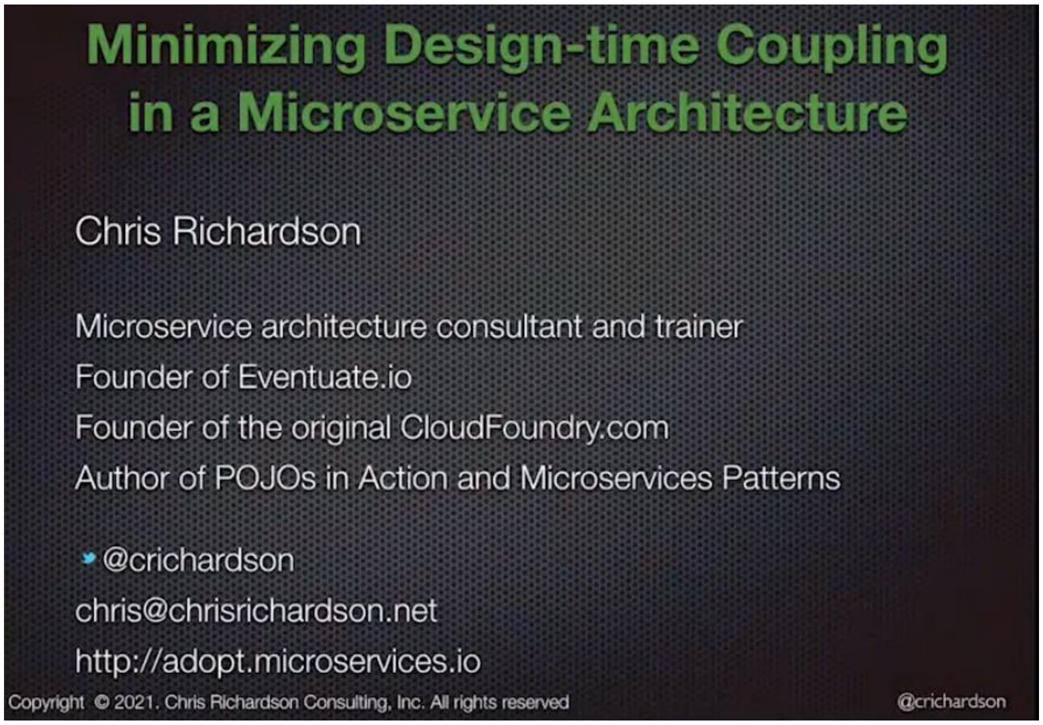
TRACK
Microservice - The First Decade

SESSION

Minimizing Design Time Coupling a Microservice Architecture

Chris Richardson
Creator of microservices.io; Author of Microservices patterns & Java Champion

Chris Richardson discusses design-time coupling in a microservice architecture and why it's essential to minimize it, describing how to design service APIs to reduce coupling.

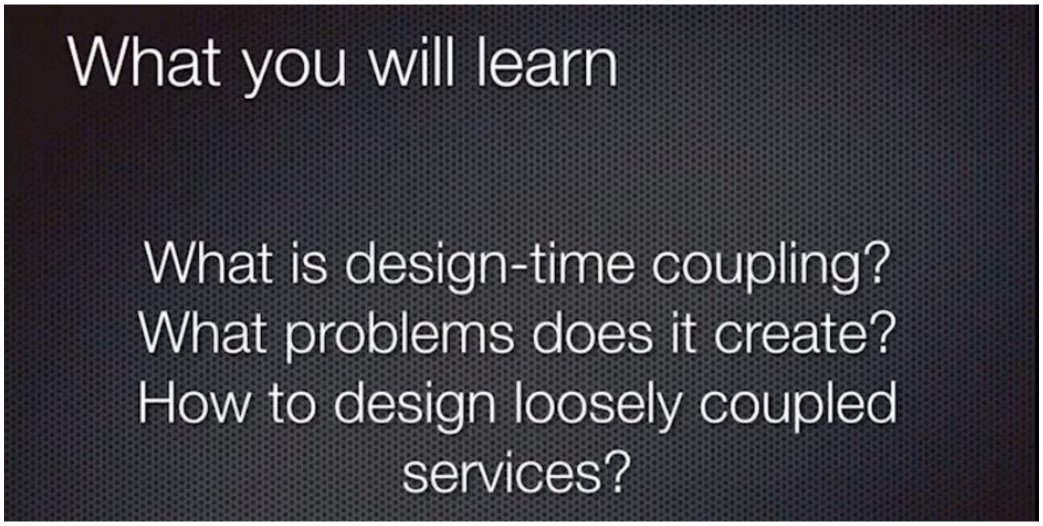


Minimizing Design-time Coupling in a Microservice Architecture

Chris Richardson

Microservice architecture consultant and trainer
Founder of Eventuate.io
Founder of the original CloudFoundry.com
Author of POJOs in Action and Microservices Patterns

@crichardson
chris@chrisrichardson.net
http://adopt.microservices.io



What you will learn

What is design-time coupling?
What problems does it create?
How to design loosely coupled services?

