Built to Last

A domain-driven approach to beautiful systems

Andrew Hao
@andrewhao

Carbon Five

Welcome to your first day at Delorean!

It's like Uber... for time travel!

And it's a hot mess!

We moved a teensy bit too fast:

1. Too many teams in one codebase

And it's a hot mess!

We moved a teensy bit too fast:

1. Too many teams in one codebase
2. Changing a feature changes multiple codebases

And it's a hot mess!

We moved a teensy bit too fast:

1. Too many teams in one codebase
2. Changing a feature changes multiple codebases
3. Concepts inconsistently named

And it's a hot mess!

We moved a teensy bit too fast:

1. Too many teams in one codebase
2. Changing a feature changes multiple codebases
3. Concepts inconsistently named
4. Ship, ship, ship! (No time to refactor)

Hi, I'm Andrew

Friendly neighborhood programmer at Carbon Five

Carbon Five

I've been thinking about beautiful systems

In the language - syntax, form, expressiveness

I've been thinking about beautiful systems

In the language - syntax, form, expressiveness

In the tooling - developer ergonomics

I've been thinking about beautiful systems

In the language - syntax, form, expressiveness

In the tooling - developer ergonomics

In the tests - test practices & coverage

I've been thinking about beautiful systems

In the language - syntax, form, expressiveness

In the tooling - developer ergonomics

In the tests - test practices & coverage

In its **longevity** - whether it stands the test of time with changing business and product requirements

Long-lasting systems

Just large enough - knows its boundaries

Long-lasting systems

Just large enough - knows its boundaries

Highly cohesive and loosely coupled

Long-lasting systems

Just large enough - knows its boundaries

Highly cohesive and loosely coupled

Precise semantics that fully express the business domain

A blast from the past

Information hiding

D.L. Parnas - "On the Criteria to Be Used in Decomposing Systems into Modules"

Programming Techniques
R. Morris Editor

On the Criteria To Be Used in Decomposing Systems into Modules

D.L. Parnas
Carnegie-Mellon University

This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. The effectiveness of a "modularization" is dependent upon the criteria used in dividing the system into modules. A system design problem is presented and both a conventional and unconventional decomposition are described. It is shown that the unconventional decompositions have distinct advantages for the goals outlined. The criteria used in arriving at the decompositions are discussed. The unconventional decomposition, if implemented with the conventional assumption that a module consists of one or more subroutines, will be less efficient in most cases. An alternative approach to implementation which does not have this effect is sketched.

Key Words and Phrases: software, modules, modularity, software engineering, KWIC index, software design
CR Categories: 4.0

Introduction

A lucid statement of the philosophy of modular programming can be found in a 1970 textbook on the design of system programs by Gouthier and Pont [1, ¶10.23], which we quote below:[1]

> A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently; there are few scheduling problems in synchronizing the completion of several tasks before checkout can begin. Finally, the system is maintained in modular fashion; system errors and deficiencies can be traced to specific system modules, thus limiting the scope of detailed error searching.

Usually nothing is said about the criteria to be used in dividing the system into modules. This paper will discuss that issue and, by means of examples, suggest some criteria which can be used in decomposing a system into modules.

A Brief Status Report

"We propose instead that one begins with a **list of difficult design decisions** or **design decisions which are likely to change**.

"Each module is then designed to **hide such a decision from the others.**" (Emphasis added)

From software program to the entire system

Where are the difficult design decisions in this company that are likely to change?

From software program to the entire system

Where are the difficult design decisions in this company that are likely to change?

Within the business groups that generate them!

A peek into the life of our systems:

- Marketing wants us to generate 5000 promo codes

A peek into the life of our systems:

- Marketing wants us to generate 5000 promo codes
- Finance needs us to implement a new audit log

A peek into the life of our systems:

- Marketing wants us to generate 5000 promo codes
- Finance needs us to implement a new audit log
- Product teams want us to launch new food delivery features.

A peek into the life of our systems:

- Marketing wants us to generate 5000 promo codes
- Finance needs us to implement a new audit log
- Product teams want us to launch new food delivery features.
- Marketing wants us to invalidate 2000 of the 5000 codes

A peek into the life of our systems:

- Marketing wants us to generate 5000 promo codes
- Finance needs us to implement a new audit log
- Product teams want us to launch new food delivery features.
- Marketing wants us to invalidate 2000 of the 5000 codes
- Finance needs us to add another attribute to the audit log

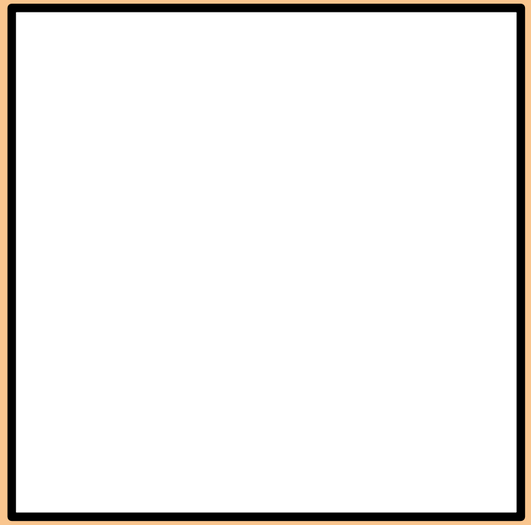
A peek into the life of our systems:

- Marketing wants us to generate 5000 promo codes
- Finance needs us to implement a new audit log
- Product teams want us to launch new food delivery features.
- Marketing wants us to invalidate 2000 of the 5000 codes
- Finance needs us to add another attribute to the audit log
- Product teams want us to launch food delivery in a second market

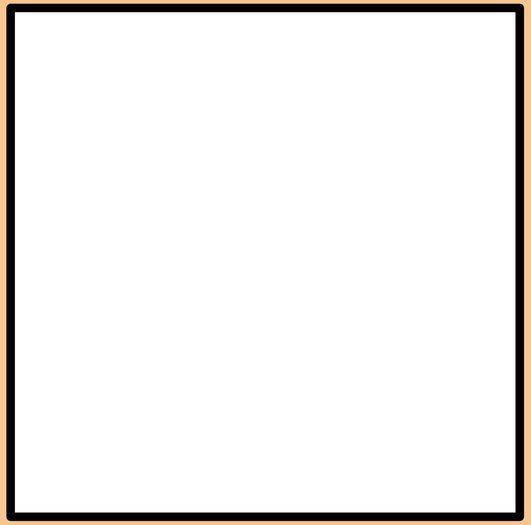
A peek into the life of our systems:

- Marketing wants us to generate 5000 promo codes
- Finance needs us to implement a new audit log
- Product teams want us to launch new food delivery features.
- Marketing wants us to invalidate 2000 of the 5000 codes
- Finance needs us to add another attribute to the audit log
- Product teams want us to launch food delivery in a second market

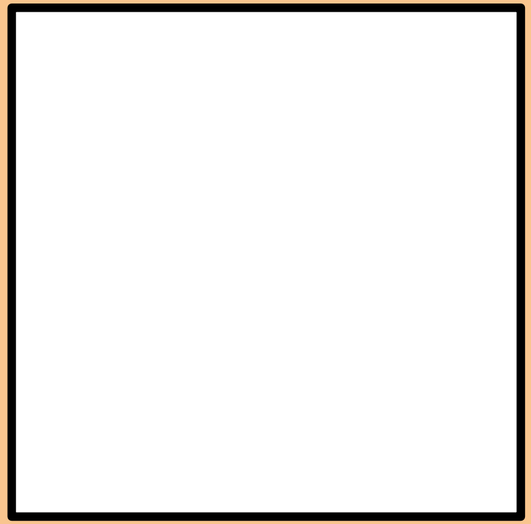
(That sounds like change!)



