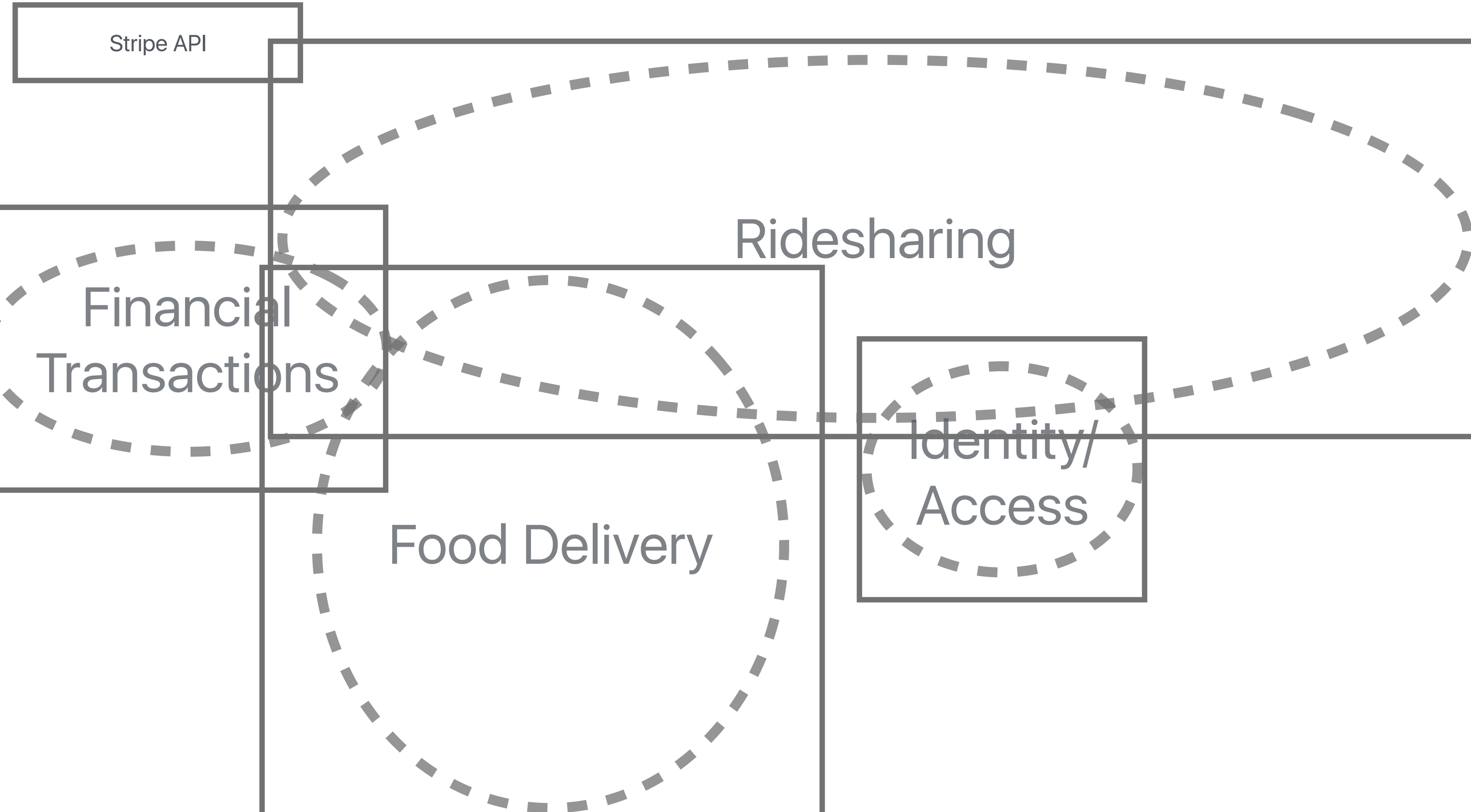

Stripe API
U
D
Ridesharing
Financial Transactions
Identity/ Access
Food Delivery
DeloreanMonolith


U
Identity/
Access
Food Delivery
DeloreanMonolith
U
U
D
D
D
U
Driver
Routing
Service
Fraud
Detection
Service
(etc)

This helps you see dependencies between teams, where communication will be most important.