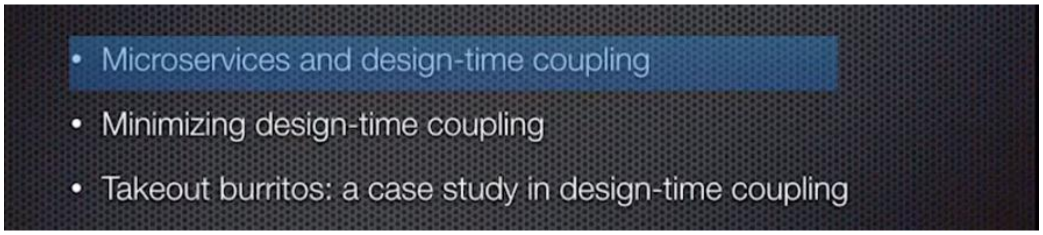
About Chris

http://adopt.microservices.io

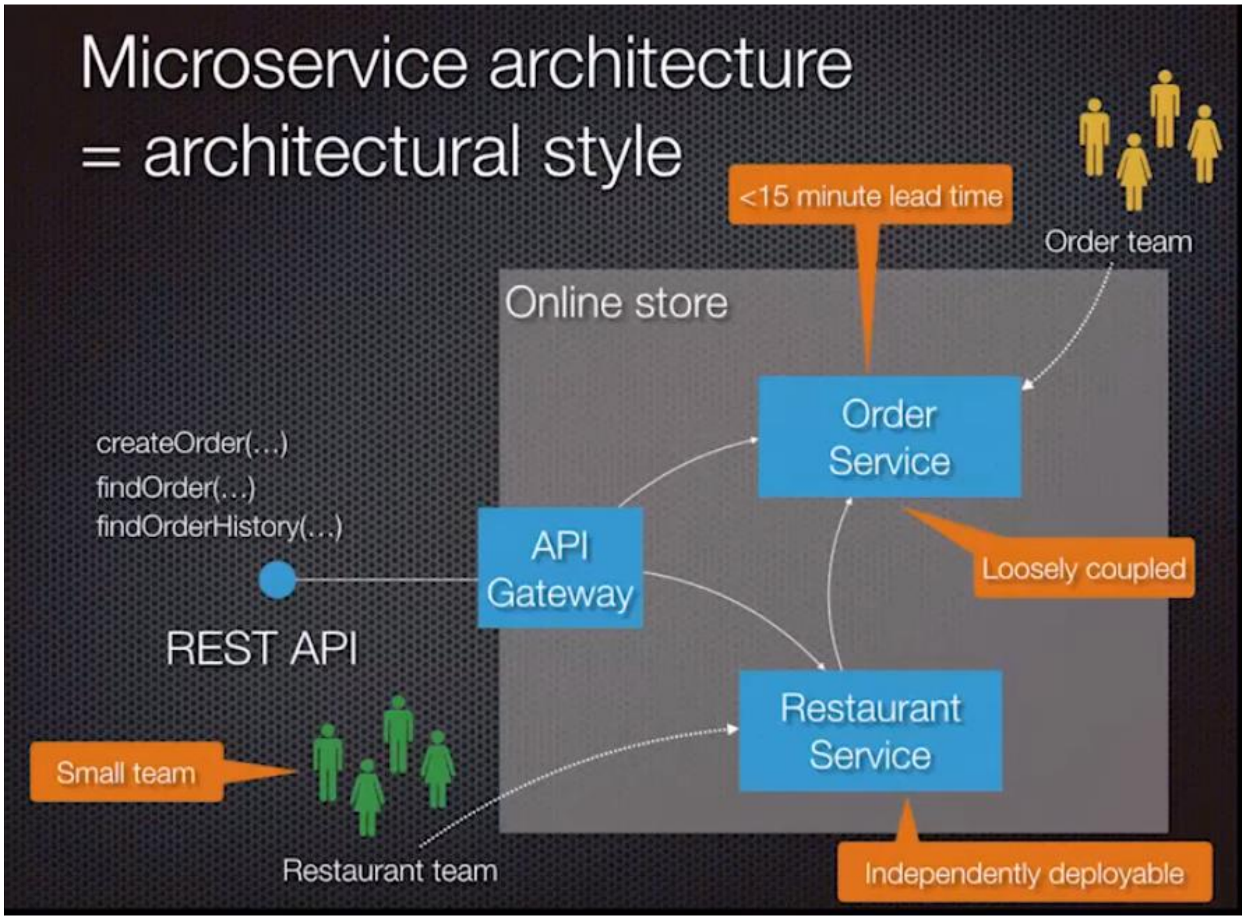
35% discount
ctwqcon21

$120 discount coupon
HIDKQIFC

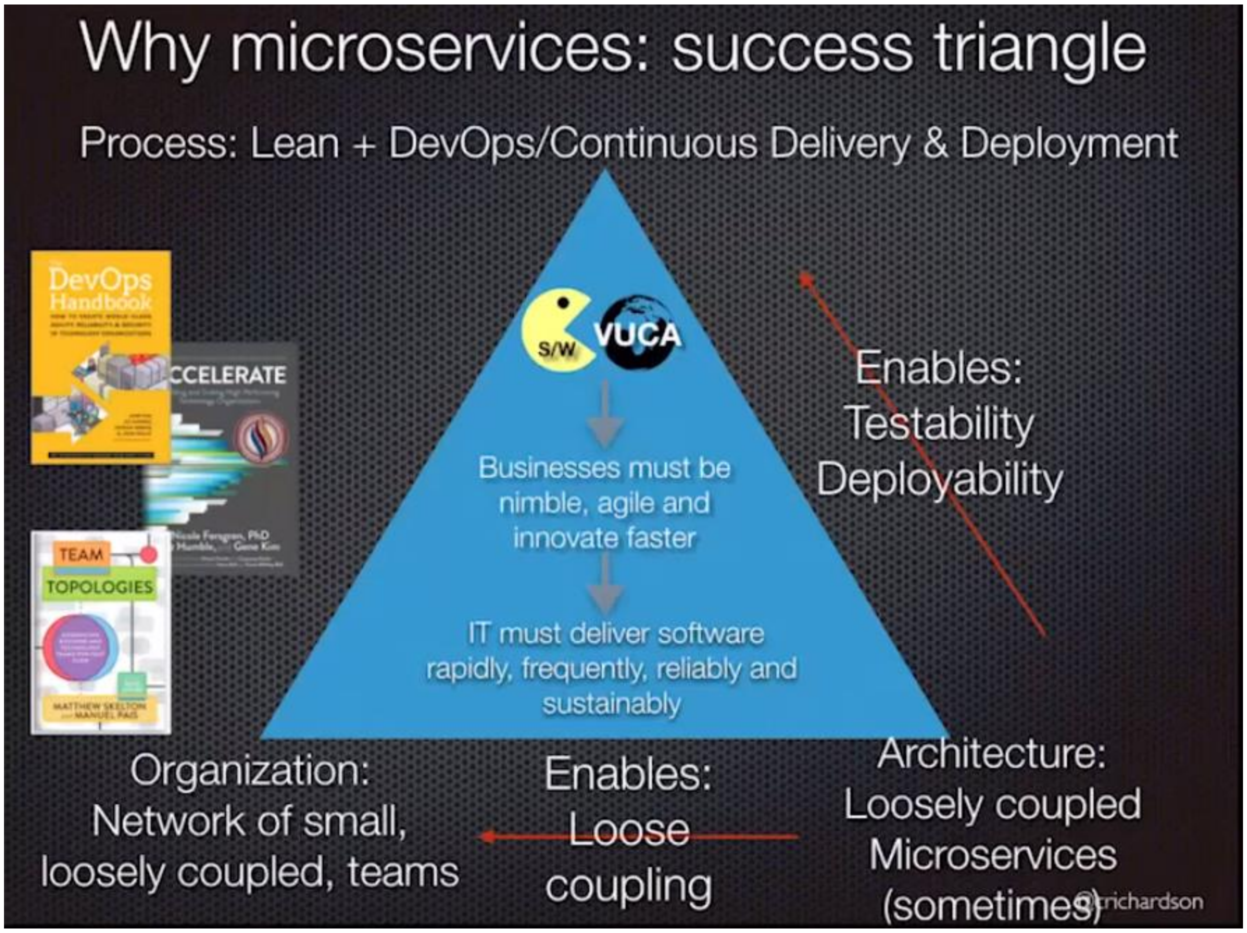
http://adopt.microservices.io



- Microservices and design-time coupling
- Minimizing design-time coupling
- Takeout burritos: a case study in design-time coupling



