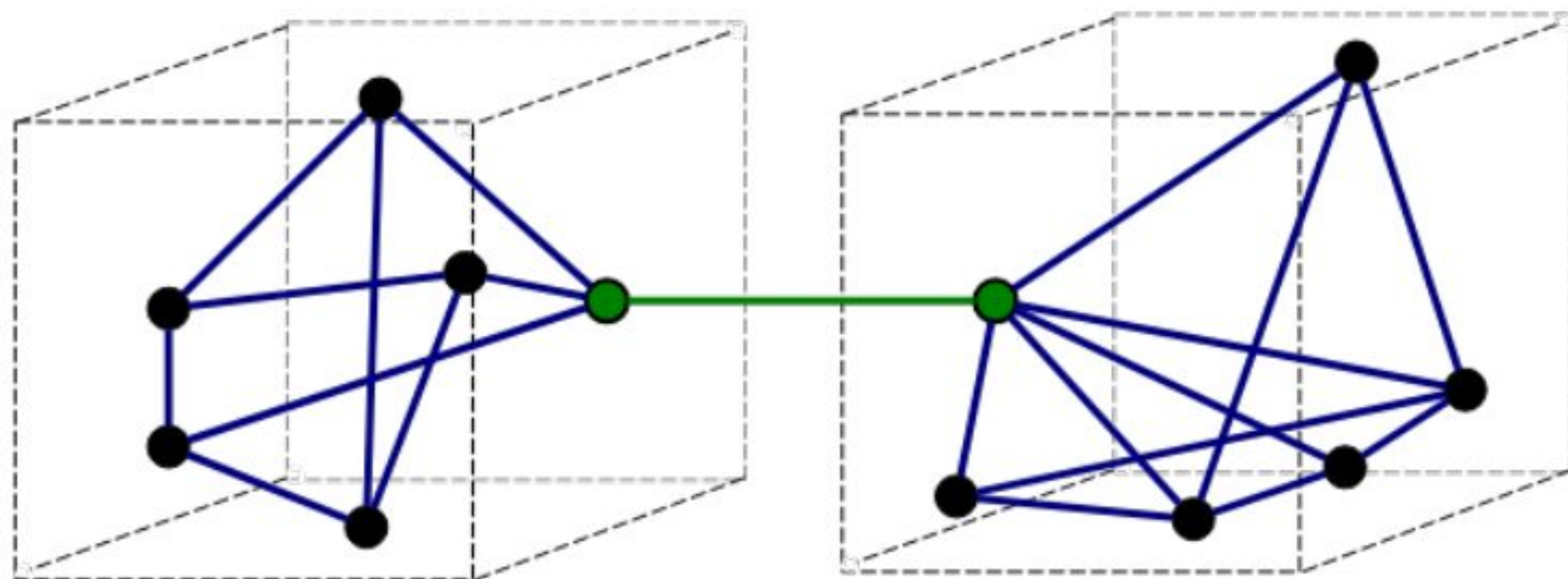
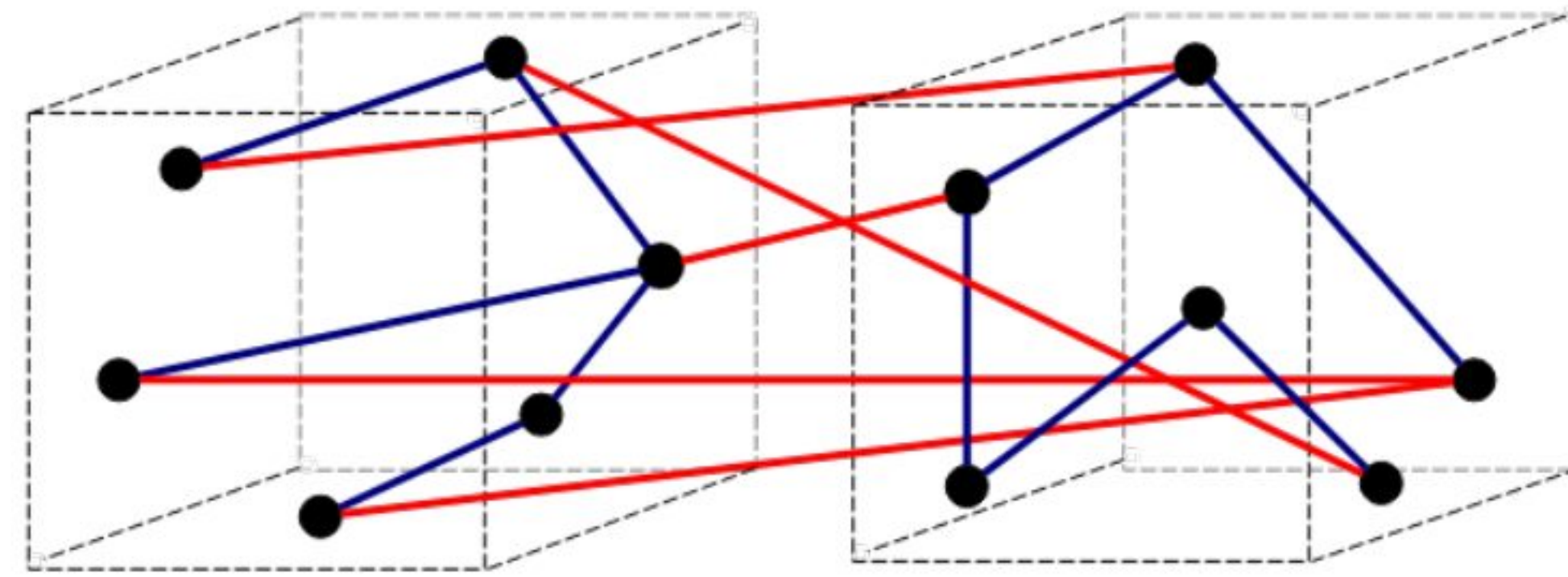




a) Good (loose coupling, high cohesion)

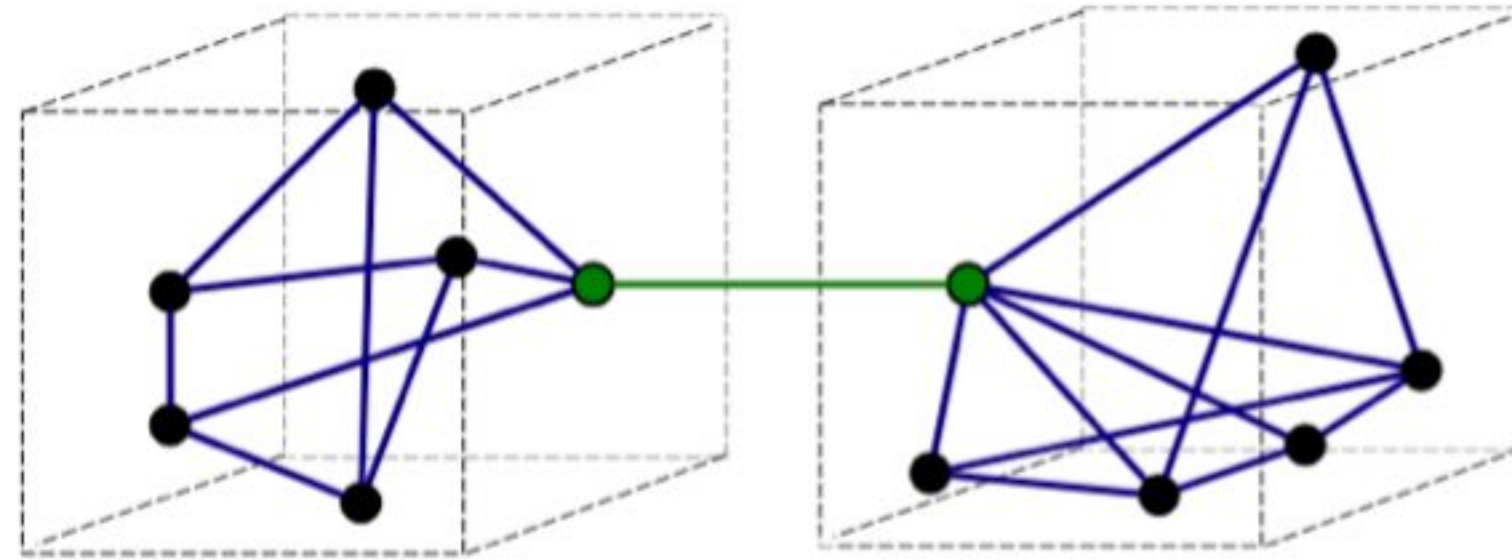


b) Bad (high coupling, low cohesion)