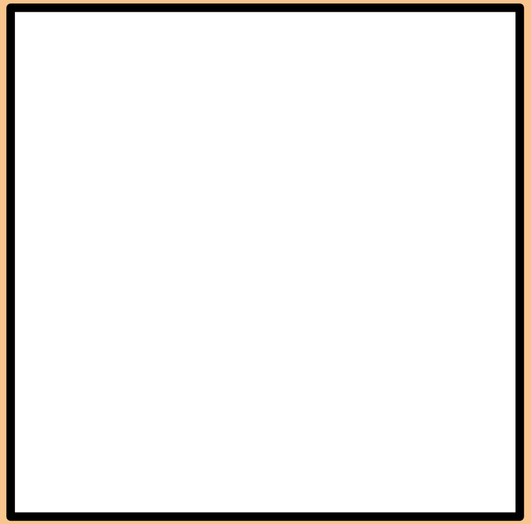
models



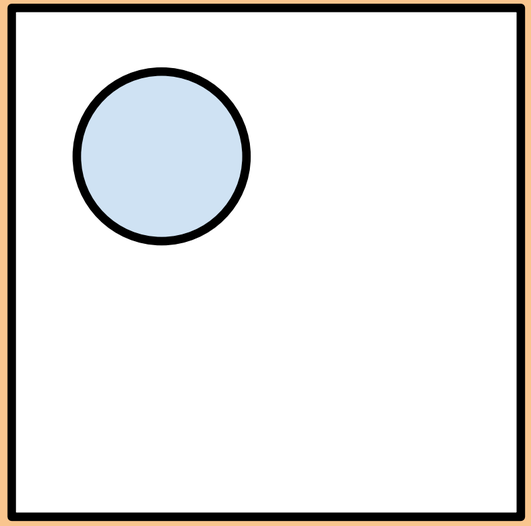
controllers



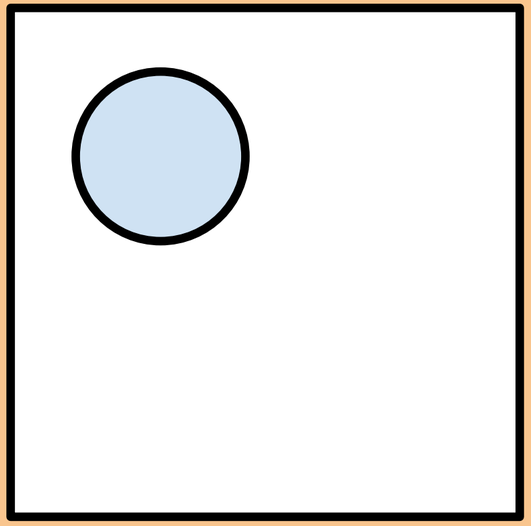
views



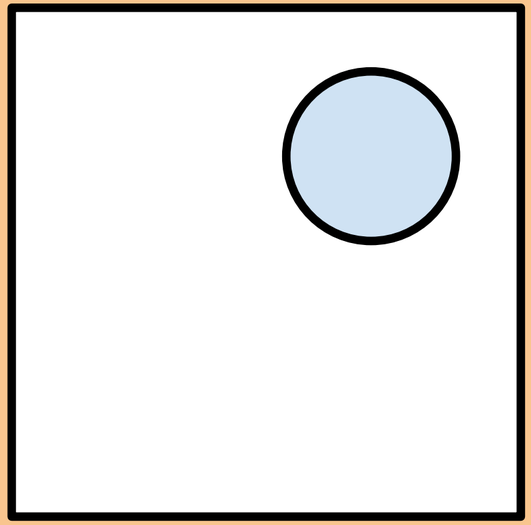
services



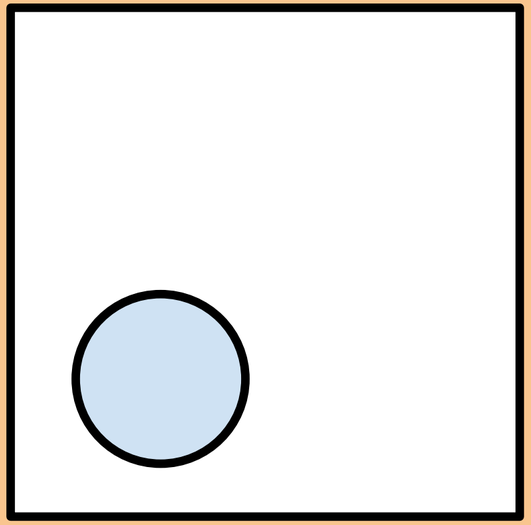
models



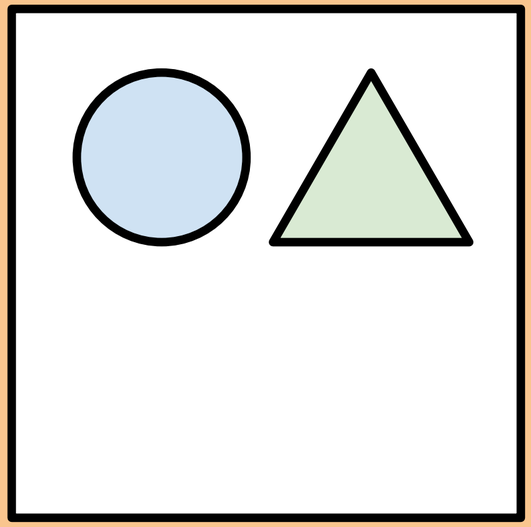
controllers



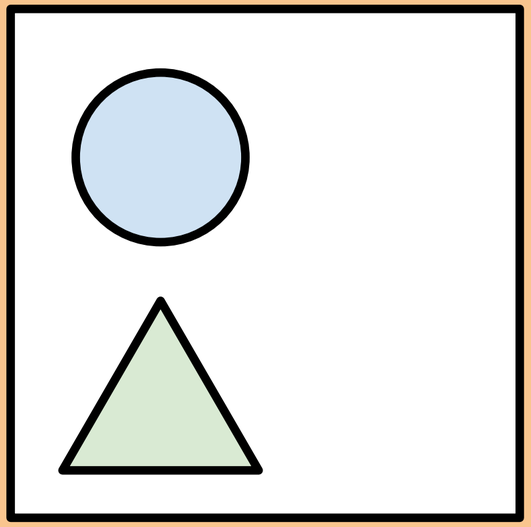
views



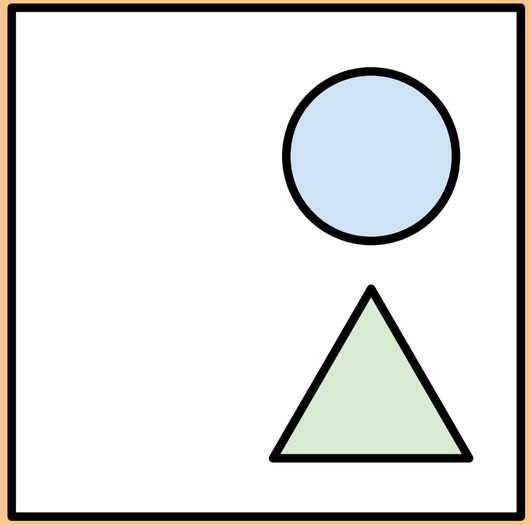
services



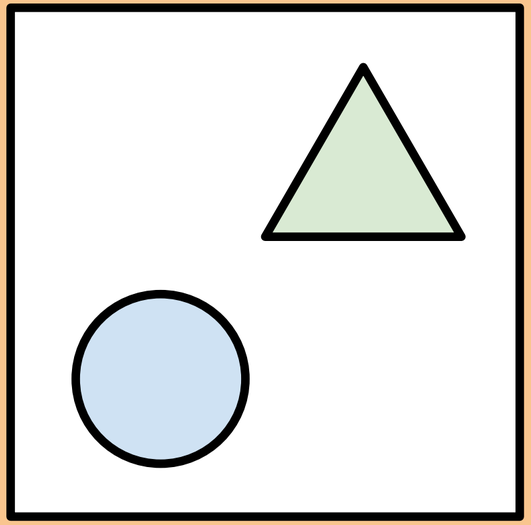
models



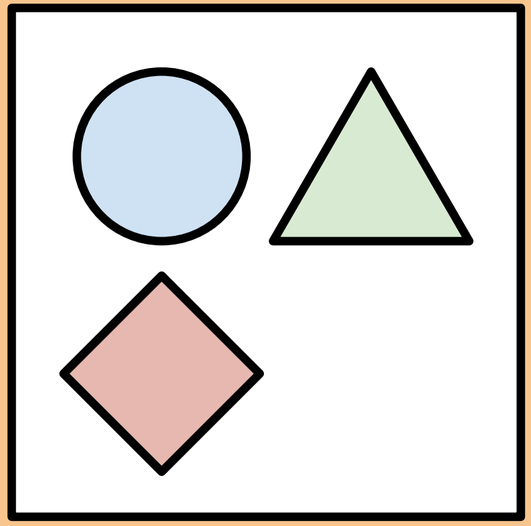
controllers



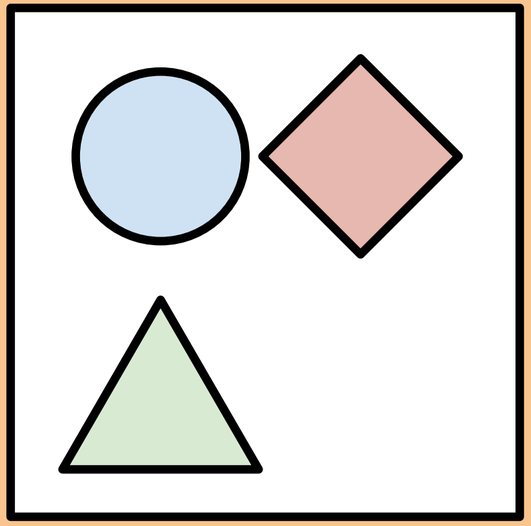
views



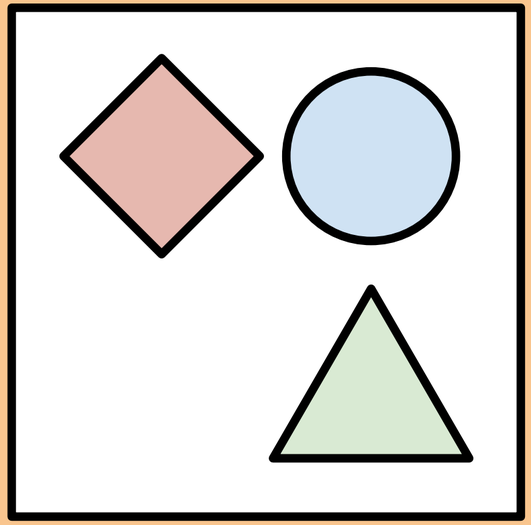
services



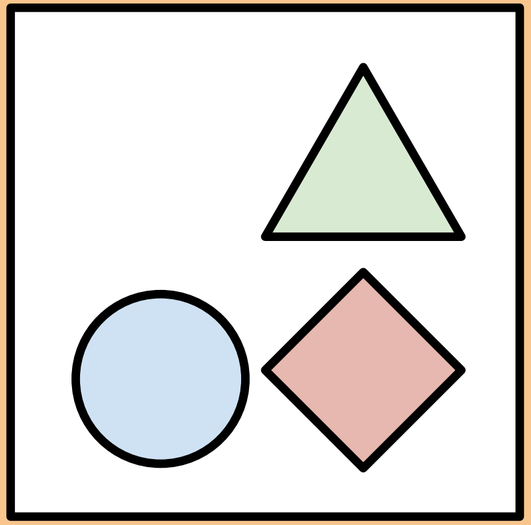
models



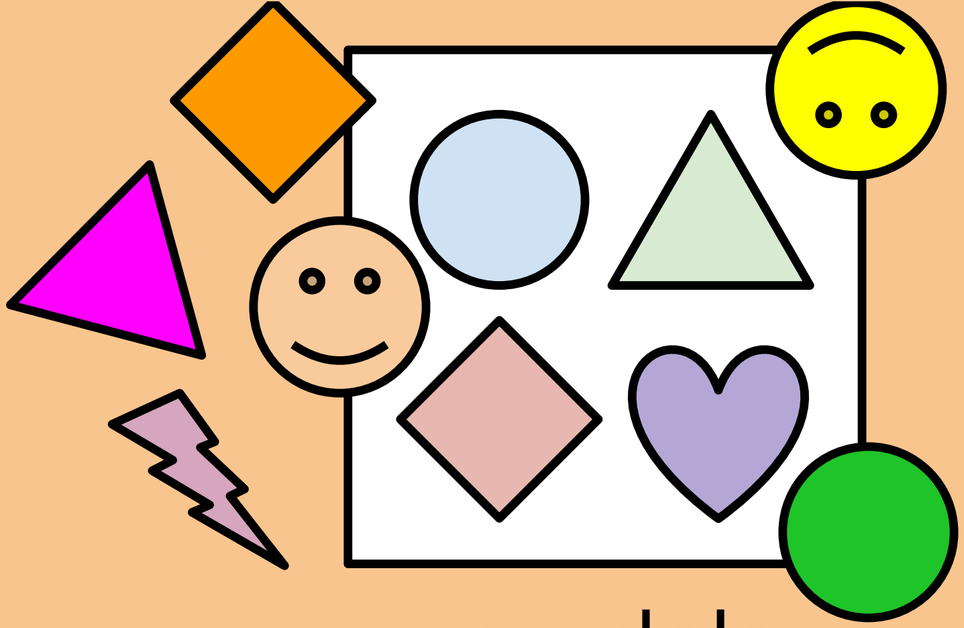
controllers



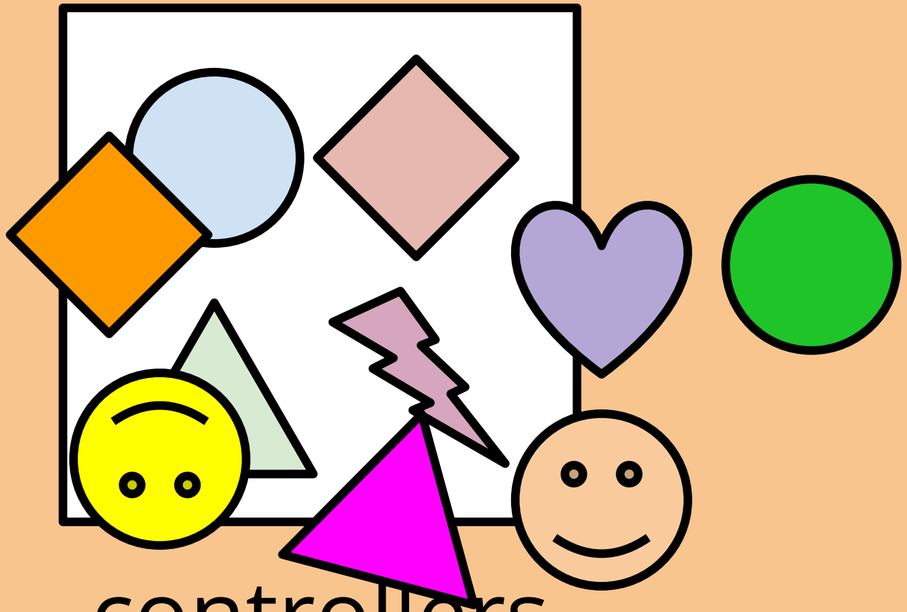
views



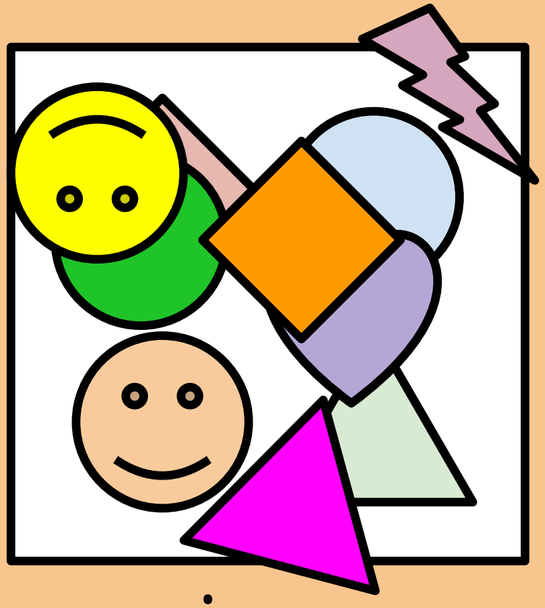
services



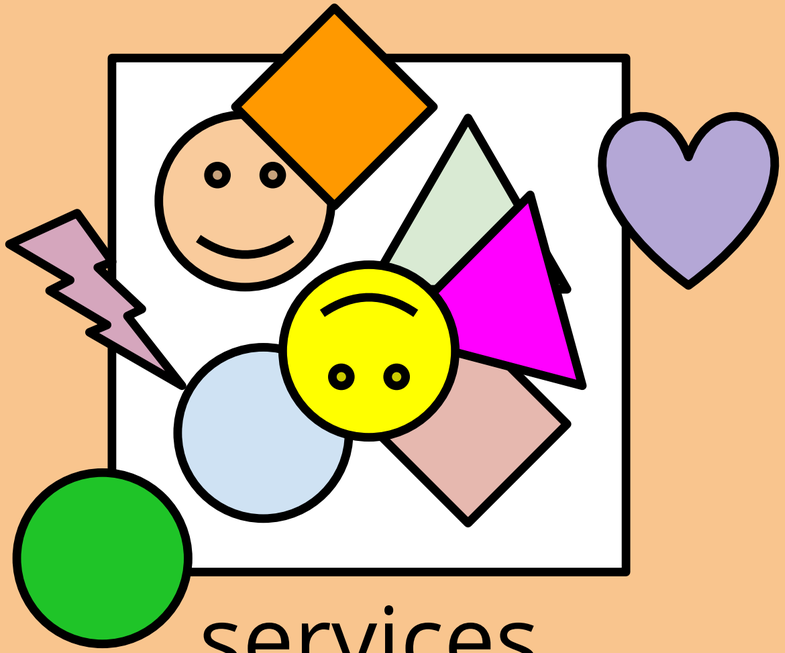
models



controllers



views



services

How do we get out of the world of the monolith?

Microservices sound hard!

How do we get out of the world of the monolith?

Microservices sound hard!

How much should I plan to extract?

How do we get out of the world of the monolith?

Microservices sound hard!

How much should I plan to extract?

What if I extract something that's too specific? Too generic?