Context Map: A tool to visualize how your software systems relate to each other and the domain(s)

Our goal is to map our bounded contexts directly to our subdomains.


Stripe API
Ridesharing
Financial Transactions
Identity/ Access
Food Delivery


Show me the code!

# *Tactic 1: Get the names right*

Adjust your class and method names to reflect the ubiquitous language

```
# old
User.without_drivers
# new
Passenger.hailing_drivers

# old
order.calculate_cost
# new
order.calculate_billing_total
```

# *Tactic 2: Namespace and modulize your domains*

# Break Rails’ folder structure conventions

```
app/domains/financial
app/domains/financial/inflation_adjustment.rb
app/domains/financial/invoice.rb
app/domains/food_delivery
app/domains/food_delivery/menu.rb
app/domains/food_delivery/menu_item.rb
app/domains/food_delivery/menu_items_controller.rb
app/domains/food_delivery/menus_controller.rb
app/domains/identity
app/domains/identity/user.rb
app/domains/identity/users_controller.rb
app/domains/rideshare
app/domains/rideshare/driver.rb
app/domains/rideshare/passenger.rb
app/domains/rideshare/service_tier.rb
```

```ruby
class Menu < ActiveRecord::Base
  belongs_to :restaurant
end
```

```ruby
module FoodDelivery
  class Menu < ActiveRecord::Base
    belongs_to :restaurant
  end
end
```

```ruby
class Trip < ActiveRecord::Base
  belongs_to :service_tier
  belongs_to :vehicle
end
```

```ruby
module Rideshare
  class Trip < ActiveRecord::Base
    belongs_to :service_tier
    belongs_to :vehicle
  end
end
```

Domain code stays together

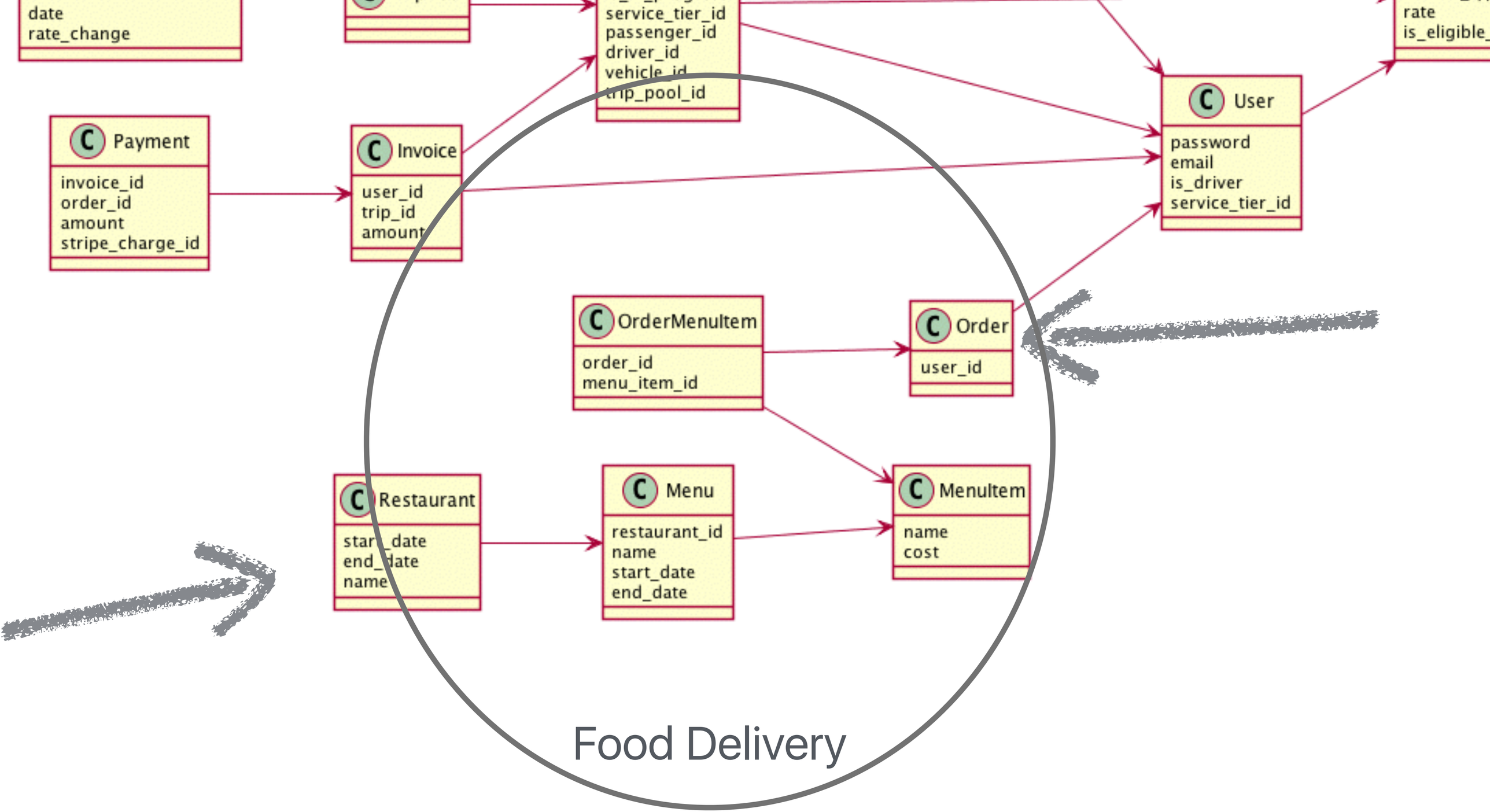
# *Tactic 3: Design with Aggregate Roots*