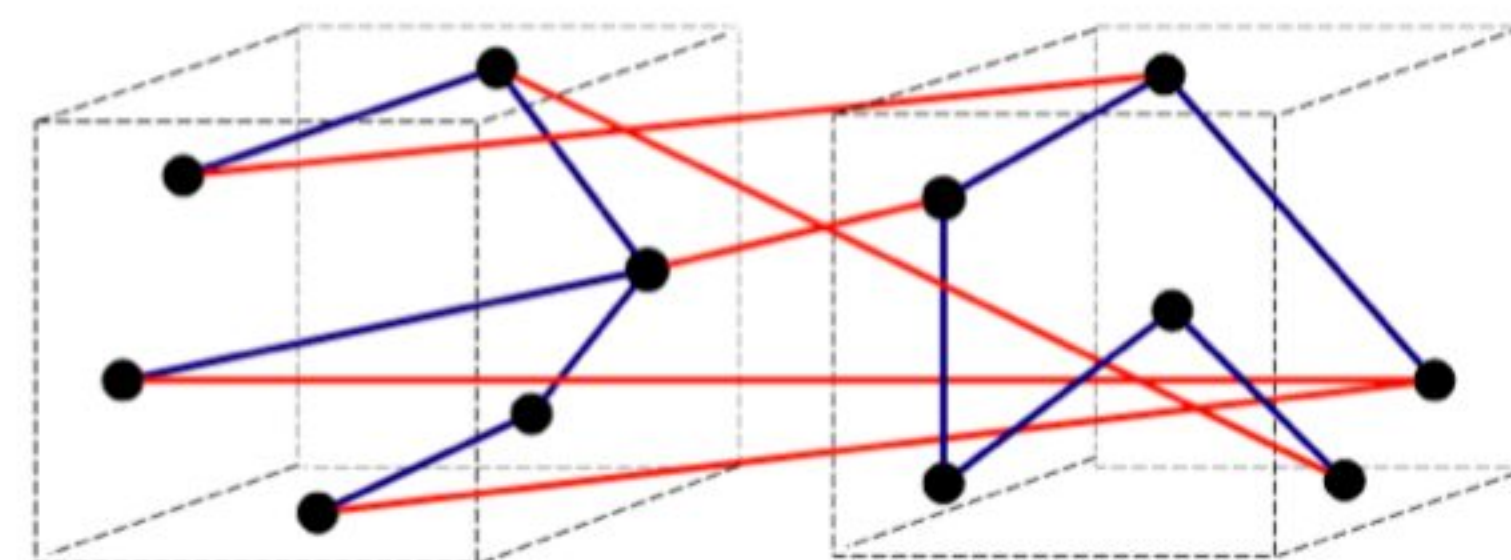
RUBY SAN FRANCISCO MEETUP

BY KONSTANTIN GREDESKOUL

DESIGN FOR LOOSE COUPLING



a) Good (loose coupling, high cohesion)



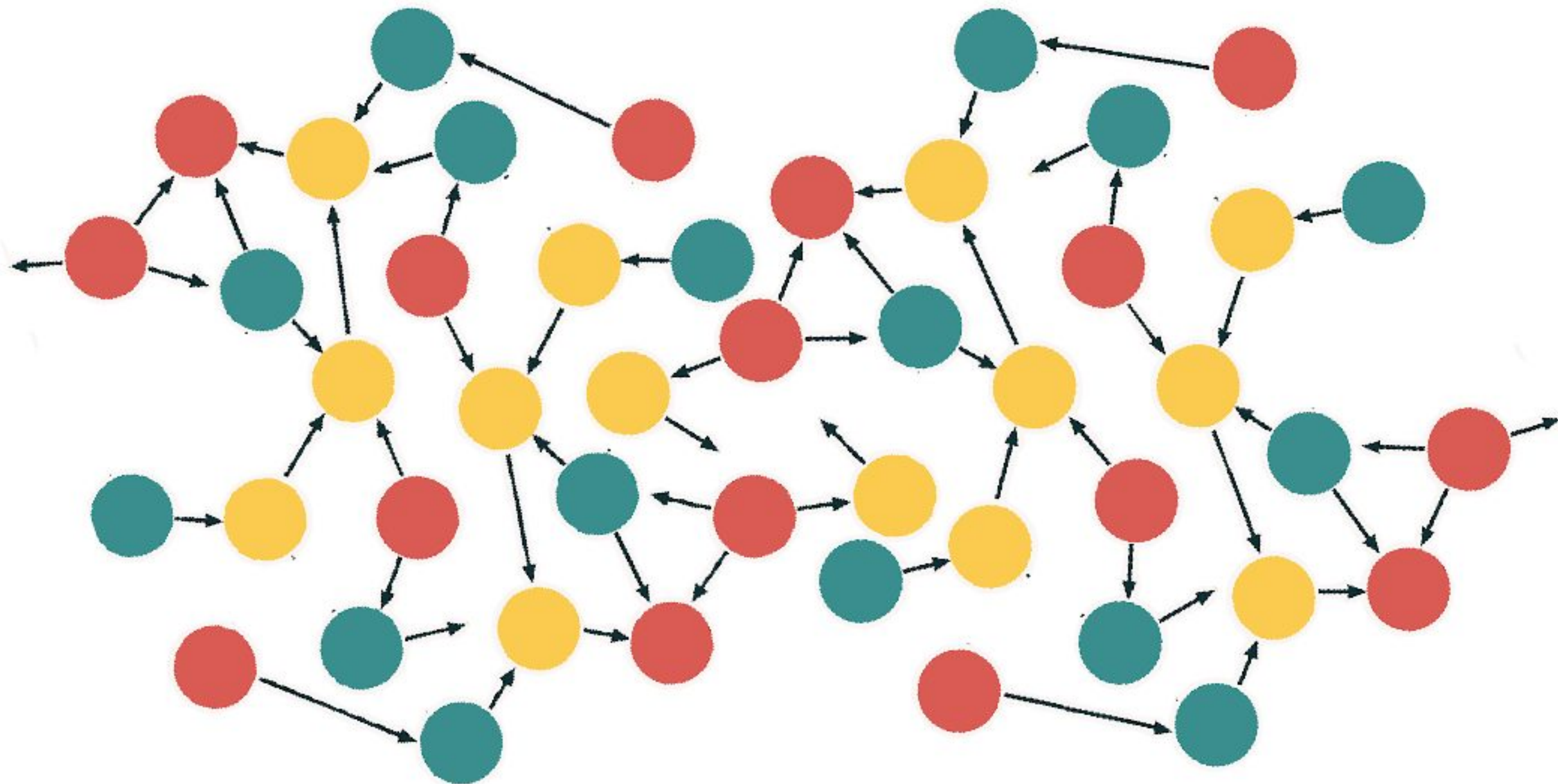
b) Bad (high coupling, low cohesion)

IN SOFTWARE DEVELOPMENT, **"TIGHT COUPLING"** REFERS TO A SITUATION WHERE COMPONENTS WITHIN A SYSTEM ARE **HIGHLY DEPENDENT ON EACH OTHER**, MEANING **CHANGES TO ONE COMPONENT CAN SIGNIFICANTLY IMPACT OTHER COMPONENTS**,

WHILE **"LOOSE COUPLING"** INDICATES THAT COMPONENTS ARE **MORE INDEPENDENT**, ALLOWING **MODIFICATIONS TO ONE COMPONENT WITHOUT MAJOR REPERCUSSIONS ON OTHERS**, MAKING THE **SYSTEM OVERALL MORE FLEXIBLE AND MAINTAINABLE**