Microservice architecture = architectural style

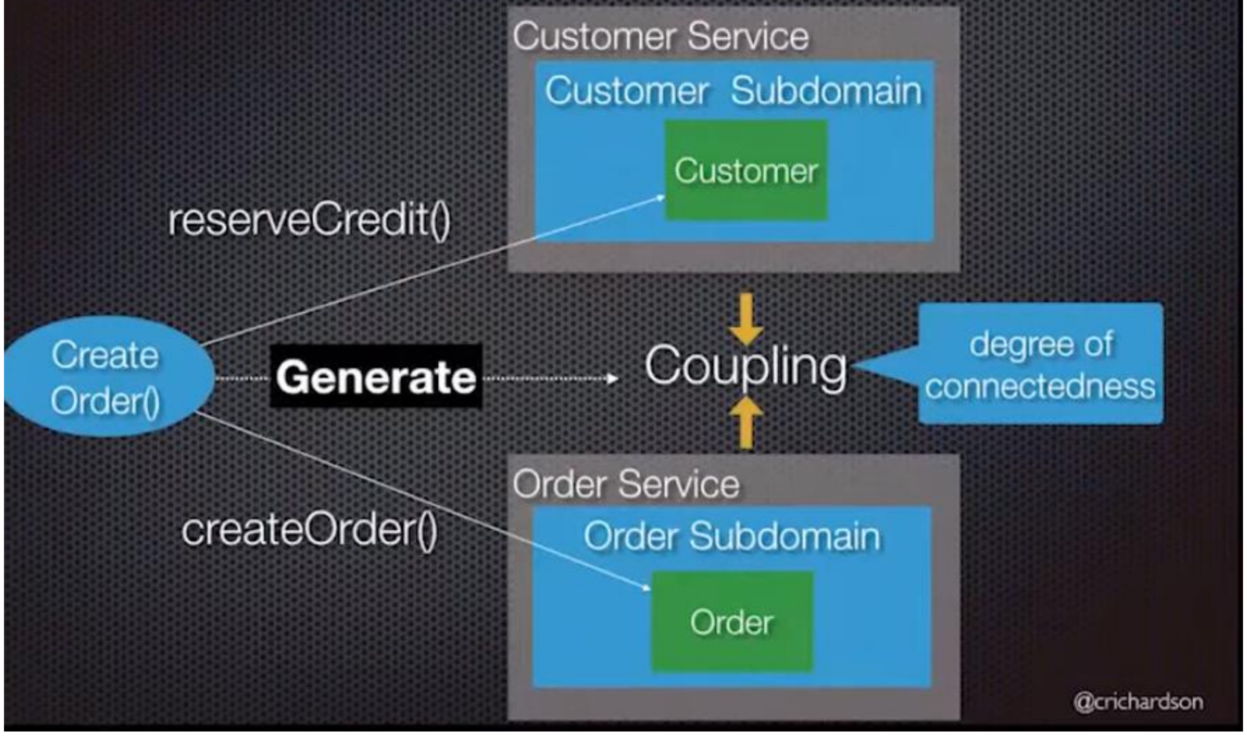


Why microservices: success triangle

Process: Lean + DevOps/Continuous Delivery & Deployment

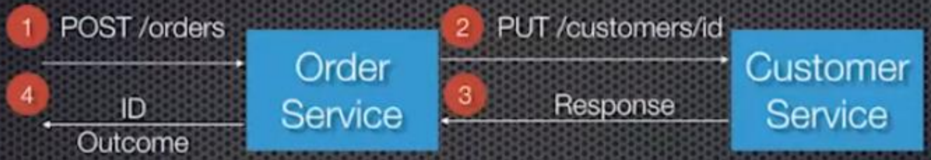
S/W
VUCA
Businesses must be nimble, agile and innovate faster
IT must deliver software rapidly, frequently, reliably and sustainably
Enables: Testability Deployability
Organization: Network of small, loosely coupled, teams
Enables: Loose coupling
Architecture: Loosely coupled Microservices (sometimes)
@crichardson