Aggregate: A collection of domain objects that can be treated as a single unit

Use aggregates to model real-world entities that belong together

Aggregate Root: An entity at the “top” of the collection that can represent the whole

FoodDelivery aggregate root: Order


date
rate_change
service_tier_id
passenger_id
driver_id
vehicle_id
trip_pool_id
rate
is_eligible_
Payment
invoice_id
order_id
amount
stripe_charge_id
Invoice
user_id
trip_id
amount
User
password
email
is_driver
service_tier_id
OrderMenuItem
order_id
menu_item_id
Order
user_id
Restaurant
start_date
end_date
name
Menu
restaurant_id
name
start_date
end_date
MenuItem
name
cost
Food Delivery

A user adds a menu item to their cart

# A User adds a MenuItem into their ShoppingCart

```ruby
module Financial
  class ShoppingCart
    def order
      @order ||= FoodDelivery::Order.new
    end

    def amount
      @order.menu_items.sum(&:cost) +
        @order.menu_items.sum(&:tax_amount)
    end

    def add(item)
      order.menu_items << item
    end

    def remove(item)
      order.menu_items.delete(item); order.save
    end
  end
end
```

```ruby
module FoodDelivery
  class Order < ActiveRecord::Base
    belongs_to :user
    has_many :order_menu_items
    has_many :menu_items, through: :order_menu_items
  end
end
```