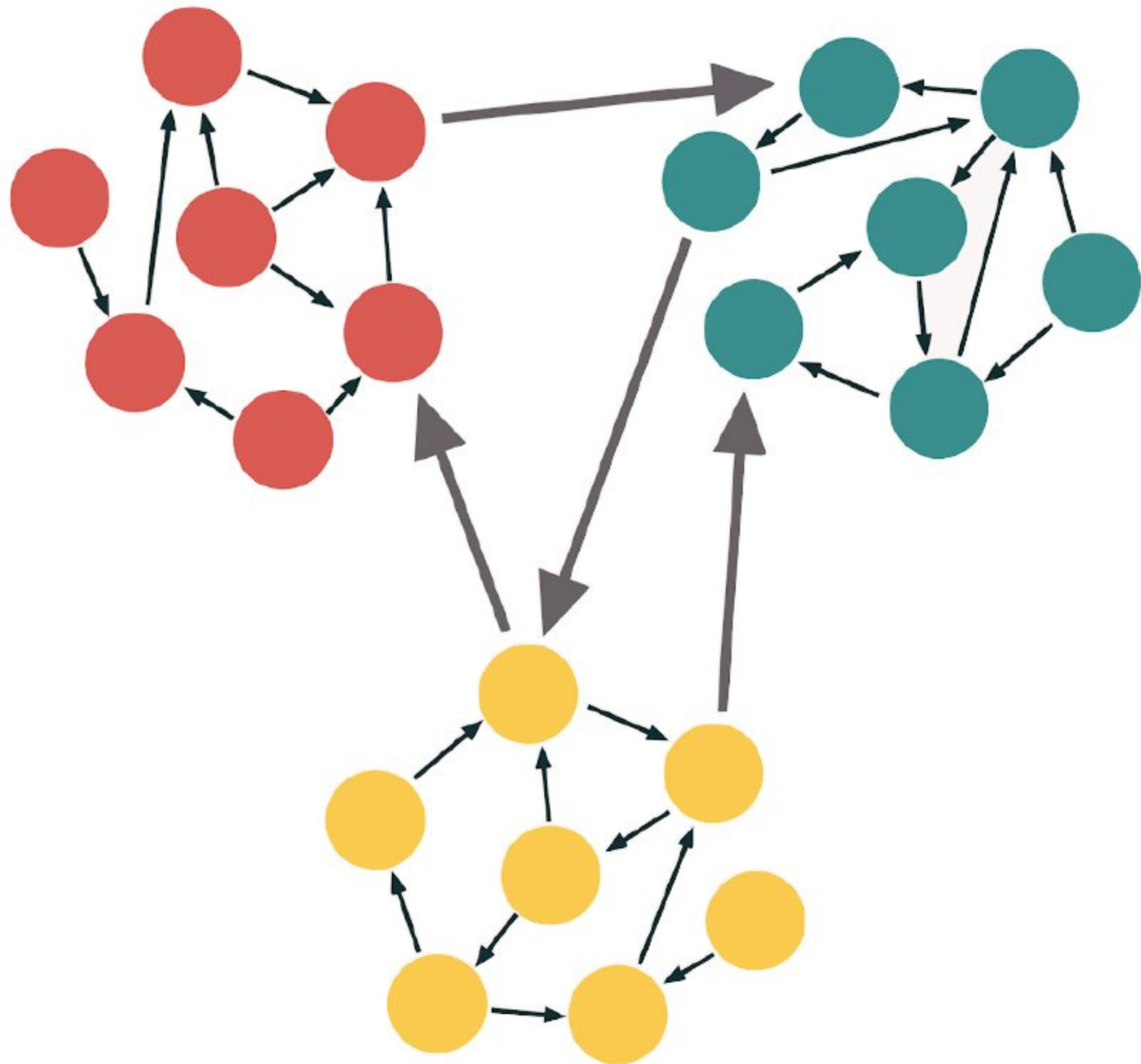
WITH THE DEFINITION OUT OF THE WAY, LOOK IT AT THIS WAY....

TIGHT COUPLING IS A NATURAL START FOR MOST APPLICATIONS



- ▶ When you begin a new application, Rails or not, you will likely start by tightly coupling your code.
- ▶ Surface area is still small and heavy abstractions are not yet necessary.
- ▶ But things get harder with exponential complexity if the tangled spaghetti is not modularized.

WHAT IS THE ALTERNATIVE?



- ▶ You design your application as a set of **interoperating modules** (or sub-systems) with clear boundaries and enforcement capabilities.
- ▶ Start thinking about it when the app no longer fits in any individual head (or you can think ahead knowing this awaits you).
- ▶ Or, when many team members start to think that there are "too many cooks in the kitchen"...
- ▶ Or, when the developer productivity screeches to a halt, or onboarding takes more than six months.

BUT WHY DO WE CARE ABOUT THIS?

WHAT DO WE GET FROM IT, AND WHAT DO WE LOSE?

TO ANSWER THAT, WE SHOULD STEP BACK A BIT

- ▶ I would imagine that many folks in the audience write software for a living (or perhaps aspire to) — software that's currently running in production, generating revenue, and which any number of other businesses (B2B) or end-users (B2C), or both (B2B2C) depend on.

- ▶ Question!

What makes software "good"? Or more specifically, and at the same time more generally, **how do you know the software meets the needs of the business?**

THE ANSWER IS DIFFERENT DEPENDING ON WHO YOU ASK

- ▶ CEO or Head of Product:
 - ▶ How well does the software adapt to ever changing requirements?
 - ▶ How quickly can we prototype, test and launch new ideas as product features?
 - ▶ Is the software reliable, scalable, and secure?
- ▶ CFO & Investors:
 - ▶ How much did it cost (and does it cost now) to maintain and advance this application? Is it massively expensive?
 - ▶ One way of measuring this is finding the cost of running the service for a single user.

WHAT DOES THE BUSINESS ULTIMATELY WANT FROM THE APP? AND THE ENGINEERING ORGANIZATION?

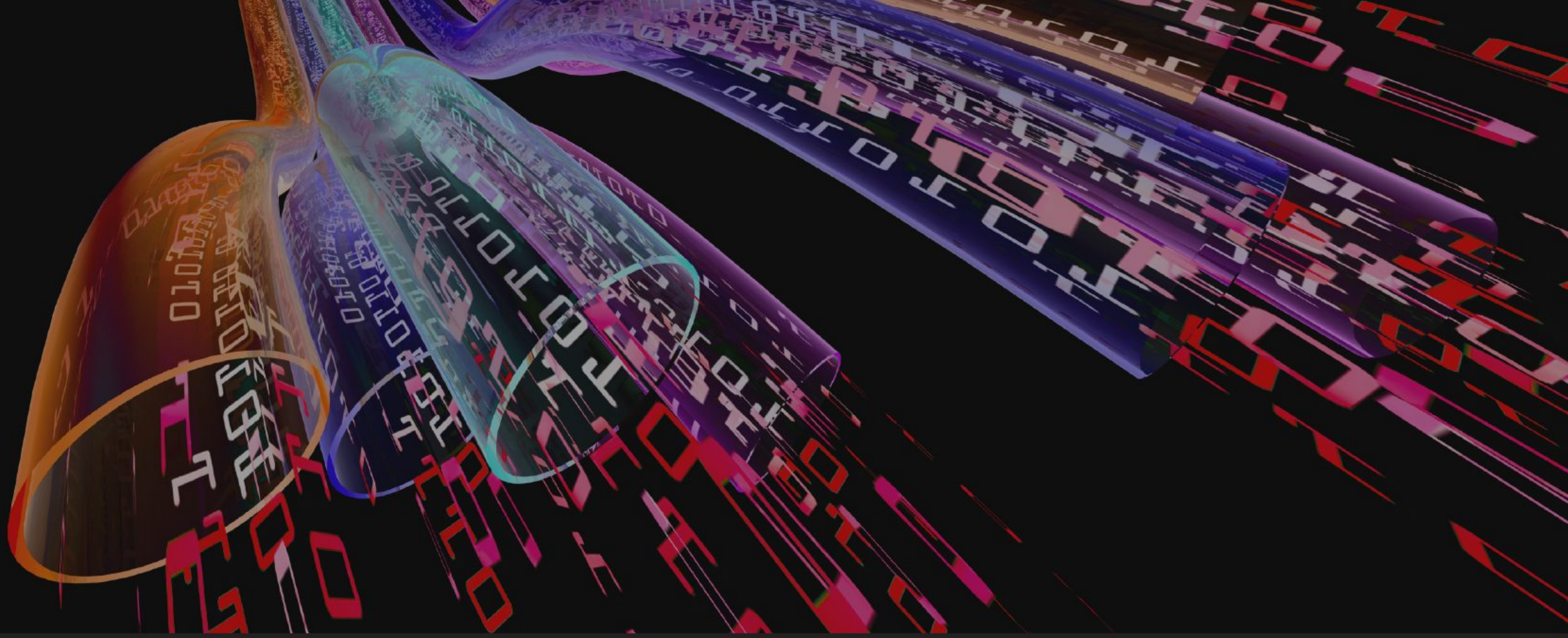
- ▶ That the app works correctly (low defect rate)
- ▶ It works within the agreed SLA (99.99% uptime) and performance SLA
- ▶ It supports rapid prototyping (which can be messy)
- ▶ It supports fast feature development, including tests.
- ▶ Metrics above should not decline with application or team size.

OVER THE LIFECYCLE OF THE APP'S LIFE, SOMETHING ELSE ARISES: THE TECHNICAL DEBT

- ▶ As the software launches, it moves into a new phase of maintenance and enhancements (feature development).
 - ▶ It is believed that most businesses that thrive and have long-lasting applications, end up with the cost of maintenance being orders of magnitude larger than the initial cost of building and launching the green-field app.
- ▶ As the codebase grows, becomes messy, large, often not fully tested, it starts to carry a **heavy technical debt**.
- ▶ This debt often **prevents the business from moving FAST** when the app is large **UNLESS** the technical debt is being continually addressed.

SO THE QUESTION THEREFORE BECOMES...