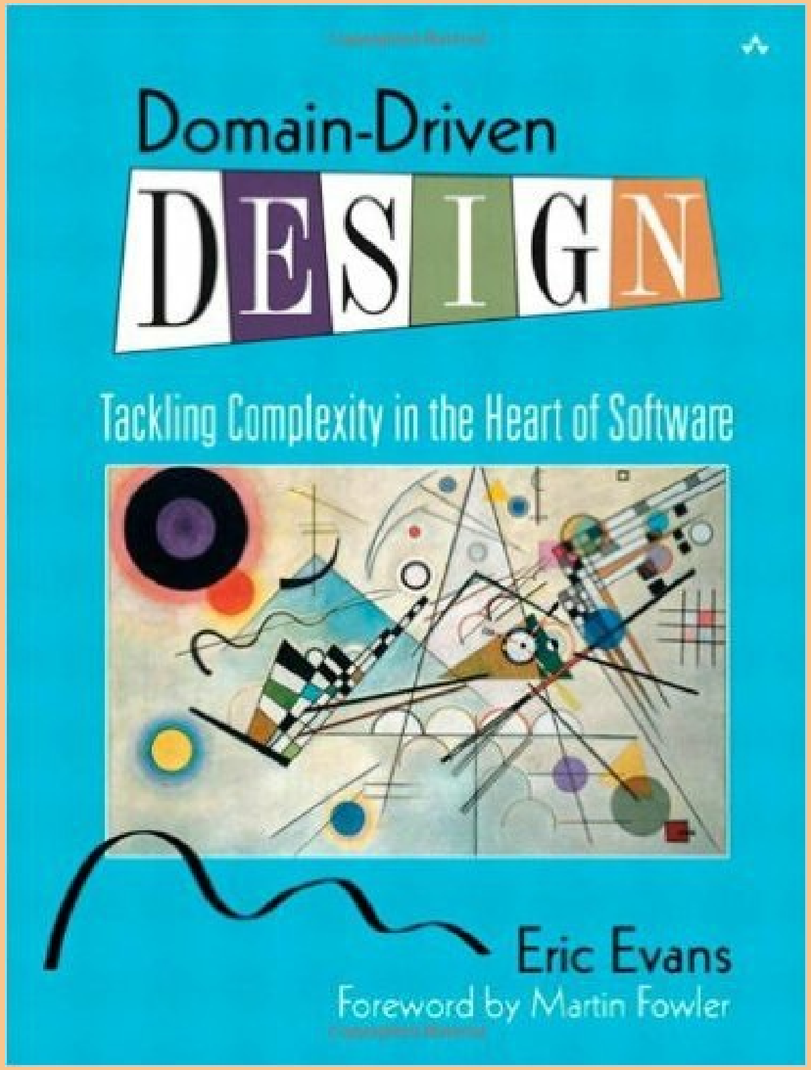
How do we get out of the world of the monolith?

Microservices sound hard!

How much should I plan to extract?

What if I extract something that's too specific? Too generic?

If only there were something to help me visualize what I need...


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler

Introducing Domain-Driven Design

Published by Eric Evans in 2003

DDD is both a set of high-level strategic design activities and concrete software patterns

Today:

We will build a **Context Map** and use it to introduce DDD concepts

We will learn some **refactoring patterns** we can use to shape our systems.

Definition!

Ubiquitous Language

A **Ubiquitous Language** is a shared set of concepts, terms and definitions between the business stakeholders and the technical staff.

Use the language to drive the design of the system.

Apply It!

Develop a Glossary

Get your business domain experts and technical staff together in a room and build a definition list of the concepts and the actions in your domain.

Glossary

Nouns - concepts (a.k.a. entities)

Verbs - actions (a.k.a. events)

A sample glossary

Driver: [Entity] A User providing driver services. S/he typically owns a Vehicle.

A sample glossary

Driver: [Entity] A User providing driver services. S/he typically owns a Vehicle.

~~Rider~~ **Passenger**: [Entity] A User seeking a ride to a specified [time-traveling] location.

A sample glossary

Driver: [Entity] A User providing driver services. S/he typically owns a Vehicle.

~~Rider~~ **Passenger**: [Entity] A User seeking a ride to a specified [time-traveling] location.

HailedDriver: [Event] A user has signaled their intent to seek out a ride.

A sample glossary

Driver: [Entity] A User providing driver services. S/he typically owns a Vehicle.

~~Rider~~ **Passenger**: [Entity] A User seeking a ride to a specified [time-traveling] location.

HailedDriver: [Event] A user has signaled their intent to seek out a ride.

ChargedCreditCard: [Event] A customer credit card has been charged for a transaction.

Apply It!

Rename concepts in code

Listen to the language, and see if the wording flows.

Renaming concepts in code is appropriate here!

Apply It!

Rename concepts in code

Listen to the language, and see if the wording flows.

Renaming concepts in code is appropriate here!

`user.request_trip` ➡ `passenger.hail_driver`

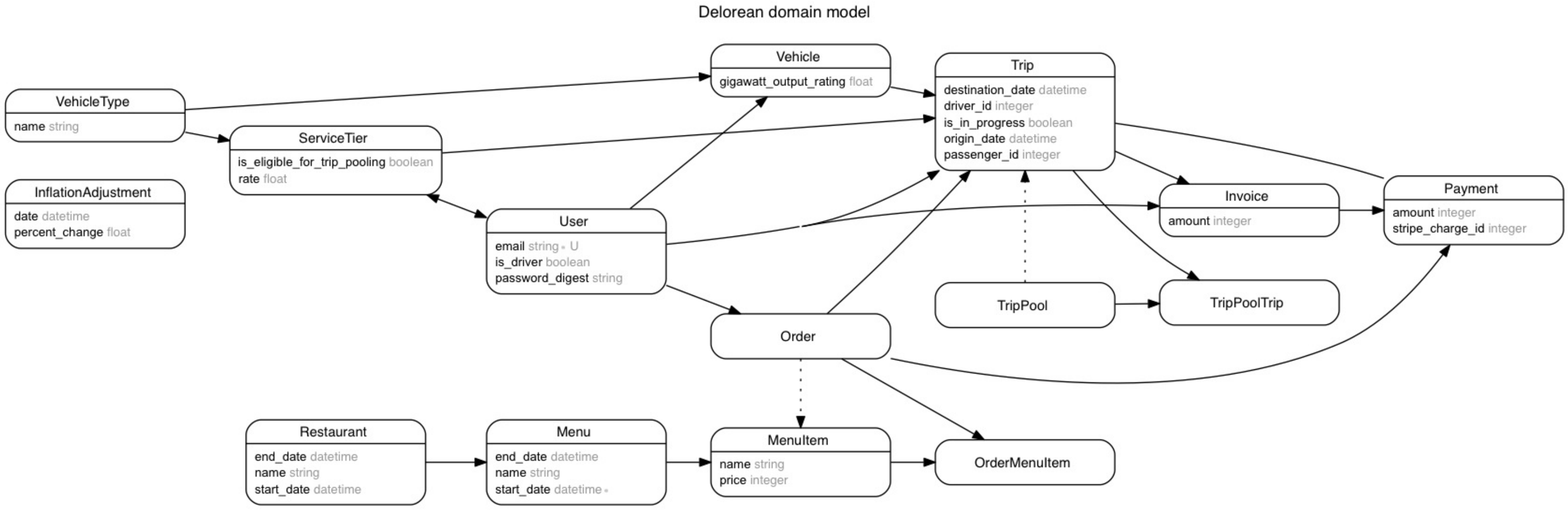
Apply It!

Visualize Your System

Let's generate an ERD diagram!

I like to generate mine with a gem like `railroady` or `rails-erd`

If you have multiple systems, do this for each system.


Delorean domain model
VehicleType
name string
InflationAdjustment
date datetime
percent_change float
ServiceTier
is_eligible_for_trip_pooling boolean
rate float
Vehicle
gigawatt_output_rating float
User
email string * U
is_driver boolean
password_digest string
Trip
destination_date datetime
driver_id integer
is_in_progress boolean
origin_date datetime
passenger_id integer
Invoice
amount integer
Payment
amount integer
stripe_charge_id integer
TripPool
TripPoolTrip
Order
Restaurant
end_date datetime
name string
start_date datetime
Menu
end_date datetime
name string
start_date datetime *
MenuItem
name string
price integer
OrderMenuItem

Yikes.

Definition!

Core domain

The **Core Domain** is the thing that your business does that makes it unique.

Definition!

Core domain

The **Core Domain** is the thing that your business does that makes it unique.

Delorean Core Domain: **Transportation**

Definition!

Supporting domains

A **Supporting Domain** (or Subdomain) are the areas of the business that play roles in making the **Core Domain** happen.

Definition!

Supporting domains

A **Supporting Domain** (or Subdomain) are the areas of the business that play roles in making the **Core Domain** happen.

- **Driver Routing** (route me from X to Y)

Definition!

Supporting domains

A **Supporting Domain** (or Subdomain) are the areas of the business that play roles in making the **Core Domain** happen.

- **Driver Routing** (route me from X to Y)
- **Financial Transactions** (charge the card, pay the driver)

Definition!

Supporting domains

A **Supporting Domain** (or Subdomain) are the areas of the business that play roles in making the **Core Domain** happen.

- **Driver Routing** (route me from X to Y)
- **Financial Transactions** (charge the card, pay the driver)
- **Optimization & Analytics** (track business metrics)

Definition!

Supporting domains

A **Supporting Domain** (or Subdomain) are the areas of the business that play roles in making the **Core Domain** happen.

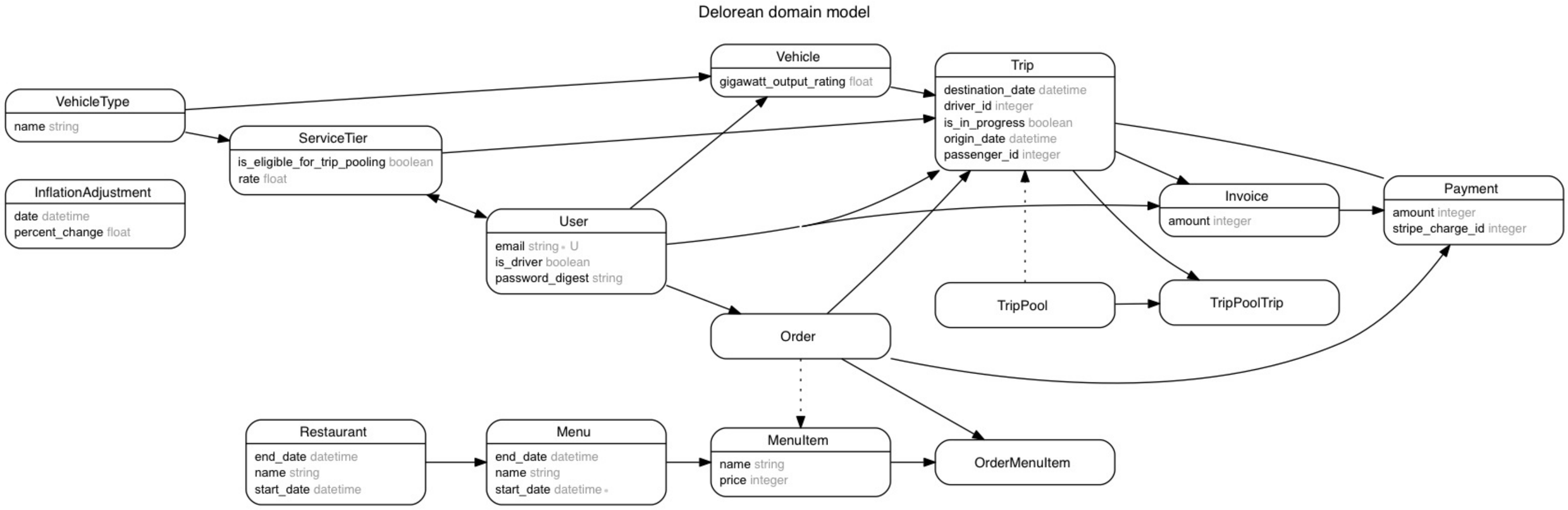
- **Driver Routing** (route me from X to Y)
- **Financial Transactions** (charge the card, pay the driver)
- **Optimization & Analytics** (track business metrics)
- **Customer Support** (keep people happy)

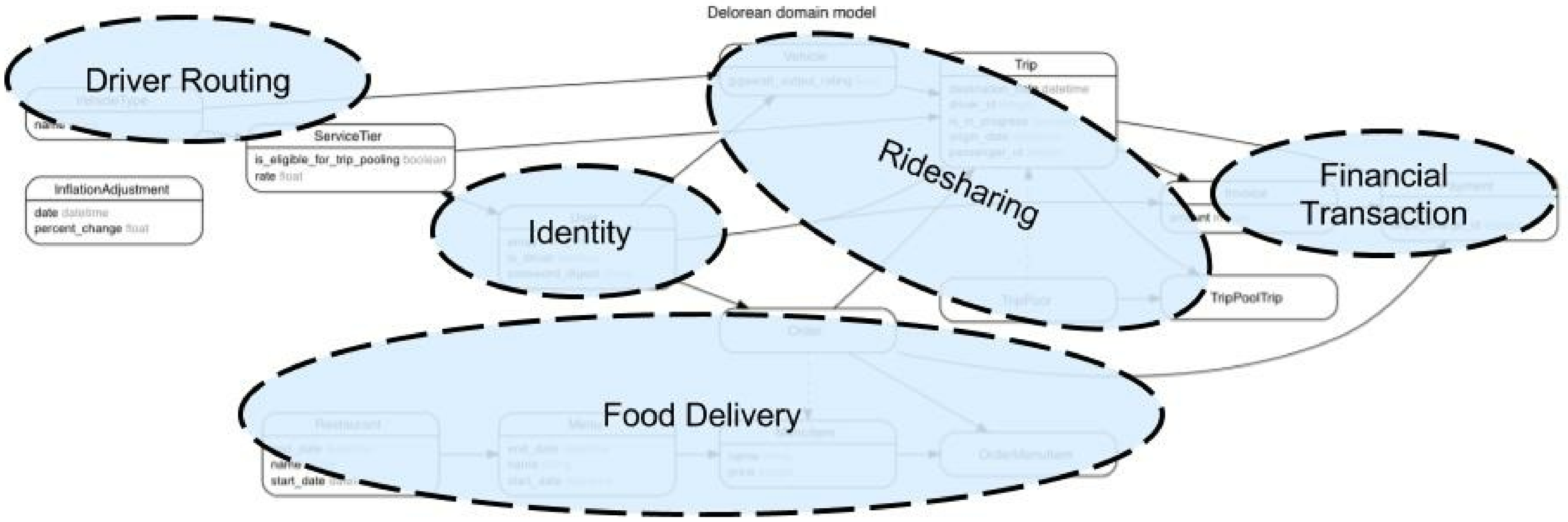
Apply It!