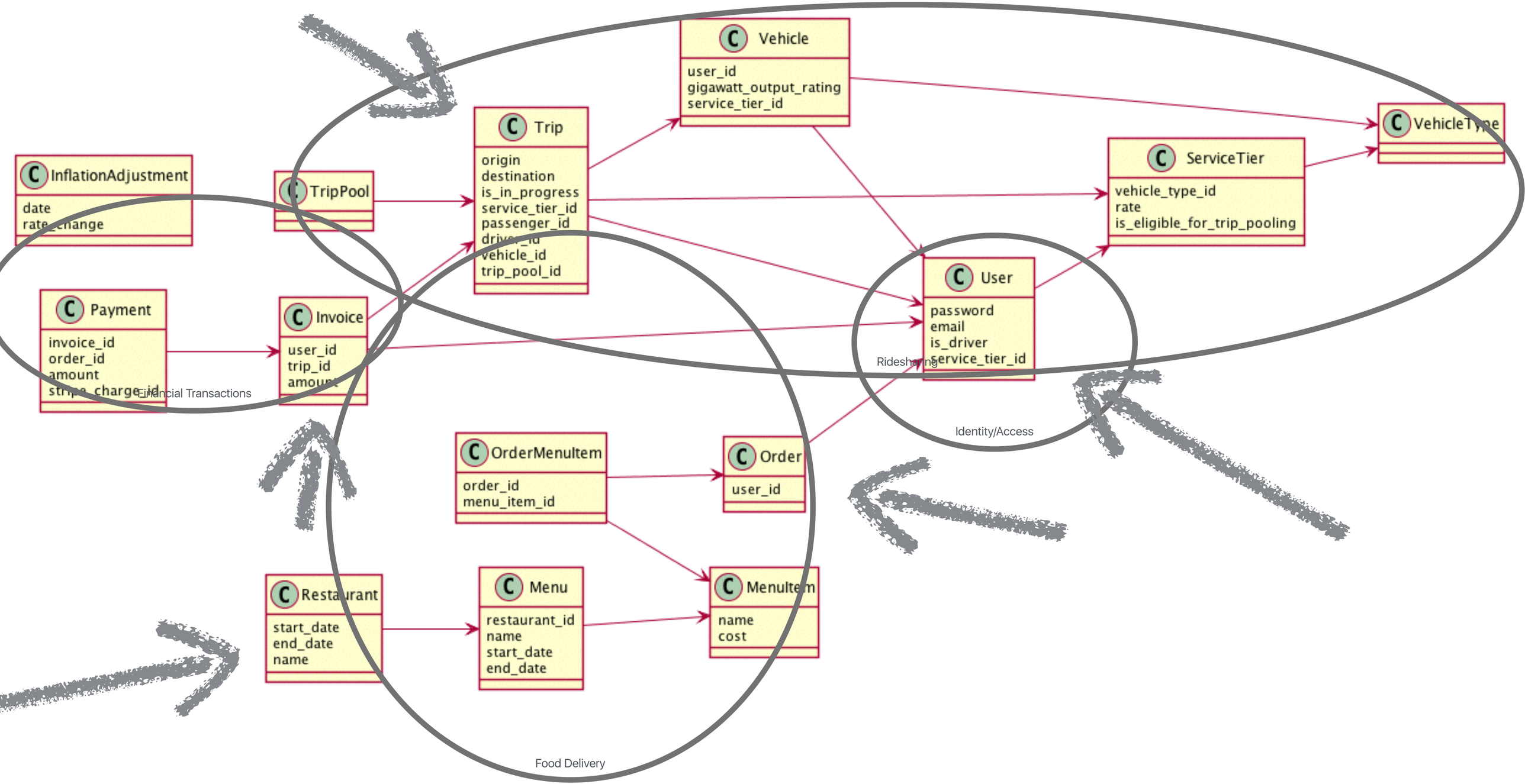
```ruby
module FoodDelivery
  class Order < ActiveRecord::Base
    belongs_to :user
    has_many :order_menu_items
    has_many :menu_items, through: :order_menu_items

    def total_cost
      item_cost + tax_cost
    end

    def item_cost
      menu_items.sum(&:cost)
    end

    def tax_cost
      menu_items.sum(&:tax_amount)
    end

    def add_item!(menu_item)
      menu_items << menu_item
    end

    def remove_item!(menu_item)
      menu_items.delete(menu_item); save
    end
  end
end
```

```ruby
module Financial
  class ShoppingCart
    def order
      @order ||= FoodDelivery::Order.new
    end

    def amount
      @order.total_cost
    end

    def add(item)
      order.add_item!(item)
    end

    def remove(item)
      order.remove_item!(item)
    end
  end
end
```

**Calculation logic kept in FoodDelivery domain**

**Implementation-agnostic**

The root items of these aggregates are the only entities that external callers may fetch