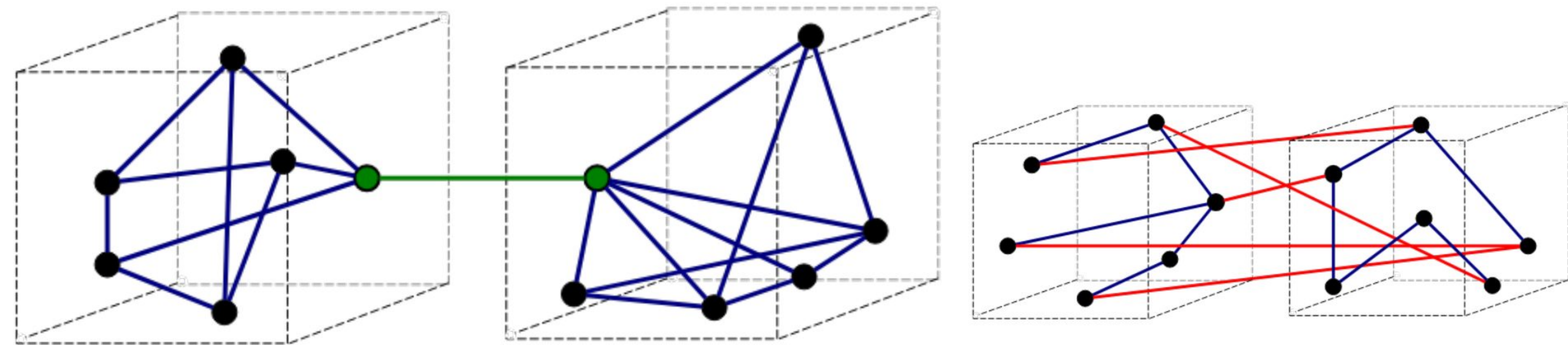
HOW DO WE GO FROM A SMALL MESSY APP WITH SUPER-HIGH PRODUCTIVITY TO A VERY LARGE APP WITHOUT LOOSING TOO MUCH DEV VELOCITY?



ONE OF POSSIBLE ANSWERS COULD BE THIS:

SOFTWARE DESIGN THAT'S FORWARD-LOOKING, AS IN IT ANTICIPATES THE FUTURE, BUT DOES NOT PRESCRIBE IT.

AND WHAT'S THE BEST WAY TO DESIGN SOFTWARE SO THAT IT ANTICIPATES FUTURE (CHANGES), WITHOUT PREMATURELY OPTIMIZING OR PRESCRIBING SOLUTIONS TO YET UNKNOWN PROBLEMS?



ONE OF THE SIMPLEST CONCEPTS TO GRASP,
BUT ALSO HARDEST TO DO CONSISTENTLY AND WELL

LOOSE COUPLING

BUT WHAT ARE THE DIFFERENT WAYS WE CAN ACTUALLY WRITE LOOSELY COUPLED CODE?

1. BY MODULARIZING OUR APPLICATION AND USING **FACADE** PATTERN TO INTERFACE WITH OTHER MODULES (**PACKWERK**)
2. BY USING **OBSERVABLE** PATTERN, DECOUPLING THE SUBSCRIBER FROM EVENT PRODUCER... EG USING EVENTS.
3. BY USING **INVERSION OF CONTROL** OR DEPENDENCY INJECTION PATTERN, AVOIDING DIRECT INSTANTIATION

WHERE DO WE GO FROM HERE?

- ▶ We'll sketch out loosely coupled modules for a typical e-comm marketplace (based on a real application)
- ▶ We'll look at several patterns that participate in loose coupling
 - ▶ Such as **Facade** or a **Library** (Gem)
 - ▶ **Inversion of Control**
- ▶ Look at how **Packwerk** can be applied incrementally to gradually migrate a very large Rails app into well-defined modules.