Discover the domains on your diagram

Look for clustered groupings.

You might discover some domains you never even thought you had!


Delorean domain model
VehicleType
name string
InflationAdjustment
date datetime
percent_change float
ServiceTier
is_eligible_for_trip_pooling boolean
rate float
Vehicle
gigawatt_output_rating float
User
email string * U
is_driver boolean
password_digest string
Trip
destination_date datetime
driver_id integer
is_in_progress boolean
origin_date datetime
passenger_id integer
Invoice
amount integer
Payment
amount integer
stripe_charge_id integer
TripPool
TripPoolTrip
Order
Restaurant
end_date datetime
name string
start_date datetime
Menu
end_date datetime
name string
start_date datetime *
MenuItem
name string
price integer
OrderMenuItem


Delorean domain model
Driver Routing
Identity
Ridesharing
Financial Transaction
Food Delivery
ServiceTier
is_eligible_for_trip_pooling boolean
rate float
InflationAdjustment
date datetime
percent_change float
Trip
TripPoolTrip

Congrats - we've got a list of domains in our system

And a rough mapping of what domain models go where.

Now let's talk boundaries

Boundaries in Rails:

1. Classes
2. Modules
3. Gems
4. Rails Engines
5. The Rails App
6. A separate app or API

Definition!

Bounded Context

- Concretely: a software system (like a codebase or running application)
- Linguistically: a delineation in your domain where concepts are "bounded", or contained

Bounded Contexts allow for precise language

Your domains may use conflicting, overloaded terms with nuances depending on context

Bounded Contexts allow for precise language

Your domains may use conflicting, overloaded terms with nuances depending on context

Bounded contexts allow these conflicting concepts to coexist

```ruby
class Trip
  def time
    # here be dragons...
  end

  def cost
    # here be dragons...
  end
end
```

Overloaded concept: Trip Time

Financial Transaction Context: Trip time is calculated from *vehicle moving time (minutes)*

Routing Context: Trip time is calculated from *total passenger minutes, including stopped time*

Overloaded concept: Trip Time

Financial Transaction Context: Trip time is calculated from *vehicle moving time (minutes)*

Routing Context: Trip time is calculated from *total passenger minutes, including stopped time*

Concepts share the same name, but have nuanced behaviors based on context!

Overloaded concept: Trip Cost

Financial Transaction Context: *How much $ the customer pays (dollars)*

Routing Context: *Trip efficiency (scalar coefficient)*

Overloaded concept: Trip Cost

Financial Transaction Context: *How much $ the customer pays (dollars)*

Routing Context: *Trip efficiency (scalar coefficient)*

Concepts share the same name, but are wildly different!

```ruby
# Overloaded concepts!
class Trip
  def time
    # Routing: total clock minutes
    # Financial: moving minutes
  end

  def cost
    # Routing: Routing AI subsystem efficiency metric
    # Financial: $$$ metric
  end
end
```

```ruby
# A little workaround?
class Trip
  def elapsed_time
  end

  def moving_time
  end

  def routing_efficiency_cost
  end

  def money_cost
  end
end
```

How could we fix it?

In DDD, we would introduce two **Bounded Contexts**:

- one for the **Financial Transaction** `Trip`
- another for the **Routing** `Trip`

These Trips can now coexist within their own software boundaries, with all their linguistic nuances intact!

Apply It!

Overlay your bounded contexts

Next up - with a different color pen or marker, draw lines around system boundaries / bounded contexts.

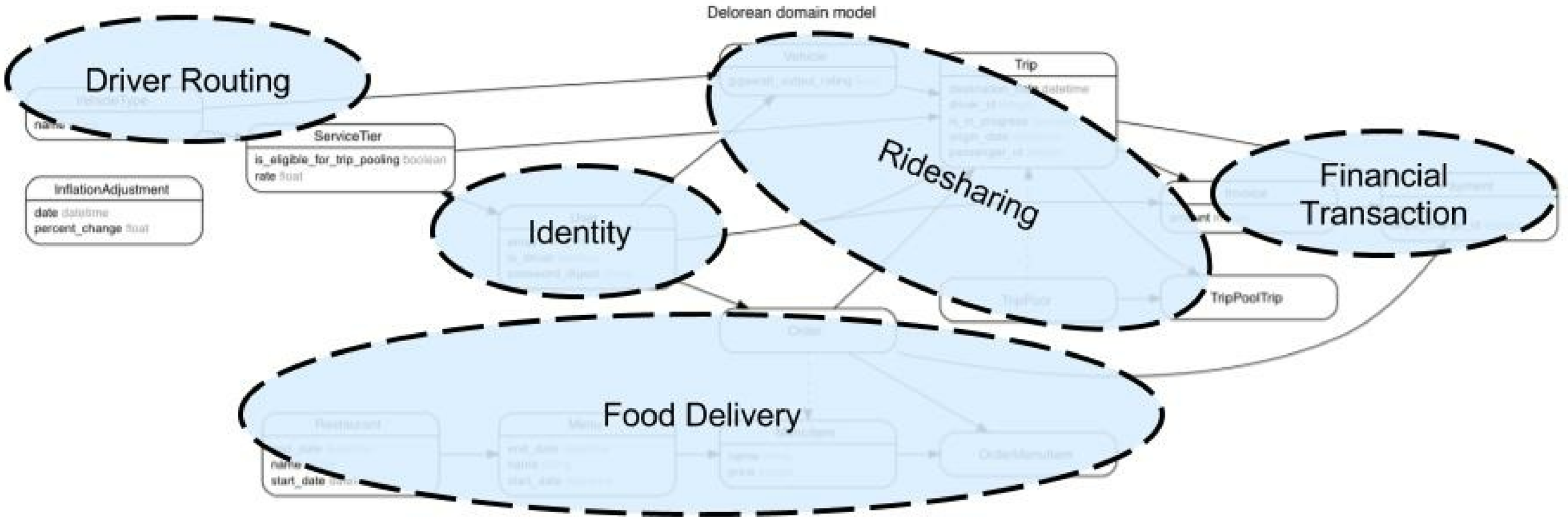
Apply It!

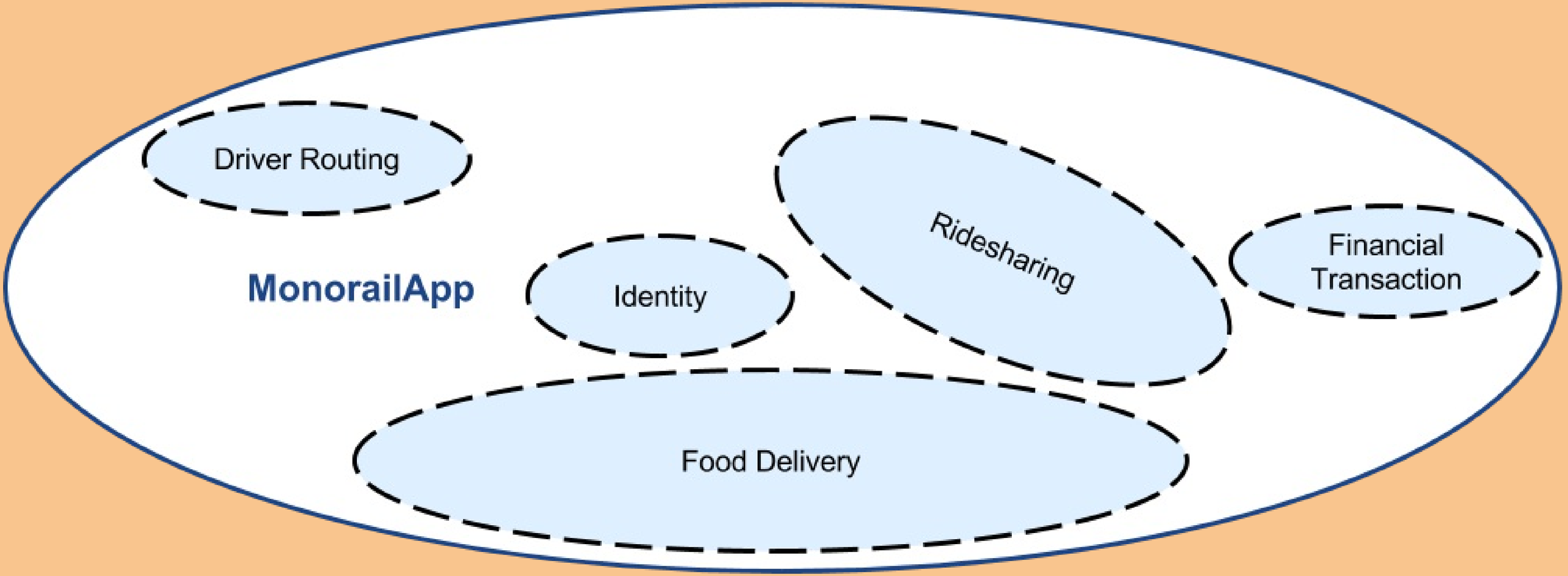
Overlay your bounded contexts

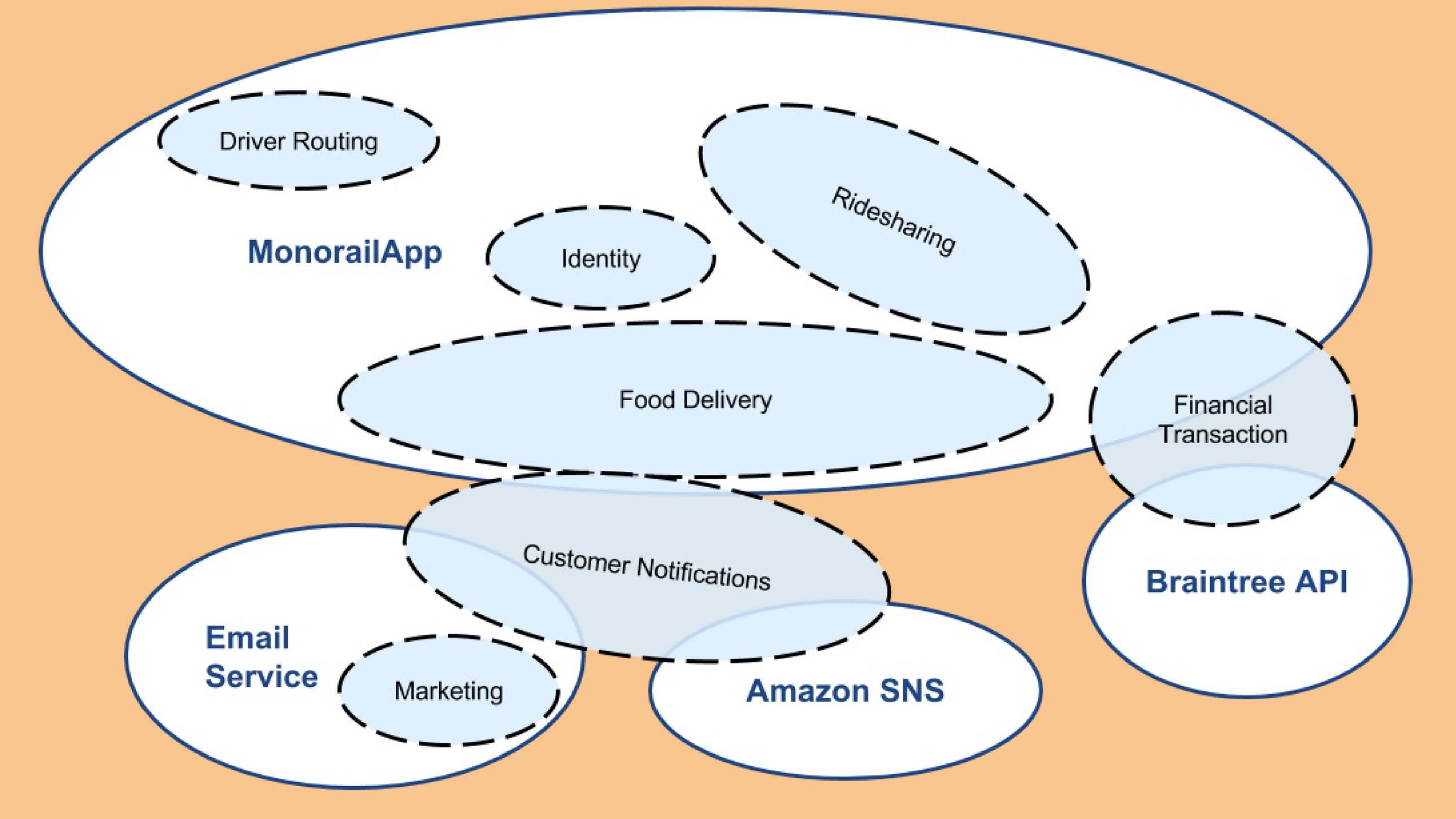
Next up - with a different color pen or marker, draw lines around system boundaries / bounded contexts.