Vehicle
user_id
gigawatt_output_rating
service_tier_id
VehicleType
Trip
origin
destination
is_in_progress
service_tier_id
passenger_id
driver_id
vehicle_id
trip_pool_id
ServiceTier
vehicle_type_id
rate
is_eligible_for_trip_pooling
InflationAdjustment
date
rate_change
TripPool
User
password
email
is_driver
service_tier_id
Payment
invoice_id
order_id
amount
stripe_charge_id
Invoice
user_id
trip_id
amount
Financial Transactions
Ridesharing
Identity/Access
OrderMenuItem
order_id
menu_item_id
Order
user_id
Restaurant
start_date
end_date
name
Menu
restaurant_id
name
start_date
end_date
MenuItem
name
cost
Food Delivery

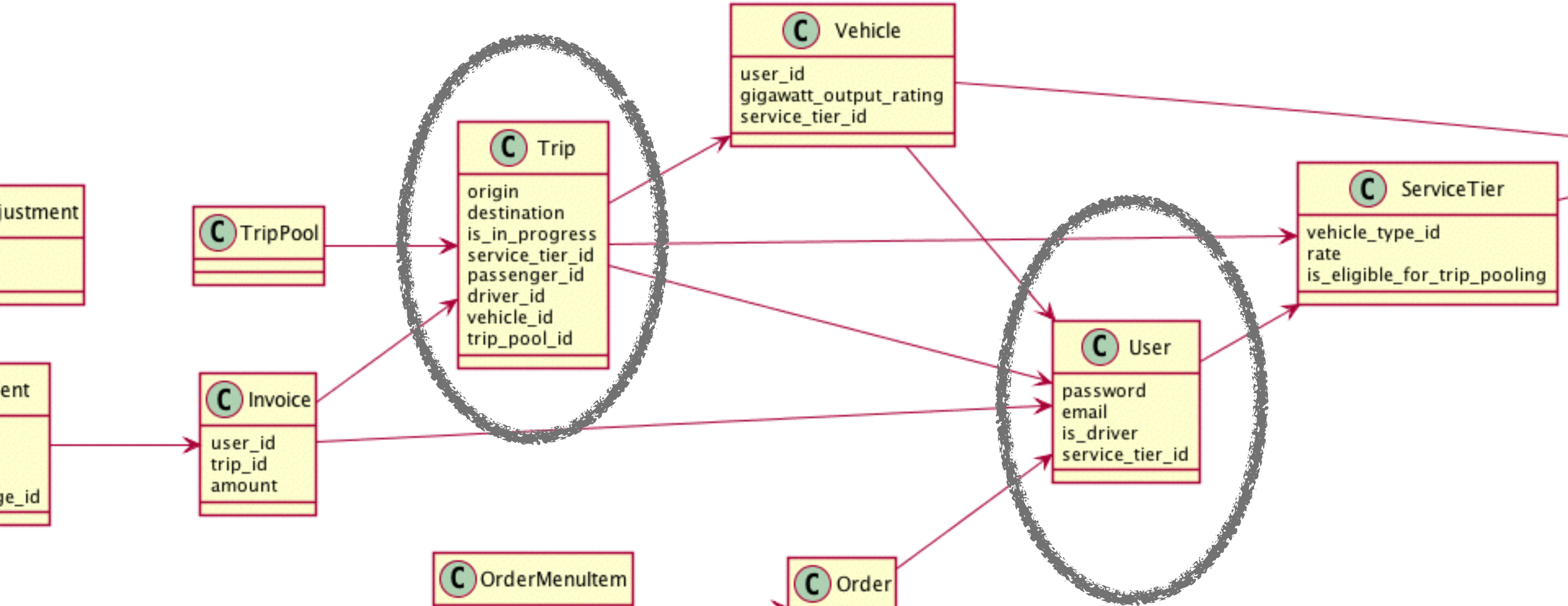
Building good interfaces for a service oriented future!