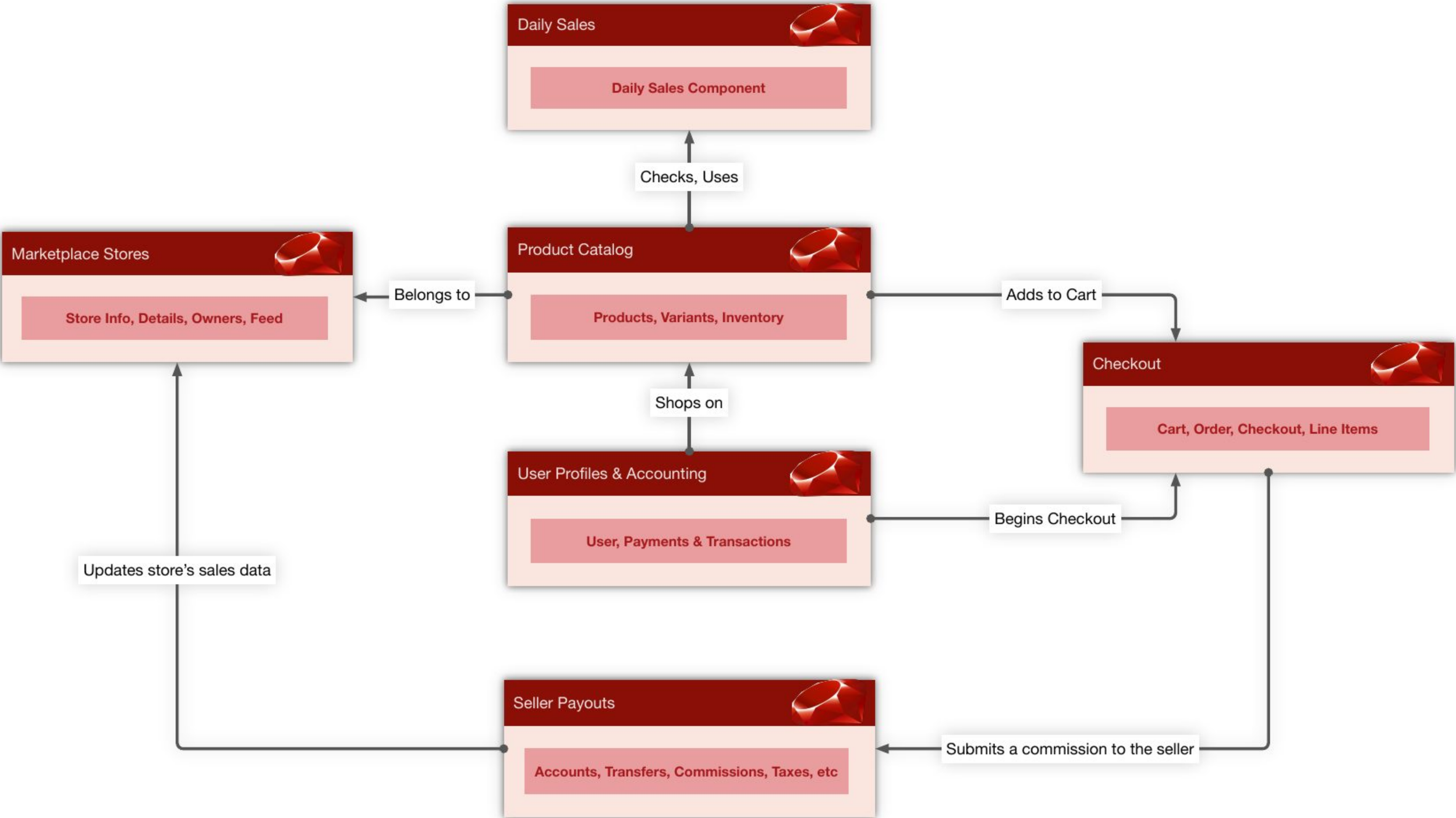


E-COMM MARKETPLACE

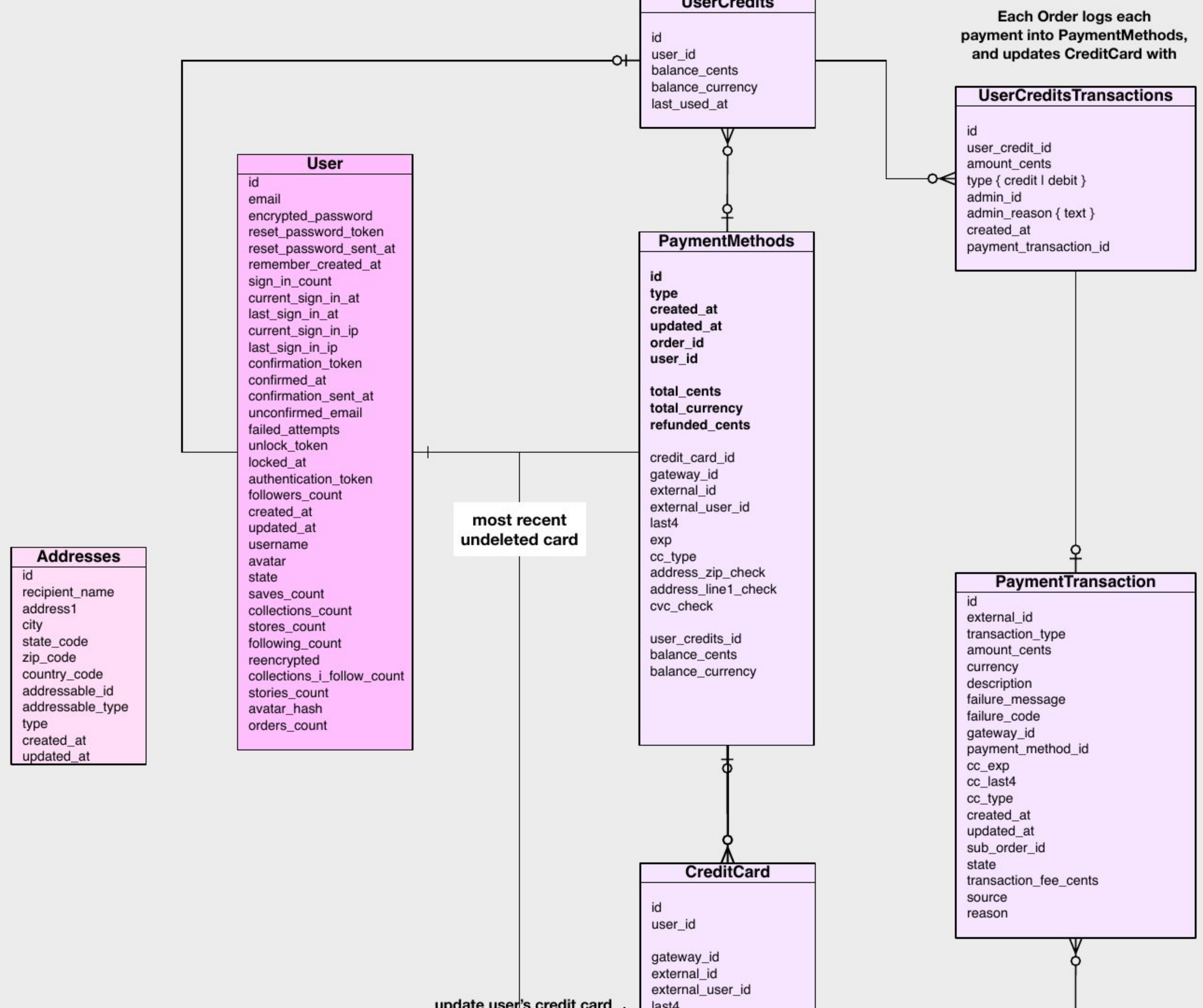
EXAMPLE: E-COMMERCE MARKETPLACE (THINK ETSY, ETC)

- ▶ Let's look at how would be break out our marketplace data model into separate modules that can be linked together.
- ▶ We want to combine classes that are tightly coupled together, but separate them from other parts of the system that's not directly related.
- ▶ We'll break up the marketplace Data Model into six modules for this example, but your mileage may vary.

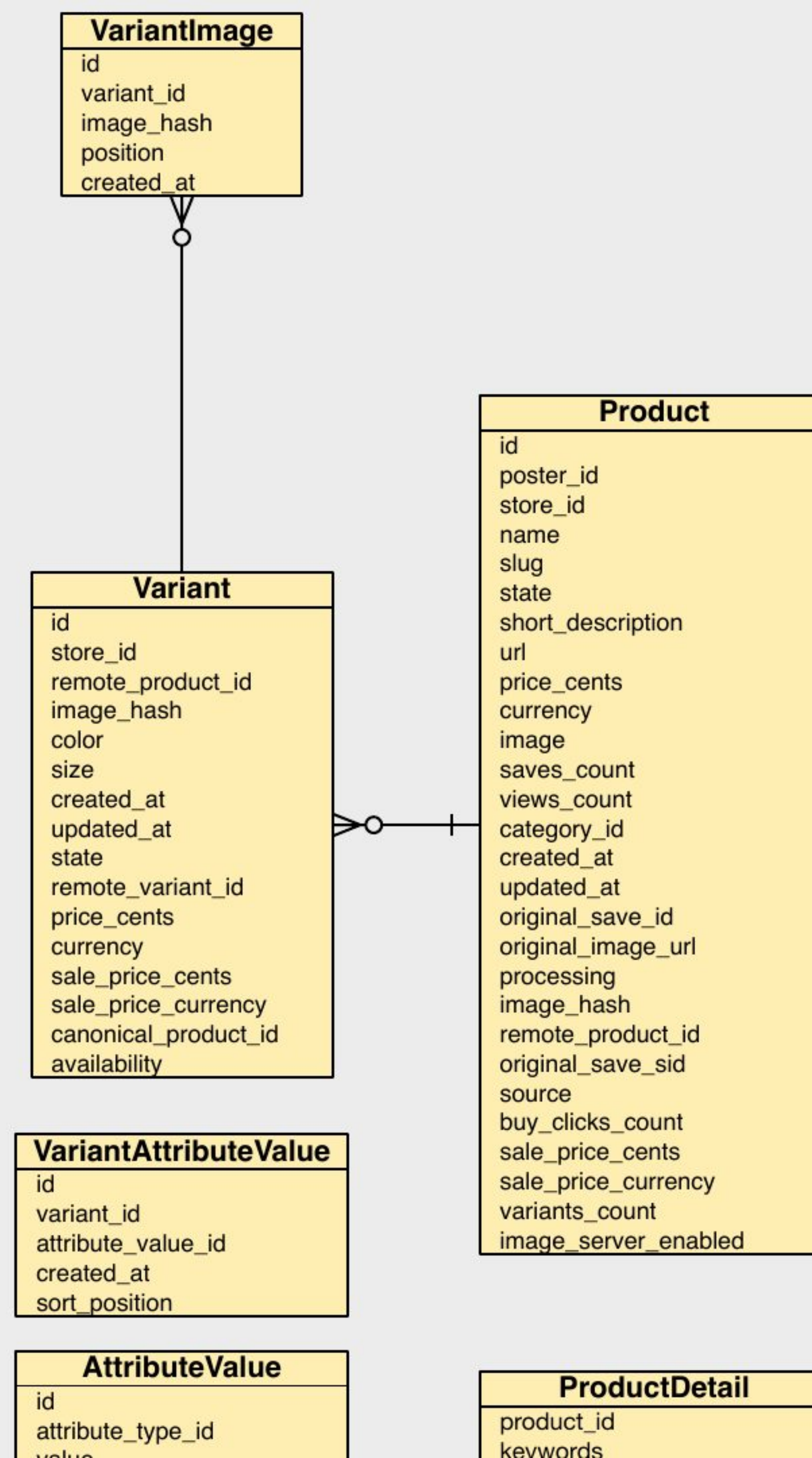
MODULARIZING E-COMMERCE MARKETPLACE



USER DATA & PAYMENT DATA



PRODUCT CATALOG, PRODUCT VARIANTS, PRICING, PRODUCT IMAGES, ETC.



MARKETPLACE STORES, STORE DETAILS, OWNERS, PRODUCT FEED INFOS, ETC

StoreStateTransition
id
from
to
store_id
comment
created_at
updated_at

StoreFeedSignup
id
store_id
feed_url
created_at
updated_at
store_state
username
password
domain
user_id
state

Store
id
domain
state
products_count
followers_count
created_at
updated_at
parent_store_id
name
avatar
price_category
category_id
managers_count
slug
display_name
product_auto_expiration_days
checkout_type
active_products_count
admin_checkout_only
beta_app_checkout_only