Operations that span services generate coupling

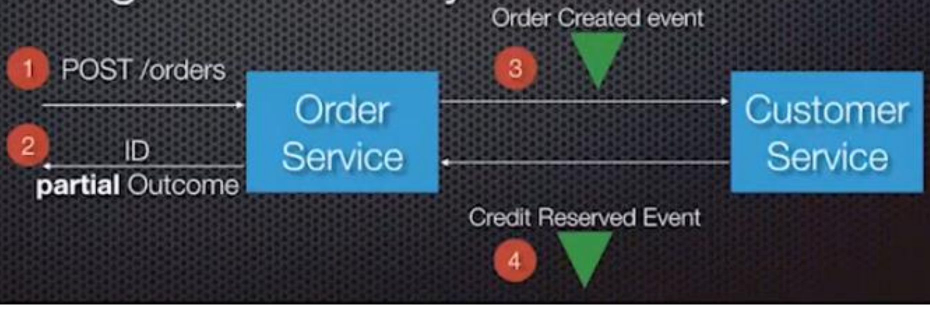


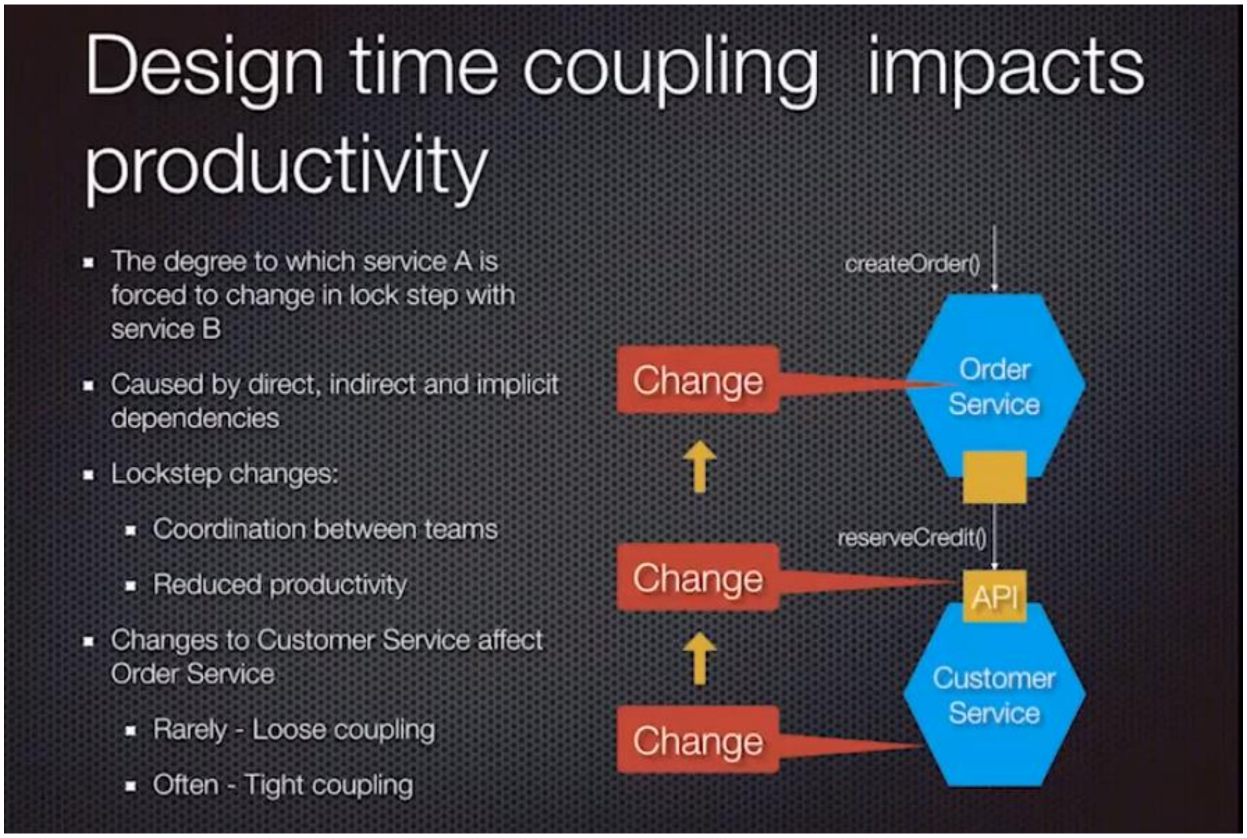
Runtime coupling impacts availability

Tight: lower availability



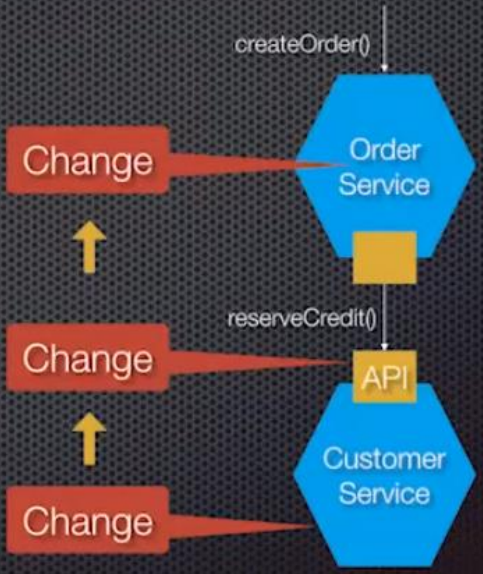
Loose: higher availability

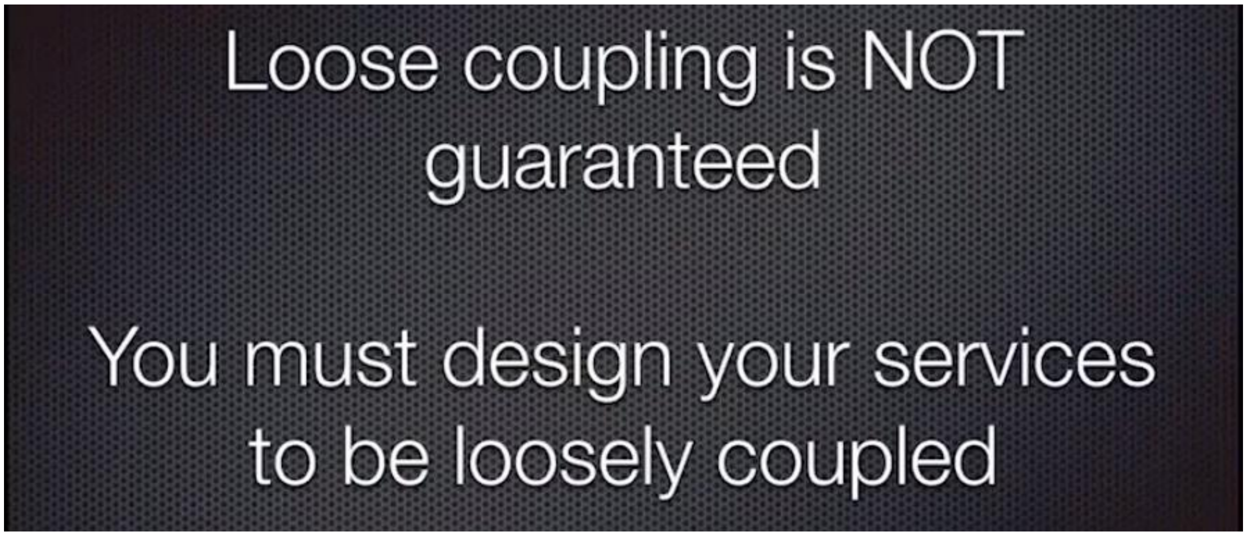




Design time coupling impacts productivity

- The degree to which service A is forced to change in lock step with service B
- Caused by direct, indirect and implicit dependencies
- Lockstep changes:
  - Coordination between teams
  - Reduced productivity
- Changes to Customer Service affect Order Service
  - Rarely - Loose coupling
  - Often - Tight coupling

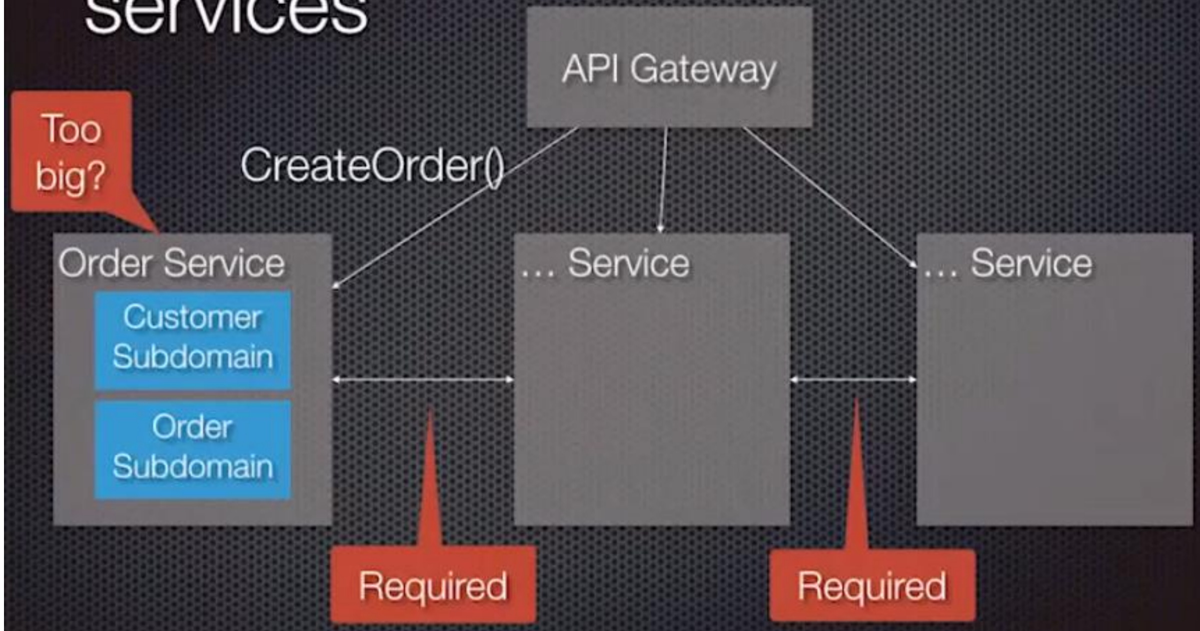




Loose coupling is NOT guaranteed

You must design your services to be loosely coupled

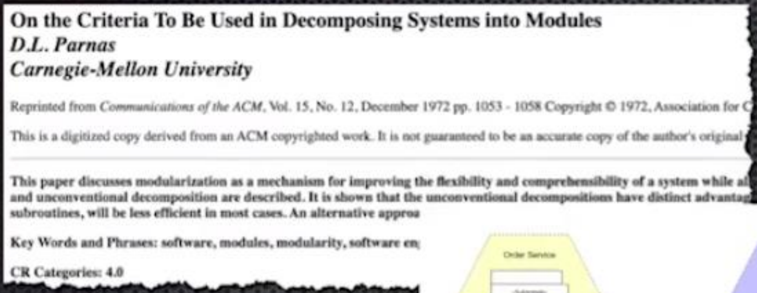
Ideally: no coupling between services



In practice: collaboration is unavoidable ⇒ need to minimize coupling

- Minimizing design-time coupling

Modularity and loose coupling is an old idea



On the Criteria To Be Used in Decomposing Systems into Modules
D.L. Parnas
Carnegie-Mellon University

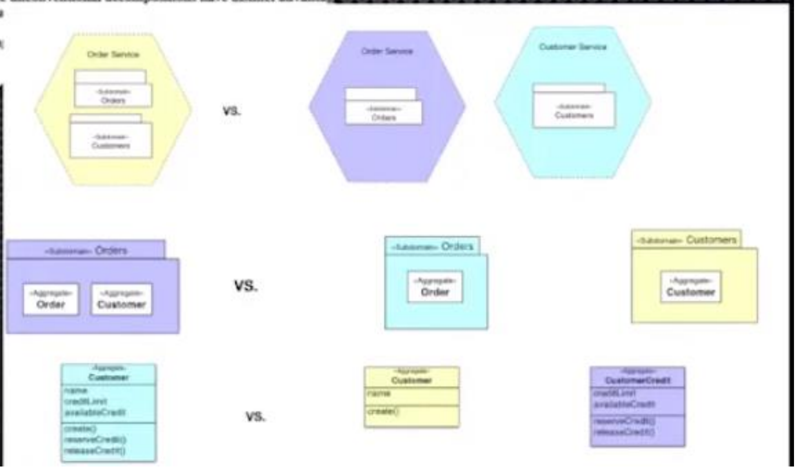
Reprinted from *Communications of the ACM*, Vol. 15, No. 12, December 1972 pp. 1053 - 1058 Copyright © 1972, Association for C