You may also find other system boundaries like:

- External cloud providers
- Other teams' services or systems


Delorean domain model
Driver Routing
Identity
Ridesharing
Financial Transaction
Food Delivery
ServiceTier
is_eligible_for_trip_pooling boolean
rate float
InflationAdjustment
date datetime
percent_change float
Trip
TripPoolTrip

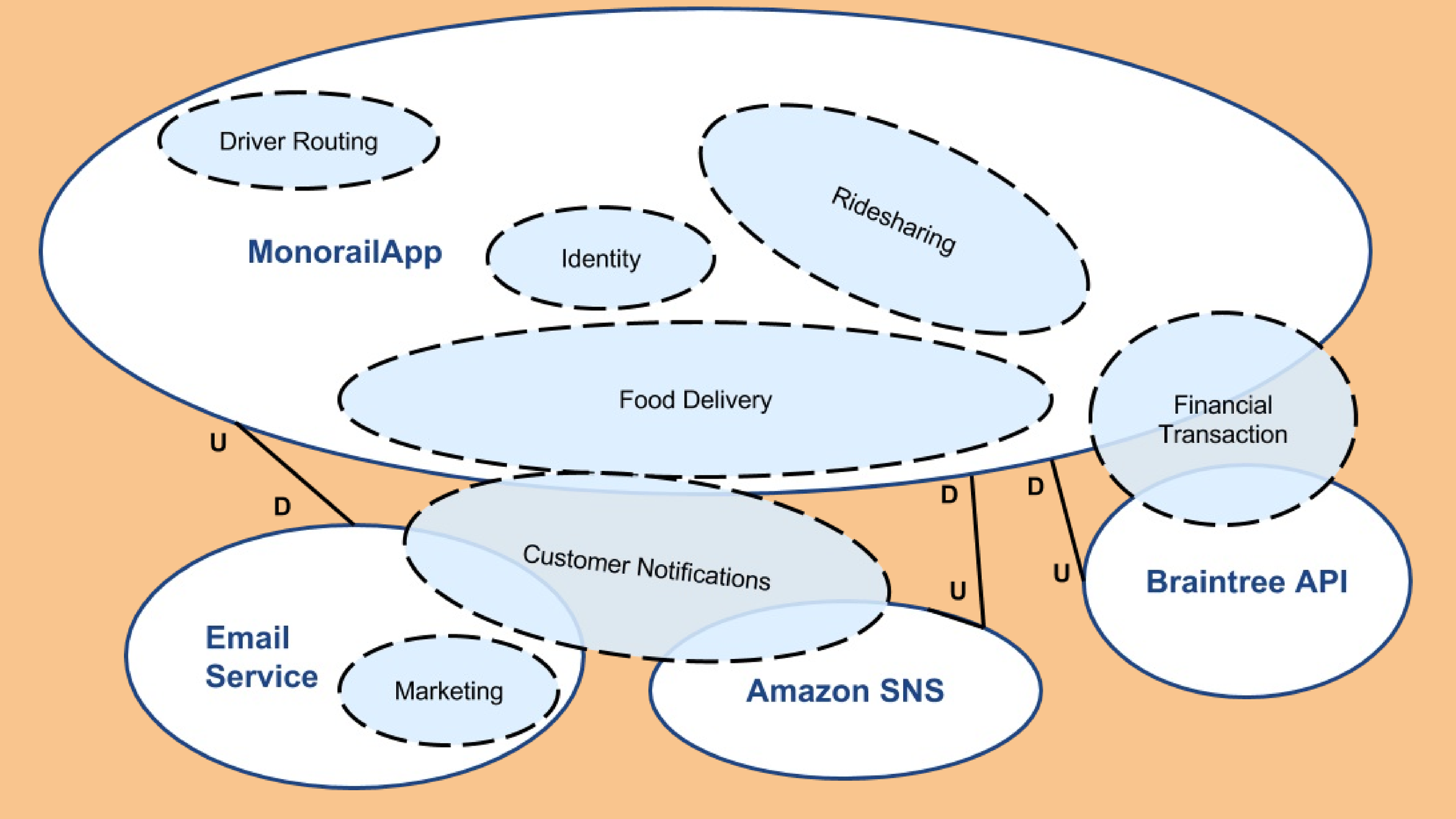

Driver Routing
MonorailApp
Identity
Ridesharing
Financial Transaction
Food Delivery


Driver Routing
MonorailApp
Identity
Ridesharing
Food Delivery
Financial Transaction
Customer Notifications
Braintree API
Email Service
Marketing
Amazon SNS

Draw out the dependencies

Draw lines indicating data flow directionality

Upstream/**D**ownstream


Driver Routing
MonorailApp
Identity
Ridesharing
Food Delivery
Financial Transaction
U
D
D
D
U
U
Customer Notifications
Email Service
Marketing
Amazon SNS
Braintree API

You just made a Context Map!

A **Context Map** gives us a place to see the current system as-is (the problem space), the strategic domains, and their dependencies.

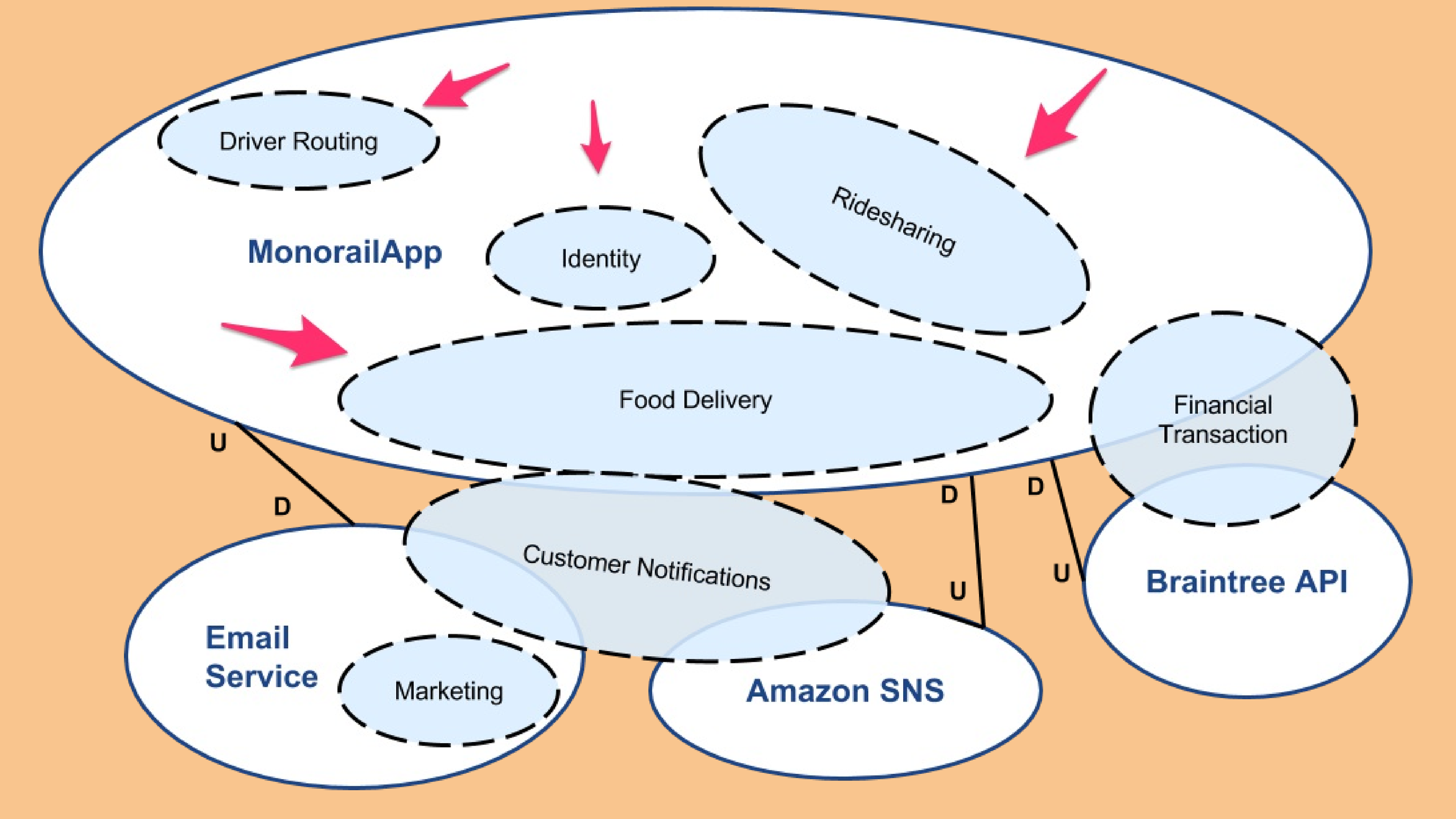
Making sense of the Context Map

We may notice a few things:

Making sense of the Context Map

We may notice a few things:

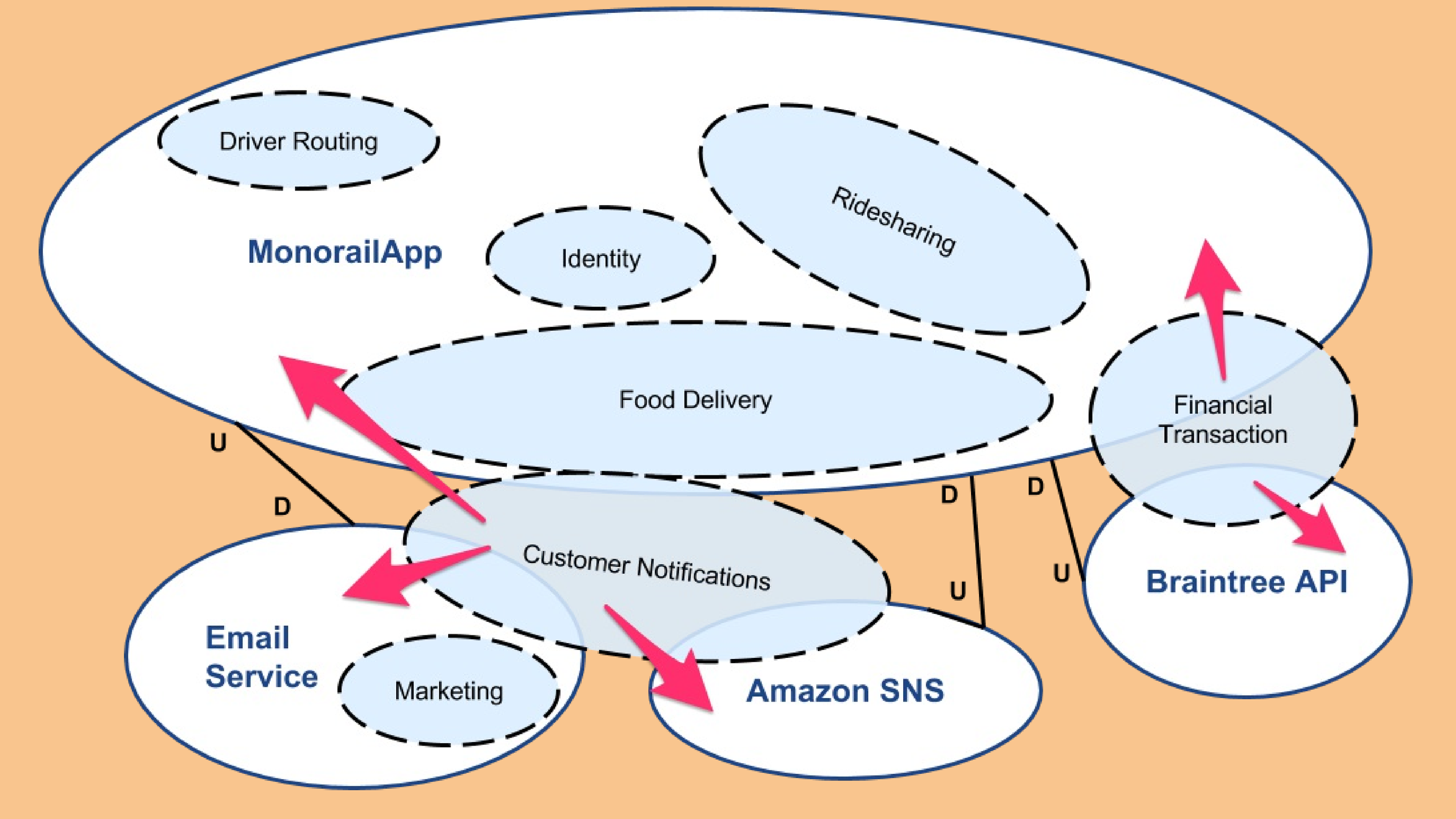
- One bounded context contains multiple sub-(supporting) domains


Driver Routing
MonorailApp
Identity
Ridesharing
Food Delivery
Financial Transaction
U
D
D
D
U
U
Customer Notifications
Email Service
Marketing
Amazon SNS
Braintree API

Making sense of the Context Map

We may notice a few things:

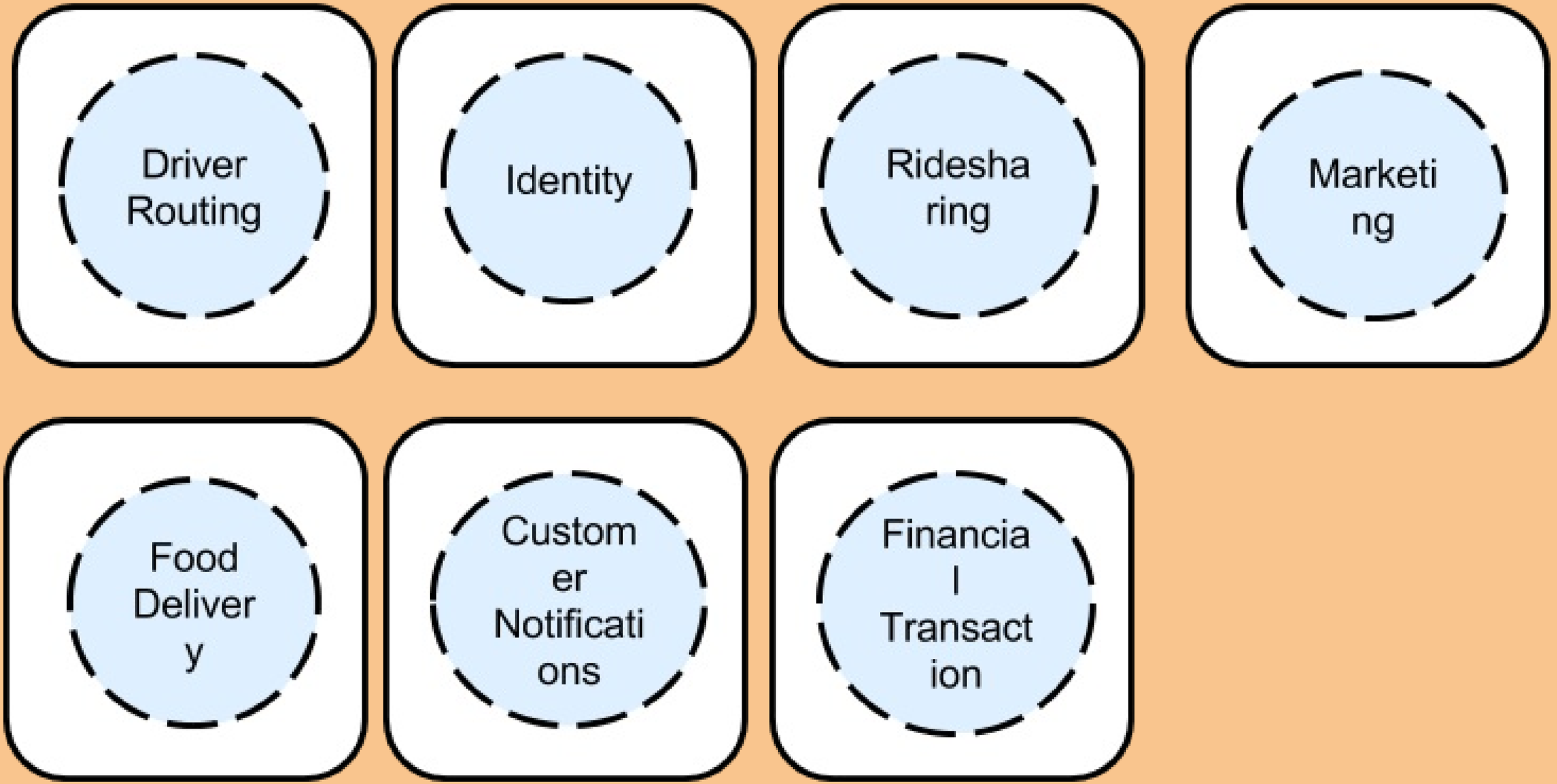
- One bounded context contains multiple sub-(supporting) domains
- Multiple bounded contexts are required to support a single domain


Driver Routing
MonorailApp
Identity
Ridesharing
Food Delivery
Financial Transaction
U
D
D
D
U
U
Customer Notifications
Email Service
Marketing
Amazon SNS
Braintree API

An Ideal Architecture

Each **Domain** should have its own **Bounded Context**

Key concept in DDD!


Driver Routing
Identity
Ridesha ring
Marketi ng
Food Deliver y
Custom er Notificati ons
Financia l Transact ion

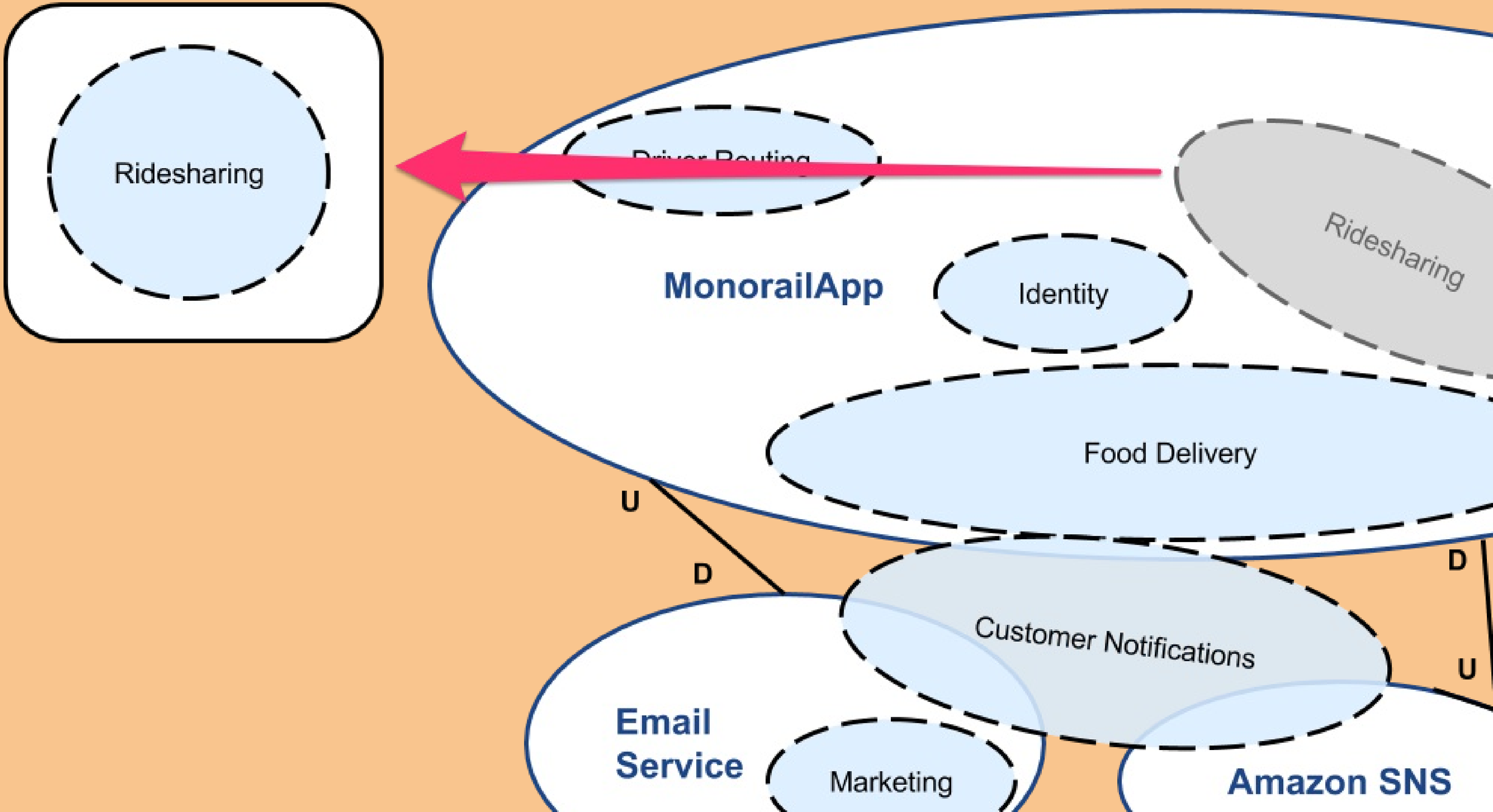
Refactoring Time

Domain-Oriented Modules & Folders

Apply It!

Break your application into domain modules

Choose one domain and make it a module.


Ridesharing
Driver Routing
Ridesharing
MonorailApp
Identity
Food Delivery
U
D
D
Customer Notifications
U
Email
Service
Marketing
Amazon SNS

```ruby
class Trip < ActiveRecord::Base
  belongs_to :vehicle
  belongs_to :passenger
  belongs_to :driver
end

class TripsController < ApplicationController
  # ...
end
```

```ruby
module Ridesharing
  class Trip < ActiveRecord::Base
    belongs_to :vehicle
    belongs_to :passenger
    belongs_to :driver
  end
end

module Ridesharing
  class TripsController < ApplicationController
    # ...
  end
end
```

```ruby
# config/routes.rb

resources :trips
```

```ruby
# config/routes.rb

namespace :ridesharing, path: '/' do
  resources :trips
end
```

```
class Invoice
  belongs_to :trip
end
```

```ruby
class Invoice
  belongs_to :trip, class_name: Ridesharing::Trip
end
```

Apply It!

Create domain-oriented folders

```
app/domains/ridesharing/trip.rb
app/domains/ridesharing/service_tier.rb
app/domains/ridesharing/vehicle.rb
app/domains/ridesharing/trips_controller.rb
app/domains/ridesharing/trips/show.html.erb
```