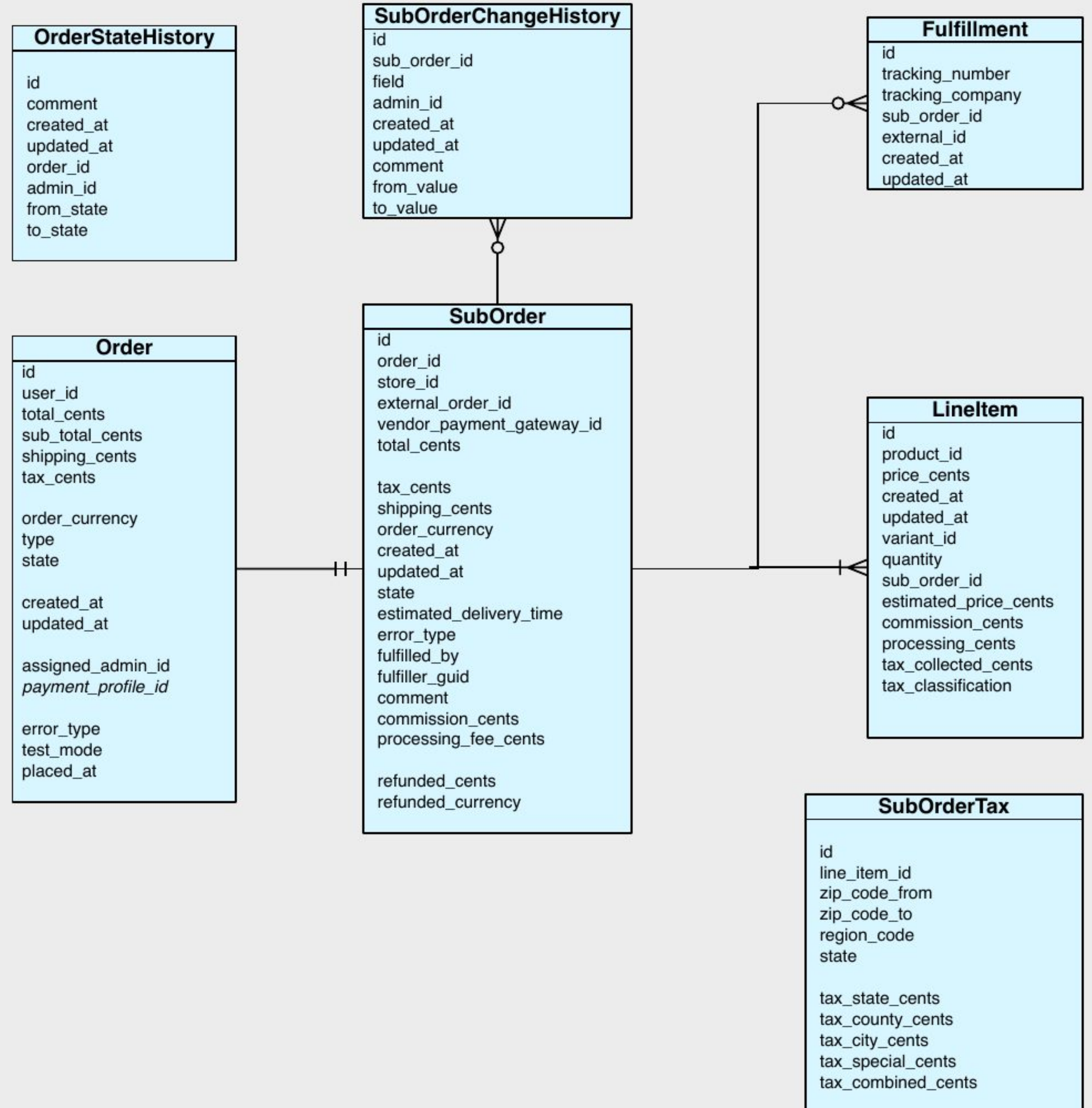
StoreOwners
id
user_id
store_id
state
created_at
updated_at

StoreProfile
store_id
city
state_province
country
created_at
updated_at

StoreDetail
id
store_id
calc_type
calc_value
created_at
updated_at
support_email
tax_rate
taxable_state
style_id
free_shipping_min_cents

StoreFeedContact
id
store_feed_signup_id
full_name
phone
email
contact_type
created_at
updated_at

ORDERS, LINE-ITEMS, CHECKOUT, SHIPMENTS, TRACKING, ORDER STATUS, TAXES, ETC.



MARKETPLACE
SELLER ACCOUNTS,
PAYMENT
TRANSFERS
(COMMISSIONS),
ETC.

VendorAccount
id
store_id
external_id
external_bank_account_id
first_name
last_name
date_of_birth
external_bank_account_last4
secret_key
publishable_key
created_at
updated_at
deleted_at

SubOrderTransfer
id
transfer_id
sub_order_id

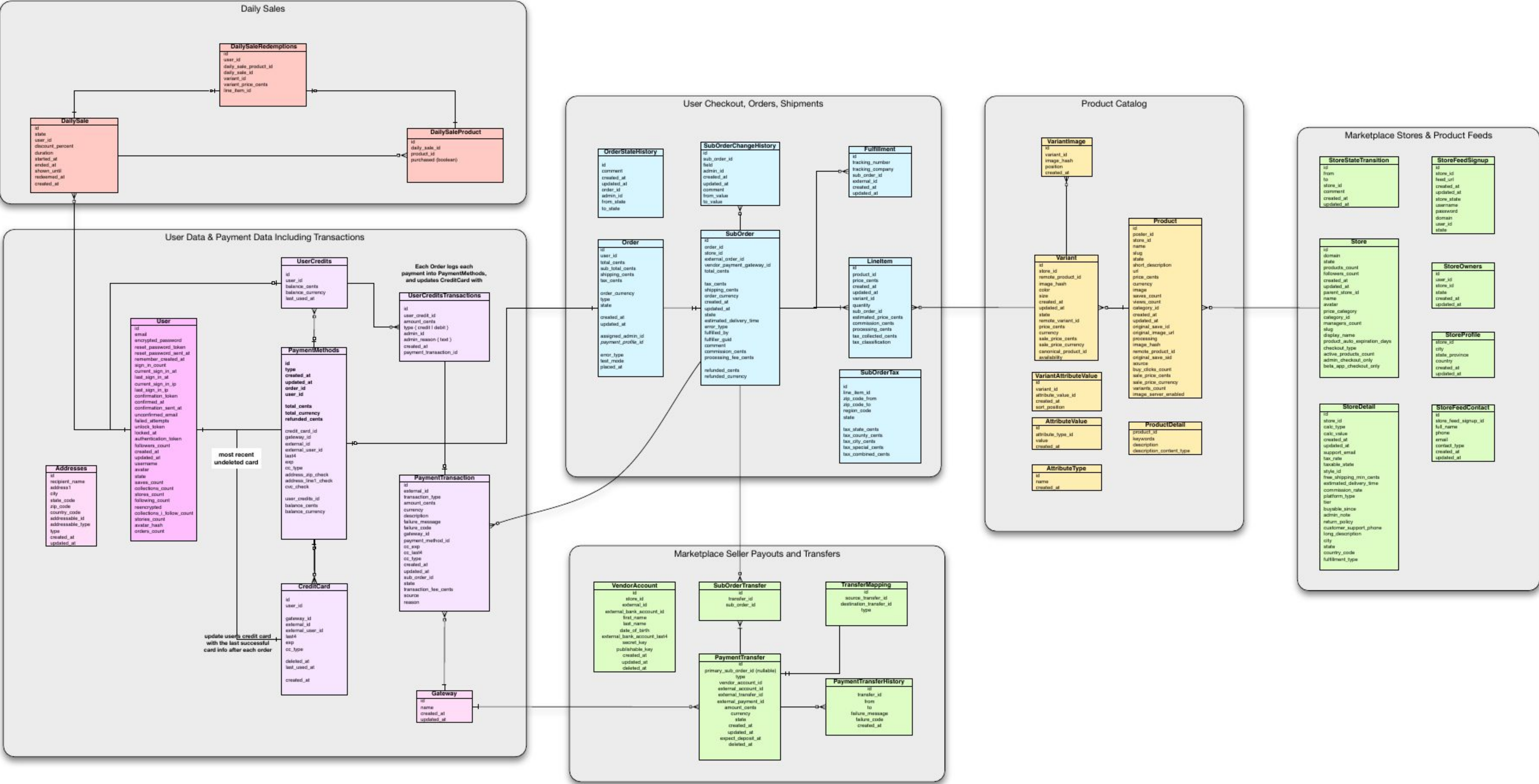
TransferMapping
id
source_transfer_id
destination_transfer_id
type

PaymentTransfer
id
primary_sub_order_id (nullable)
type
vendor_account_id
external_account_id
external_transfer_id
external_payment_id
amount_cents
currency
state
created_at
updated_at
expect_deposit_at
deleted_at

PaymentTransferHistory
id
transfer_id
from
to
failure_message
failure_code
created_at



VOLA! THE DB SCHEMA FOR A BUYER/SELLER MARKETPLACE!



SO WHAT DID WE LEARN?

- ▶ In most domains it's not too difficult to learn how application data model coalesces into modules based on cohesion
- ▶ Cohesion means high level of connectedness. The Order model will likely need to be in the same module as the ShoppingCart, LineItem, Fulfillment (shipment), and so on.
- ▶ But the Checkout module does not need to know necessarily what it's checking out. A polymorphic table of "Sellables" can define which models can be checked out and how.
- ▶ **Product is a model from another module, but it appears in the Checkout module as a Sellable DataObject, not as a Product.**
- ▶ **This is a crucial and necessary separation that has to be enforced, or modularization is pointless if Checkout module has direct access to the Product model from another module.**

WHAT OPTIONS DO WE HAVE FOR LOOSE COUPLING IN RAILS APPS?