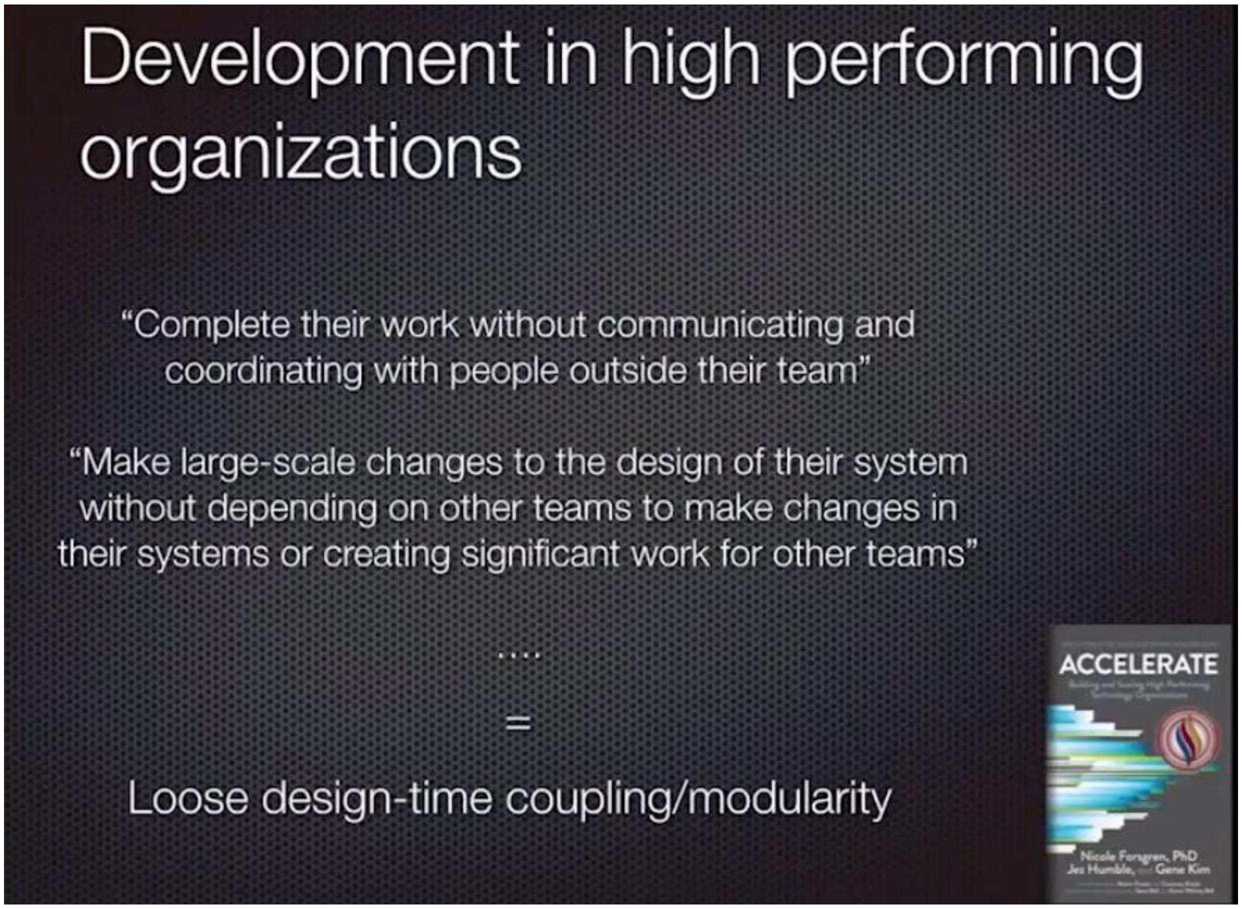
This is a digitized copy derived from an ACM copyrighted work. It is not guaranteed to be an accurate copy of the author's original

This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while al and unconventional decomposition are described. It is shown that the unconventional decompositions have distinct advantag subroutines, will be less efficient in most cases. An alternative approa

Key Words and Phrases: software, modules, modularity, software en

CR Categories: 4.0





Development in high performing organizations

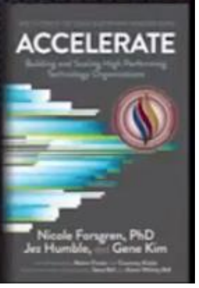
"Complete their work without communicating and coordinating with people outside their team"

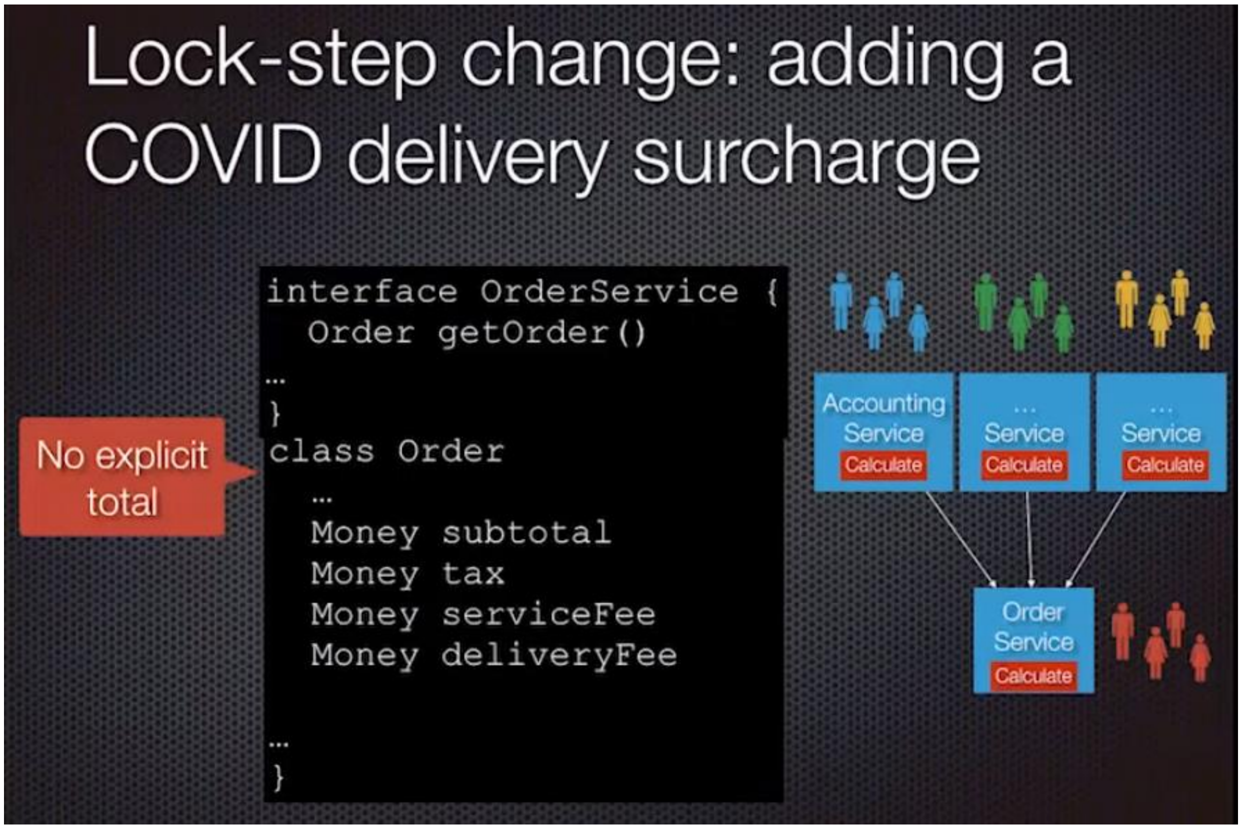
"Make large-scale changes to the design of their system without depending on other teams to make changes in their systems or creating significant work for other teams"

....