# *Tactic 4: Break database joins between domains*

`has_many`-itis!

```
module Rideshare
  class Trip < ActiveRecord::Base
    belongs_to :service_tier
    has_many :trip_pool_trips
    has_one :trip_pool, through: :trip_pool_trips
    belongs_to :driver, foreign_key: :driver_id, class_name: I
    belongs_to :passenger, foreign_key: :passenger_id, class_n
    belongs_to :vehicle
    belongs_to :order, class_name: FoodDelivery::Order
    has_one :payment
end
```

These tend to happen in your God Objects


Vehicle
user_id
gigawatt_output_rating
service_tier_id
Trip
origin
destination
is_in_progress
service_tier_id
passenger_id
driver_id
vehicle_id
trip_pool_id
TripPool
ServiceTier
vehicle_type_id
rate
is_eligible_for_trip_pooling
User
password
email
is_driver
service_tier_id
Invoice
user_id
trip_id
amount
OrderMenuItem
Order

```
module Rideshare
  class Trip < ActiveRecord::Base
    belongs_to :service_tier
    has_many :trip_pool_trips
    has_one :trip_pool, through: :trip_pool_trips
    belongs_to :driver, foreign_key: :driver_id, class_name: I
    belongs_to :passenger, foreign_key: :passenger_id, class_n
    belongs_to :vehicle
    belongs_to :order, class_name: FoodDelivery::Order
    has_one :payment
end
```

```ruby
module Rideshare
  class Trip < ActiveRecord::Base
    # belongs_to :order, class_name: FoodDelivery::Order
    def order
      FoodDeliveryAdapter.new.order_from_trip(self)
    end
  end
end
```

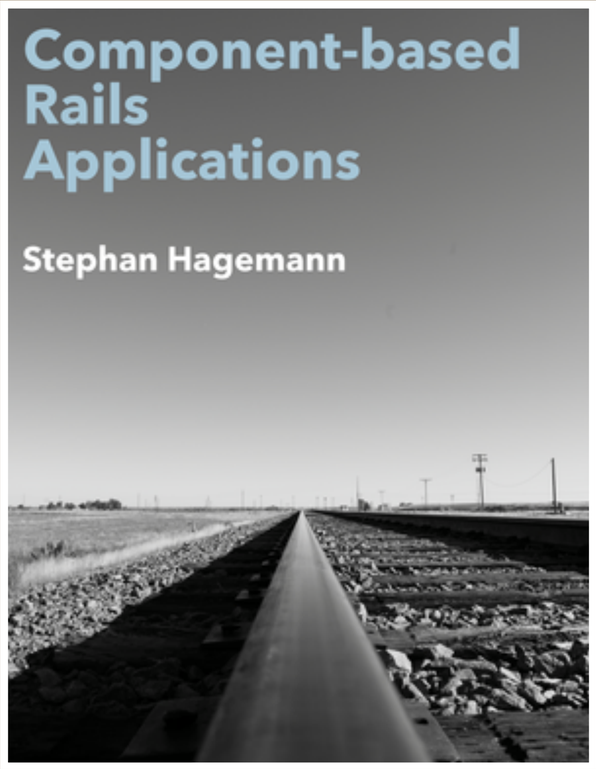
```ruby
module Rideshare
  class FoodDeliveryAdapter
    def order_from_trip(trip)
      FoodDelivery::Order.find_by(trip_id: trip.id)
    end
  end
end
```

Decoupling domains now will ease your architectural transitions later

*You can do this all in small steps!*

# Toward a distributed architecture

# Namespaced, Isolated Modules to Rails Engines




Component-Based Rails Engines (Stephan Hagemann)

Rails engines to a remote (micro-)service

Good architecture goes a very long way.

# 🚕 To recap 💨

Drew things on a wall

Came up with a language

Moved code around

Team is on the same page

*Big idea: Your software speaks the same language as your domain experts*

*Big idea: Bounded contexts are separators for linguistic drivers. Keep your language consistent in one and only one system*

*Big idea: Domain code should live together*

*Big idea: Adapters between domains enforce domain purity*

*Big idea: Incremental changes*

# Thanks!

**Sample code**: https://github.com/andrewhao/delorean
**Slides**: https://github.com/andrewhao/dddrail-talk

**Twitter** @andrewhao
**Github** @andrewhao