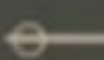
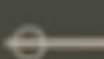
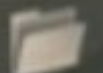
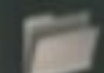
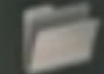
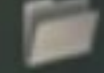
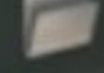
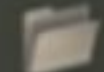
- ▶ **Packwerk** — organize Rails components (app, views, models) into the typical rails folders, but grouped together under the "packs" folder.
- ▶ **Rails Engines** — does not have good boundary enforcement
- ▶ **Ruby Gems** — great for libraries, CLI tools, even some Rails plugins. When appropriate gems are a fantastic way to extract some low-level functionality.
- ▶ The dependency should always be from the app to the gem, and never backwards, unless inversion of control is applied.

WHAT IS PACKWERK? ▶ <https://github.com/Shopify/packwerk>

- ▶ Packwerk is a Ruby gem used to enforce boundaries and modularize Rails applications.
- ▶ Packwerk can be used to:
 - ▶ Combine groups of files into packages
 - ▶ Define package-level constant visibility (i.e. have publicly accessible constants)
 - ▶ Help existing codebases to become more modular without obstructing development

INTRODUCING PACKWERK

Introducing Packwerk: Quick Summary

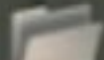
-  app
 -   models
 -   views
 -   controllers
 -  packs
 -  sales
 -  app
 -  public 
 -  models 
 -  views 
 -  controllers 
 -  reviews
 -  social
- Static code analysis
 - Detect private code access
 - Integrate into CI
 - Write a file to track todos

► Source: <https://bit.ly/packwerk>

Talk on Packwerk by Marc Reynolds at Rocky Mountain Ruby

BREAKING UP INTO MODULES

Introducing Packwerk

-  packs
 -  sales
 -  app
 -  public
 -  models
 -  sale.rb
 -  reviews
 -  app
 -  models
 -  review.rb

```
# packs/sales/app/models/sale.rb
class Sale < ActiveRecord::Base
  belongs_to :customer
  belongs_to :item
  scope :finalized, -> { where(status: "finalized") }
end

# packs/reviews/app/models/review.rb
class Review < ActiveRecord::Base
  belongs_to :item

  def sales_rolling_30
    Sale # THIS IS A VIOLATION!!!
      .finalized
      .where(item: item)
      .where("created_at >= ?", 30.days.ago)
  end
end
```

- Source: <https://bit.ly/packwerk>
Talk on Packwerk by Marc Reynolds at Rocky Mountain Ruby

VERIFYING BOUNDARIES

Introducing Packwerk

.....
📦 Finished in 72.92 seconds

packs/reviews/spec/models/review.rb:6:4

Privacy violation: '::Sale' is private to 'packs/sales' but referenced from
'packs/reviews'.

Is there a public entrypoint in 'packs/sales/app/public/' that you can use instead?

Inference details: this is a reference to ::Sale which seems to be defined in
packs/sales/app/models/sale.rb.

To receive help interpreting or resolving this error message, see:

<https://github.com/Shopify/packwerk/blob/main/TROUBLESHOOT.md#Troubleshooting-violations>

1 offense detected

No stale violations detected

► Source: <https://bit.ly/packwerk>

Talk on Packwerk by Marc Reynolds at Rocky Mountain Ruby

HOW TO INTEGRATE PACKWERK

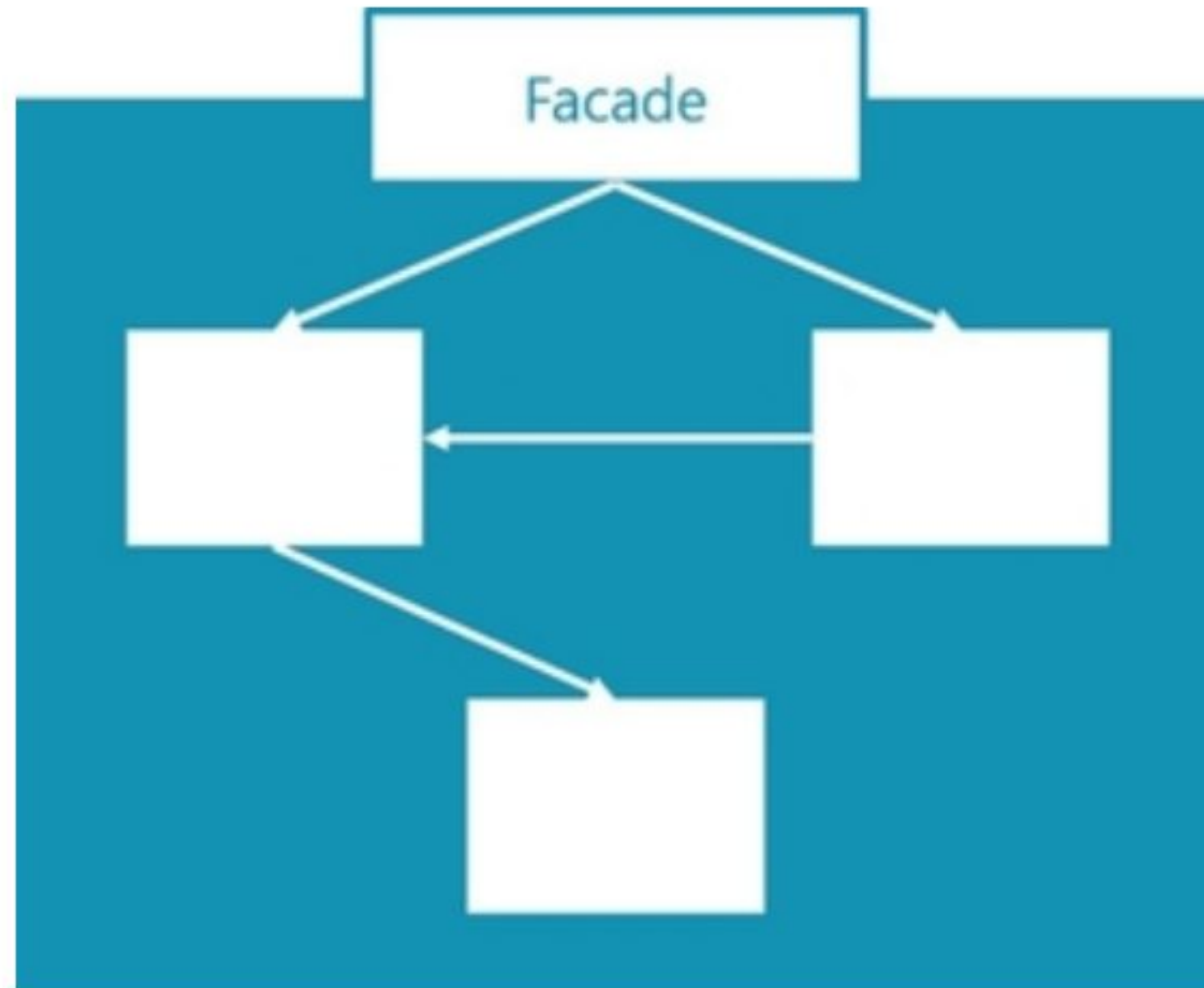
- ▶ Figure out what you have first (analyze your data model)
- ▶ Move code into packages
- ▶ Declare package dependencies
- ▶ Open public interfaces, create DataObjects
- ▶ Document public interfaces and assign code owners
- ▶ Maintain and Grow