=

Loose design-time coupling/modularity

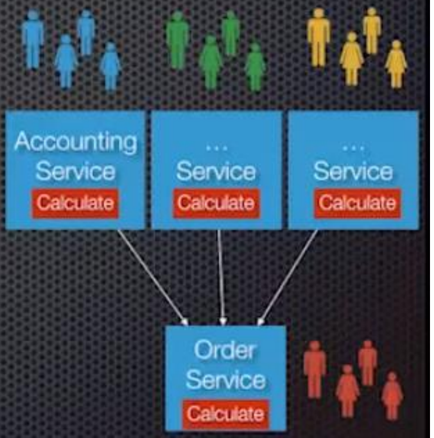


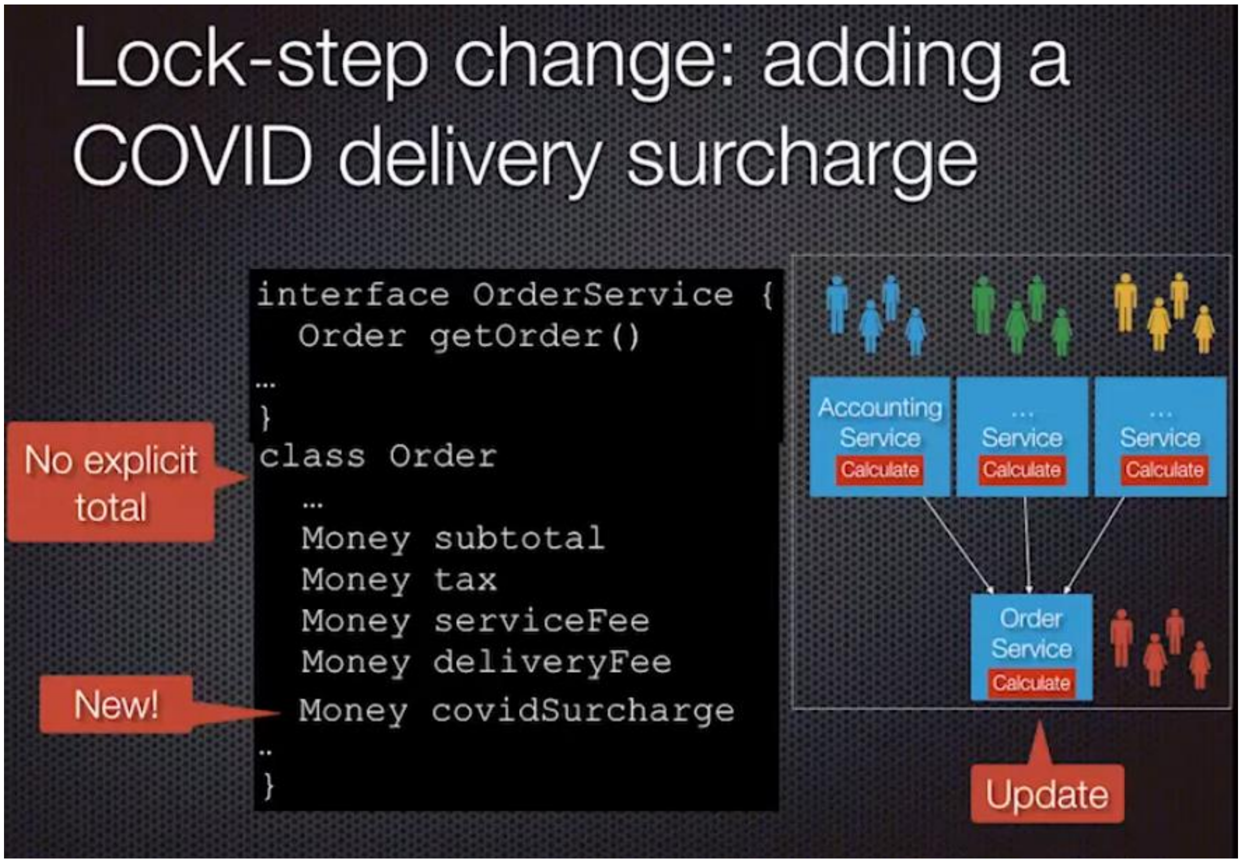


Lock-step change: adding a COVID delivery surcharge

No explicit total

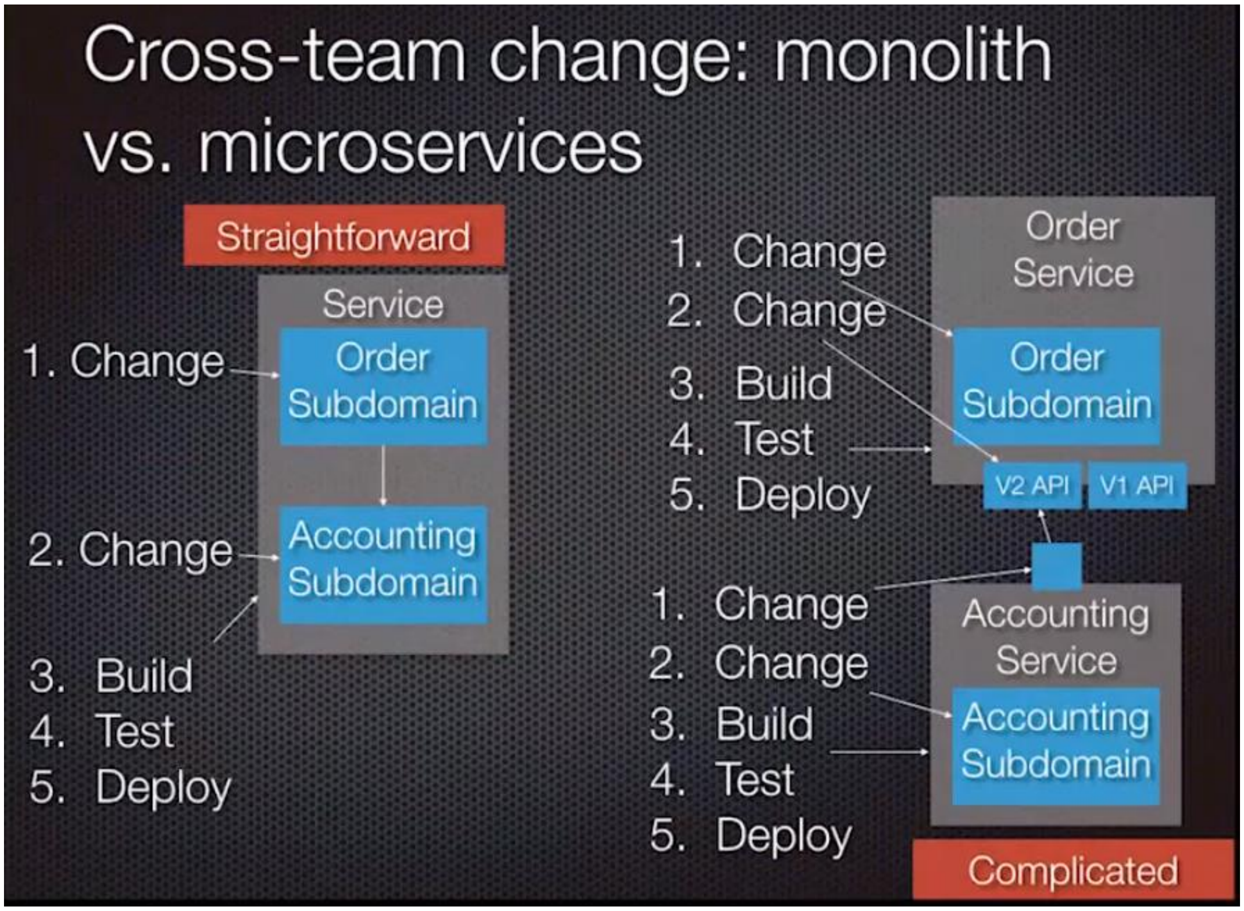
```
interface OrderService {
  Order getOrder()
…
}
class Order
  …
  Money subtotal
  Money tax
  Money serviceFee
  Money deliveryFee

…
}
```