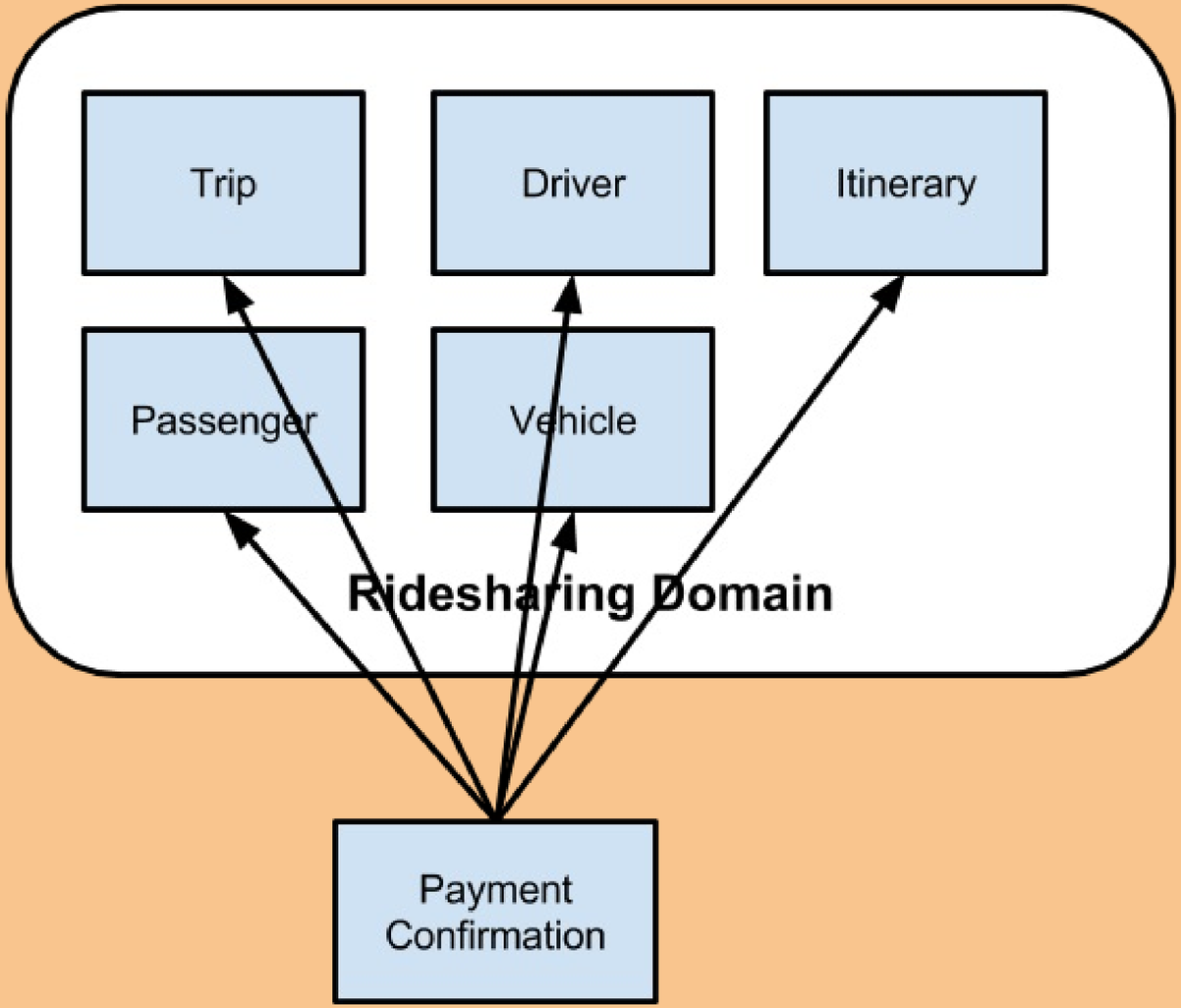
Refactoring Time

Passing around Aggregate Roots

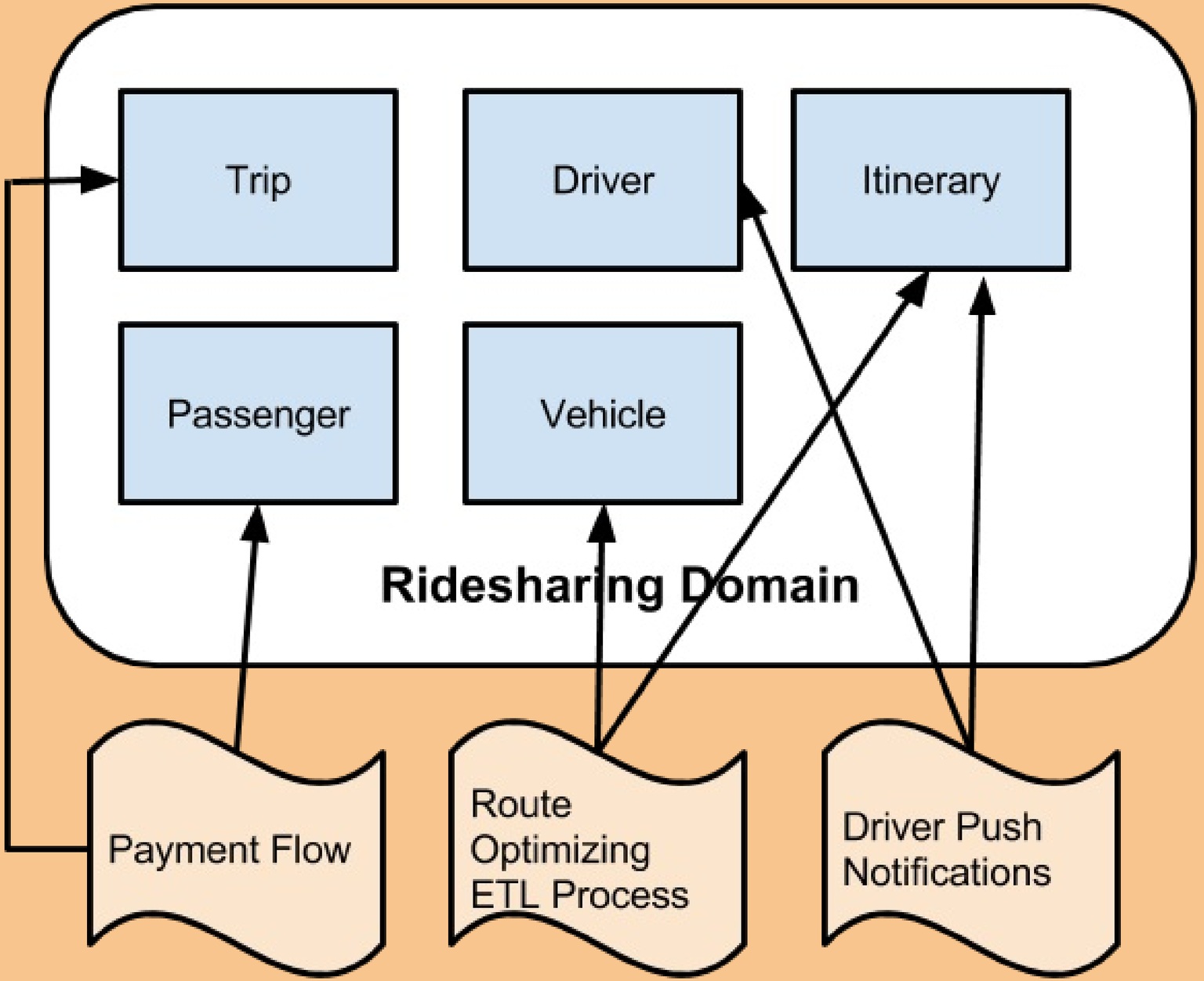
The Dreaded God Object

ActiveRecord relationships are easily abused.

Objects start knowing too much about the entire world.


Trip
Driver
Itinerary
Passenger
Vehicle
Ridesharing Domain
Payment
Confirmation

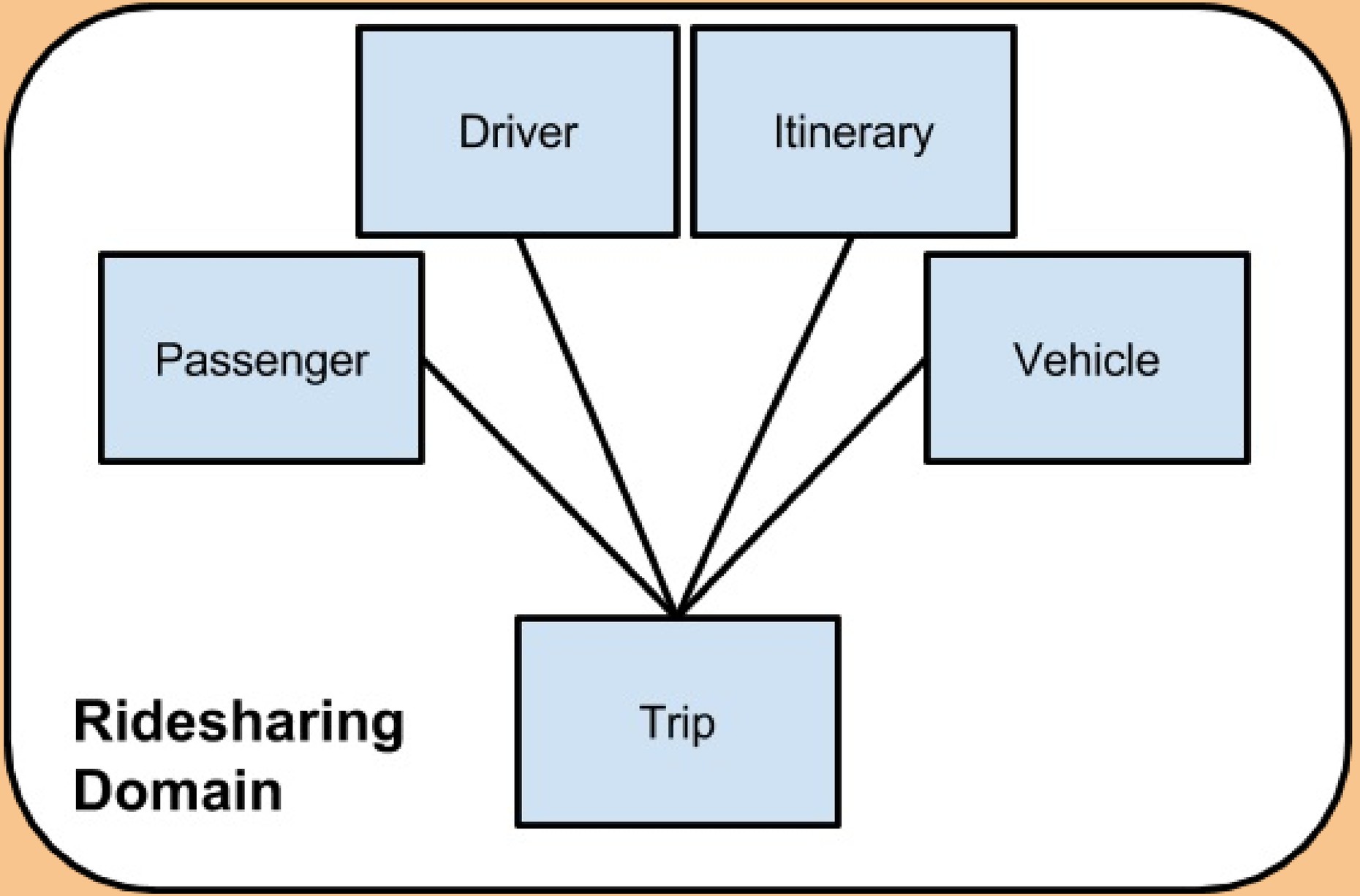
```ruby
class PaymentConfirmation
  belongs_to :trip, class_name: Ridesharing::Trip
  belongs_to :passenger, class_name: Ridesharing::Passenger
  belongs_to :credit_card
  has_many :menu_items
  belongs_to :coupon_code
  has_one :email_job
  # ad infinitum...
end
```


Trip
Driver
Itinerary
Passenger
Vehicle
Ridesharing Domain
Payment Flow
Route
Optimizing
ETL Process
Driver Push
Notifications

Definition!

Aggregate Root

Aggregate Roots are top-level domain models that reveal an object graph of related entities beneath them.


Driver
Itinerary
Passenger
Vehicle
Trip
Ridesharing Domain

Apply It!

Only expose aggregate roots

Make it a rule that each domain only exposes **Aggregate Root(s)** publicly via:

- Direct method calls
- JSON payloads
- API endpoints

Apply It!

Only expose aggregate roots

Make it a rule that each domain only exposes **Aggregate Root(s)** publicly via:

- Direct method calls
- JSON payloads
- API endpoints

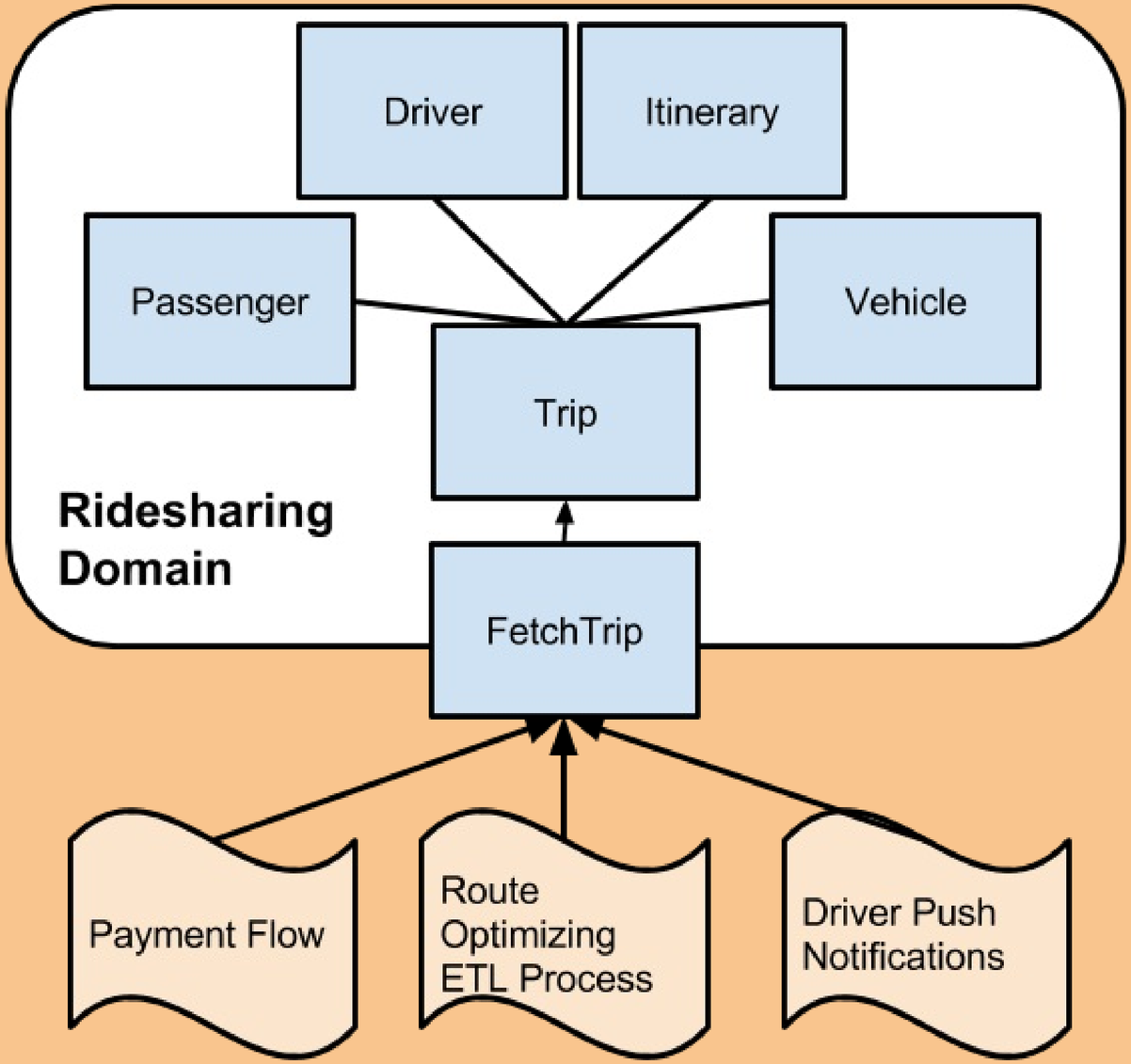
You may have multiple Aggregate Roots per domain.

Apply It!