▶ Source: <https://bit.ly/packwerk>

Talk on Packwerk by **Marc Reynolds** at Rocky Mountain Ruby

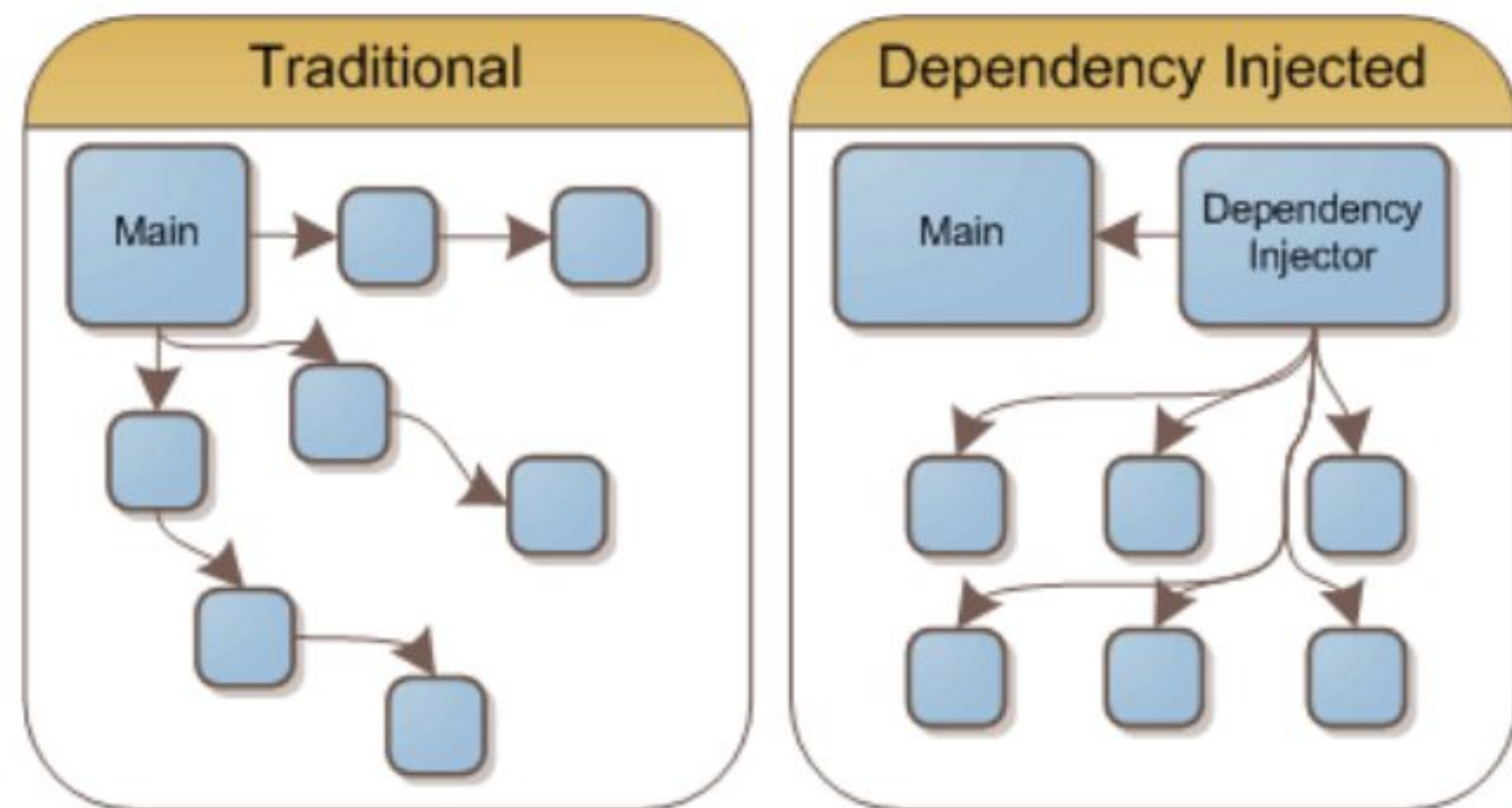
WHAT ABOUT FACADE AND INVERSION OF CONTROL?

FACADE DESIGN PATTERN



- ▶ Is a public interface for your module
- ▶ Shields access to module internals
- ▶ Does not return references to ActiveRecord objects.
- ▶ Instead returns DataObjects constructed from AR models and other data.

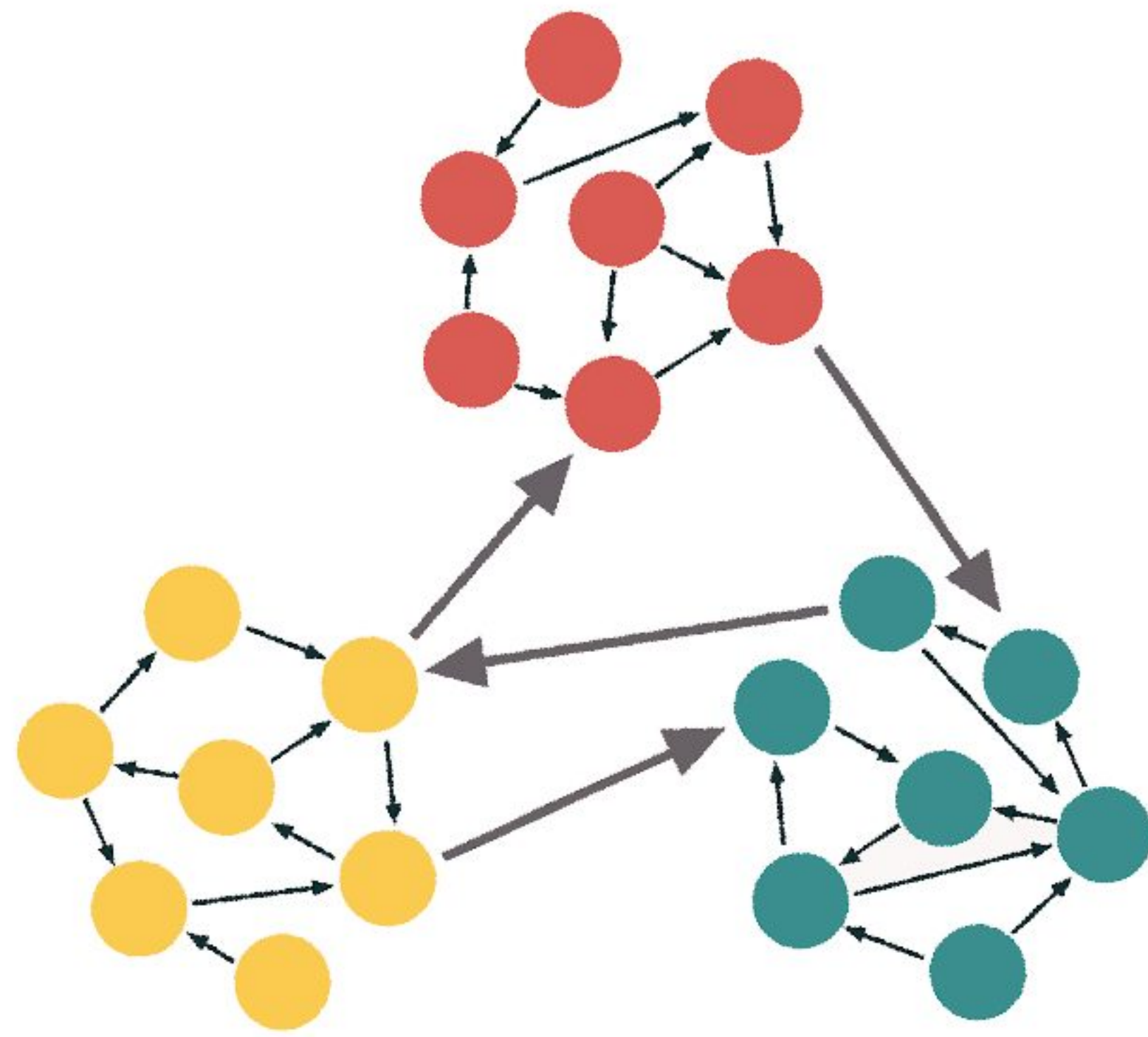
INVERSION OF CONTROL OR DEPENDENCY INJECTION



- ▶ Instead of directly instantiating classes that do not belong to the current module, we can ask an appropriate module to return us the data we need.
 - ▶ That's called inversion of control (IoC)
- ▶ Eg. the Checkout module must be able to price the product, show it's image, and description on the checkout.
- ▶ The `Checkout.checkout()` method should then be "injected" with the data objects that contain all of these details without providing direct access to the Product ActiveRecord model.

SO WHAT ARE THE TAKEAWAYS?

- ▶ Loosely coupled, modular design **does not appear out of the blue or accidentally.**
- ▶ It's an answer to the problem of ever growing **spaghetti codebase** that nobody seems to understand after a certain point.
- ▶ Modular design is a **necessity for large teams (100+)** working on a monolithic codebase. Some companies will stall at much smaller numbers.
- ▶ Loose coupling design encourages engineers to think in terms of components in their module and their **public APIs**, and use existing or **negotiate new APIs with the owners of other modules.**
 - ▶ This means a good chance of doing it right without adding any new tech debt.
- ▶ **Packwerk** offers perhaps the most flexible, incremental and well-travelled path to modularizing any Ruby codebase.
 - ▶ It is the only such tool currently used in production by Shopify and many others.



IN THE END WE CAN ALL HOPEFULLY....

**ENJOY LOOSELY COUPLED CODEBASES AND
SPEED UP DEV VELOCITY, AND REDUCE TECH DEBT.**



THAT'S ALL WE HAVE TIME FOR TODAY ...



KONSTANTIN GREDESKOUL

Thank you!

ANY QUESTIONS?

- [ACADEMIA.EDU](https://academia.edu)
- [KIG@KIG.RE](mailto:kig@kig.re)
- @KIGSTER ON GITHUB
- @KIG ON X/TWITTER

Please watch the Packwerk talk by Marc Reynolds
Rocky Mountain Ruby

<https://bit.ly/packwerk>