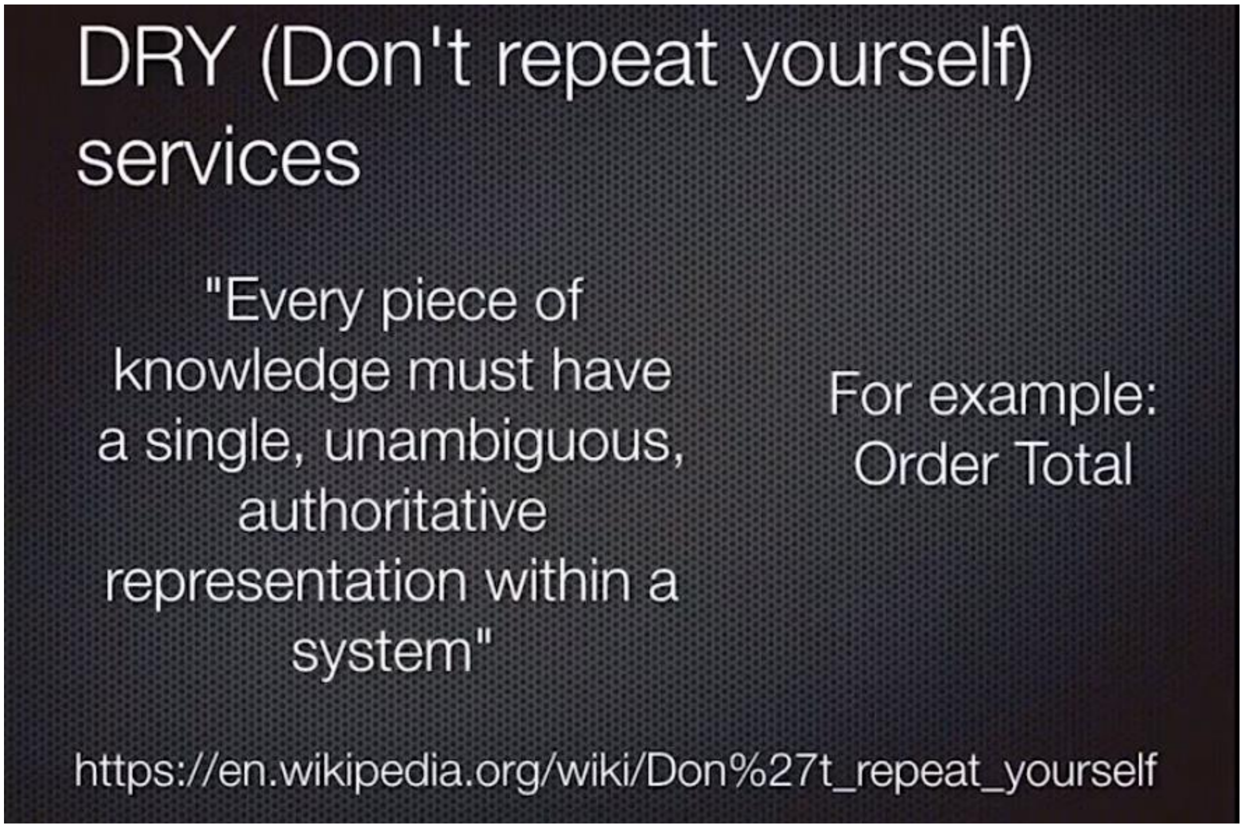


Lock-step change: adding a COVID delivery surcharge

```
interface OrderService {
  Order getOrder()
…
}
class Order
  …
  Money subtotal
  Money tax
  Money serviceFee
  Money deliveryFee
  Money covidSurcharge
..
}
```



Cross-team change: monolith vs. microservices



DRY (Don't repeat yourself) services

"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system"