Build service objects that provide Aggregate Roots

Break dependencies on AR relationships

Your source domain can provide a service that returns the **Aggregate Root** as a facade


Driver
Itinerary
Passenger
Vehicle
Trip
Ridesharing Domain
FetchTrip
Payment Flow
Route Optimizing ETL Process
Driver Push Notifications

```ruby
# Provide outside access to a core model
# for the Ridesharing domain
module Ridesharing
  class FetchTrip
    def call(id)
      Trip
        .includes(:passenger,
                  :driver, ...)
        .find(id)
      # Alternatively, return something non-AR
      # OpenStruct.new(trip: Trip.find(id), ...)
    end
  end
end
```

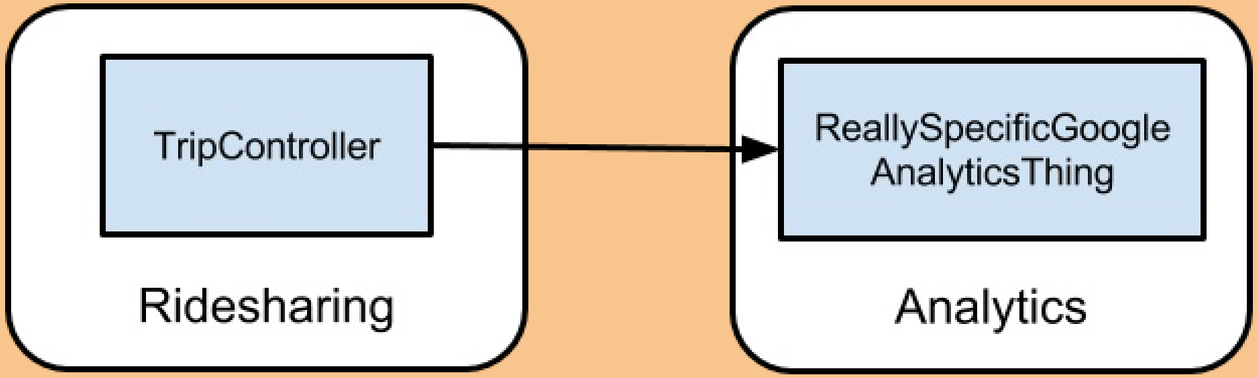
```ruby
# In the old world, we relied on AR relationships:
module FinancialTransaction
  class PaymentConfirmation
    belongs_to :trip, class_name: Ridesharing::Trip
    belongs_to :passenger, class_name: Ridesharing::Passenger
    # ...
  end
end
```

```ruby
# Now, cross-domain fetches must use the
# aggregate root service:
module FinancialTransaction
  class PaymentConfirmation
    def trip
      # Returns the Trip aggregate root
      Ridesharing::FetchTrip.new.find(payment_id)
    end
  end
end

# OLD: payment_confirmation.passenger
# NEW: payment_confirmation.trip.passenger
```

Refactoring Time

Taking advantage of events

```ruby
# Old way
module Ridesharing
  class TripController
    def create
      trip = do_something_to_create_trip(params)
      # Uh oh, this isn't a Ridesharing concern
      ReallySpecificGoogleAnalyticsThing
        .tag_manager_logging('custom_event_name',
                             ENV['GA_ID'],
                             trip)
    end
  end
end
```


TripController
Ridesharing
ReallySpecificGoogle
AnalyticsThing
Analytics

Apply It!

Publish events if you need to do something in another domain

Flip data dependency and instead broadcast that you did something.

This lowers coupling between our domains!

```ruby
# Introducing... a Domain Event Publisher
class DomainEventPublisher
  include Wisper::Publisher

  def call(event_name, *event_params)
    # Wisper then invokes registered subscriber
    # code at this point
    broadcast(event_name, *event_params)
  end
end
```

```ruby
module Ridesharing
  class TripController
    def create
      trip = do_something_to_create_trip(params)

      # Here, we fire an event, but don't care
      # what actually happens next
      DomainEventPublisher.new
        .call(:trip_created, trip.id)
    end
  end
end
```

Apply It!

Every bounded context has its own event handler

Now we add an event handler for each domain, so it knows how to handle incoming events.

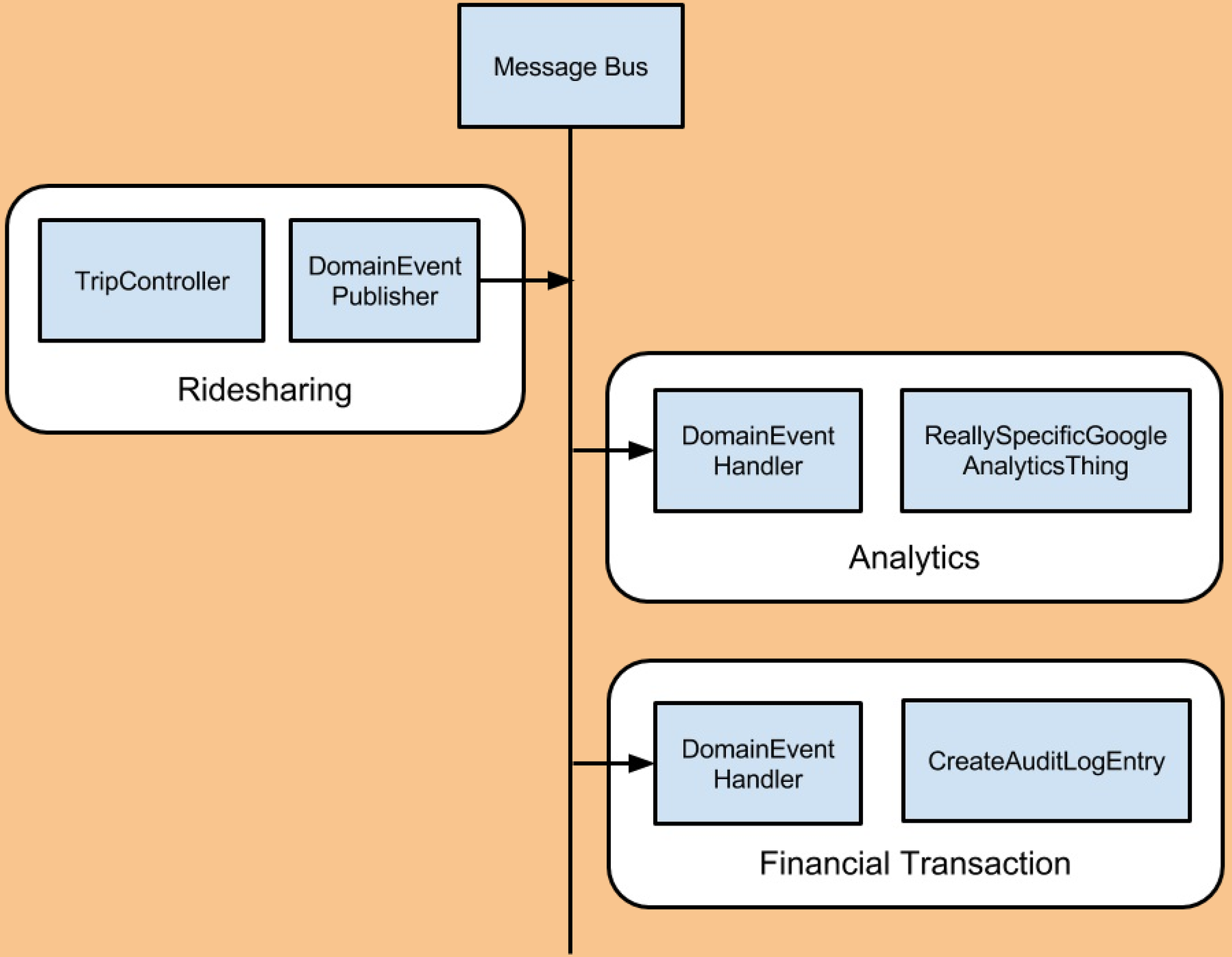
This handler will then dispatch the relevant side effects for each event, through a Command object.

```ruby
# Handles relevant domain events. Dispatches to
# Command objects that perform side effects.
module Analytics
  class DomainEventHandler

    # Method name is invoked based on the name of the
    # message. This method is invoked in response to
    # the `trip_created` event.
    def self.trip_created(trip_id)
      # handle the action here, delegate out to a
      # service/command.
      LogTripCreated.new.call(trip_id)
    end
  end
end
```

```ruby
# Hook up the handler (with a subscription) to
# the DomainEventPublisher

# config/initializers/domain_events.rb
Wisper.subscribe(Analytics::DomainEventHandler,
    scope: :DomainEventPublisher)
```

```ruby
# Meanwhile back in the Analytics domain, we
# wrap the specific GA call in a Command/service object.
module Analytics
  class LogTripCreated
    def call(params)
      ReallySpecificGoogleAnalyticsThing
        .fire_event('custom_event_name',
                    ENV['GA_ID'],
                    params['trip'])
    end
  end
end
```

```ruby
# Different domains can opt to subscribe to the same
# events!
module FinancialTransaction
  class DomainEventHandler
    def self.trip_created(trip_id)
      CreateTaxAuditLogEntry.new.call(trip_id)
      DeductGiftCardAmount.new.call(trip_id)
    end
  end
end
```


Message Bus
TripController
DomainEvent Publisher
Ridesharing
DomainEvent Handler
ReallySpecificGoogle AnalyticsThing
Analytics
DomainEvent Handler
CreateAuditLogEntry
Financial Transaction

Apply It!

Now make it truly asynchronous with ActiveJob!

This has been synchronous so far - everything happens within the same web request thread.

Wisper can hook into ActiveJob to truly process your events asynchronously in a worker queue.

Everything after `publish` now is processed by a worker!

```ruby
# Gemfile
gem 'wisper-activejob'

# config/initializers/domain_events.rb
Wisper.subscribe(Analytics::DomainEventHandler,
    scope: :DomainEventPublisher, async: true)
```

Using a message queue

Instead of using Wisper, publish a RabbitMQ event!

Each domain's event handlers are run as subscribers to an exchange topic.

Stitch Fix's Pwwka is an excellent Rails-RabbitMQ pub/sub implementation. You can also use Sneakers.

Moving beyond the basics

Advanced topics

Apply It!

Sharing entities between contexts

Shared Kernel - namespace shared models in a common module or namespace:

`User` ➡ `Common::User`

Apply It!

Sharing entities between contexts

Shared Kernel - namespace shared models in a common module or namespace:

`User` ➡ `Common::User`

This can later be packaged up in a gem if your systems are spread out

Apply It!

When you have one model that needs to belong in two domains

Sometimes, you have a concept that needs to be broken up. How can we get these concepts codified in different domains?

Concept: **Anti-Corruption Layer**

Apply It!

When you have one model that needs to belong in two domains

Sometimes, you have a concept that needs to be broken up. How can we get these concepts codified in different domains?

Concept: **Anti-Corruption Layer**

We will introduce a notion of an `Adapter` that maps an external concept to our internal concept.

```ruby
# Legacy, complicated domain model
module Common
  class Trip < ActiveRecord::Base
    def elapsed_time; end
    def moving_time; end
    def routing_efficiency_cost; end
    def money_cost; end
  end
end

# Nice, expressive domain model
module Routing
  class Trip < Struct.new(:cost, :time)
  end
end
```

```ruby
# Convert between a Common::Trip to a Routing::Trip
module Routing
  class TripAdapter
    def convert(external_trip)
      attrs = mapping_from(external_trip)
      Trip.new(mapped_attrs[:cost], mapped_attrs[:time])
    end

    def mapping_from(external_trip)
      { cost: external_trip.routing_efficiency_cost,
        time: external_trip.elapsed_time }
    end
  end
end
```

```ruby
module Routing
  class TripRepository
    def self.find_by!(*params)
      external_trip = ::Common::Trip.find_by!(*params)
      TripAdapter.new.convert(external_trip)
    end
  end
end

# Module code now, instead of calling ::Common::Trip.find_by!,
# calls Routing::TripRepository.find_by!
```

Toward the future

Where Next?

Progressive refactoring

1. Domain-oriented folders, to…
2. Rails engines, to…
3. Rails microservices with a shared AR gem and a message queue, to…
4. Fully-decoupled, polyglot microservices

Each of these evolutions is simply modeling a bounded context with stronger seams!

This may work for you if...

DDD works well if:

- You have a complex domain that needs linguistic precision.
- You work in a very large (perhaps distributed) team
- You're open to experimentation and have buy-in from your Product Owner.
- The whole team's open to trying it out (not a lone wolf).
- Other teams, too.

Eric Leclerc @eleclercca · 15 Oct 2016

Replying to @stevelounsbury

Eh hum, should be titled "write your Rails like it's a java project" :-p

1 1

Steve Lounsbury @stevelounsbury · 15 Oct 2016

For a small app, it's overkill. For a large app (hundreds/thousands of models) with a large team of devs, it's valuable.

1

Eric Leclerc @eleclercca · 16 Oct 2016

suret, that's the great thing about ruby devs, they haven't forgotten how to stay simple when simple is enough #OldJavaDude

Know when to stop!

Consider backing out if:

- You're getting that feeling of Overdesign™
- The weight of maintaining abstractions is a heavy burden
- Other teams unhappy or lost

Know when to stop!

Consider backing out if:

- You're getting that feeling of Overdesign™
- The weight of maintaining abstractions is a heavy burden
- Other teams unhappy or lost

Don't pressure yourself to follow DDD patterns "by the book".

In summary

Discovered the **Domains** in our business

Built a **Context Map** to see strategic insights

Investigated some **refactoring patterns** to shape our systems.

Thanks!

Github: andrewhao

Twitter: @andrewhao

Email: andrew@carbonfive.com

Carbon Five

Credits & Prior Art

- Evans, Eric. Domain-Driven Design: Tackling Complexity in the Heart of Software.
- Gorodinski, Lev. "Sub-domains and Bounded Contexts in Domain-Driven Design (DDD)".
- Hagemann, Stephan. Component-Based Rails Applications.
- Parnas, D.L. "On the Criteria To Be Used in Decomposing Systems into Modules".
- Vernon, Vaughan. Implementing Domain-Driven Design.
- W. P. Stevens ; G. J. Myers ; L. L. Constantine. "Structured Design" - IBM Systems Journal, Vol 13 Issue 2, 1974.