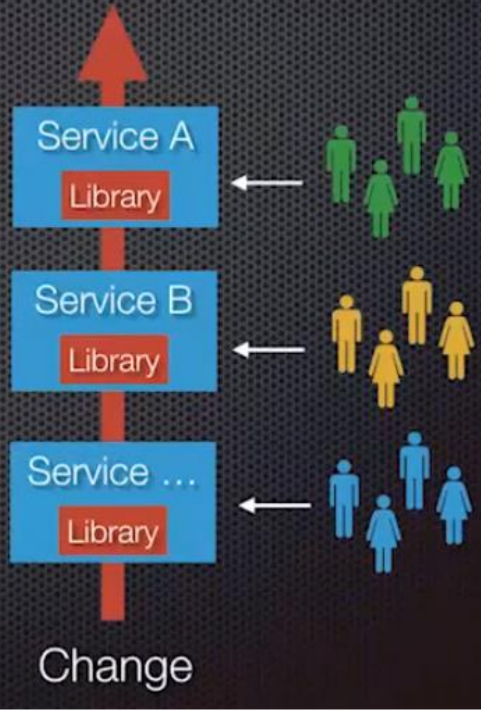
For example: Order Total

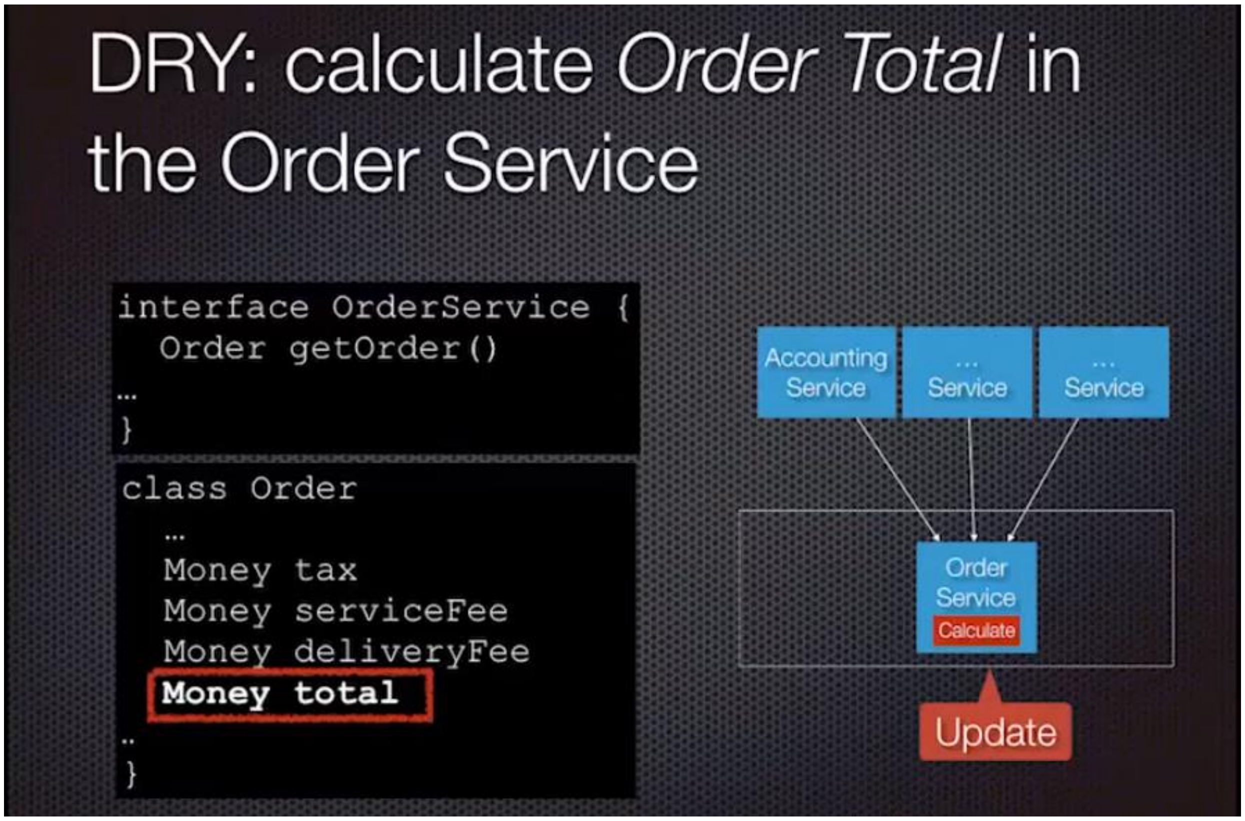
https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

Shared library that calculates Order Total != DRY

Shared libraries containing business logic that changes ⇒ requires multiple services to change/rebuild/ redeployed in lock step ❌

Shared utility libraries ✅

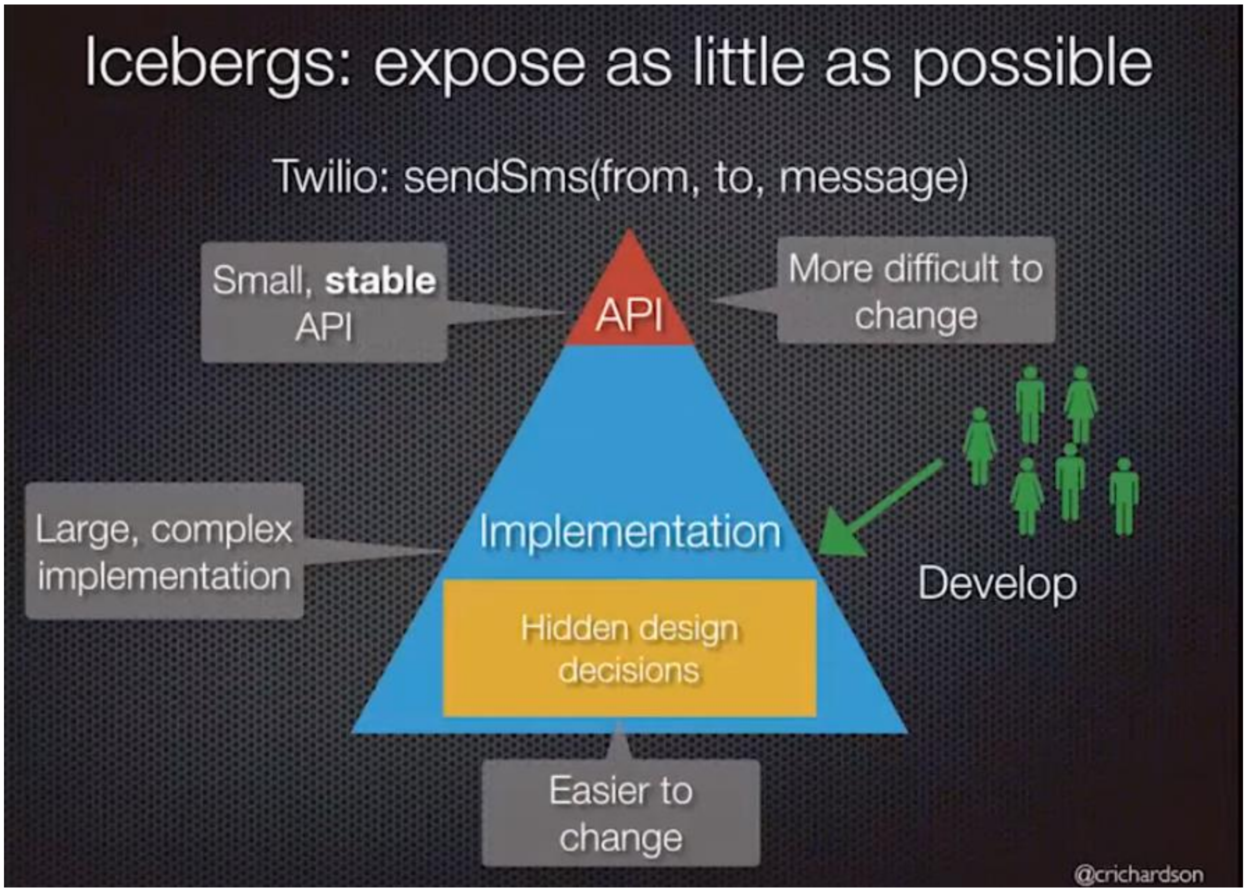




DRY: calculate *Order Total* in the Order Service

```
interface OrderService {
  Order getOrder()
…
}
```

```
class Order
  …
  Money tax
  Money serviceFee
  Money deliveryFee
  Money total
..
}
```

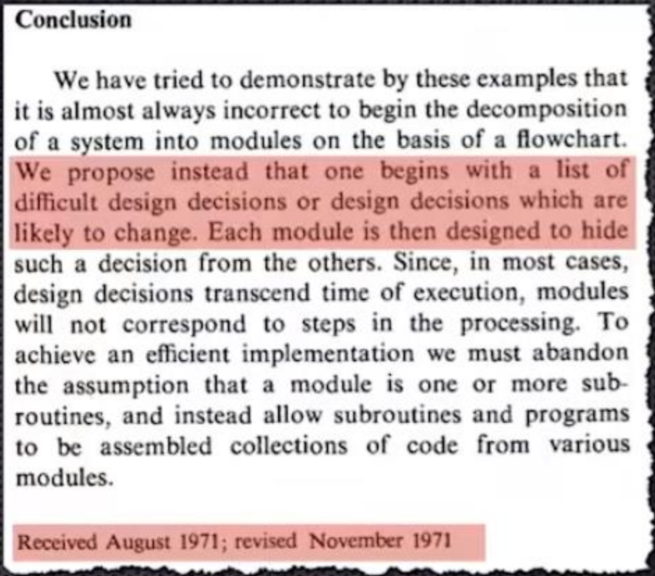


Icebergs: expose as little as possible

Twilio: sendSms(from, to, message)

@crichardson

What to encapsulate?



Conclusion

We have tried to demonstrate by these examples that it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. 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. Since, in most cases, design decisions transcend time of execution, modules will not correspond to steps in the processing. To achieve an efficient implementation we must abandon the assumption that a module is one or more subroutines, and instead allow subroutines and programs to be assembled collections of code from various modules.

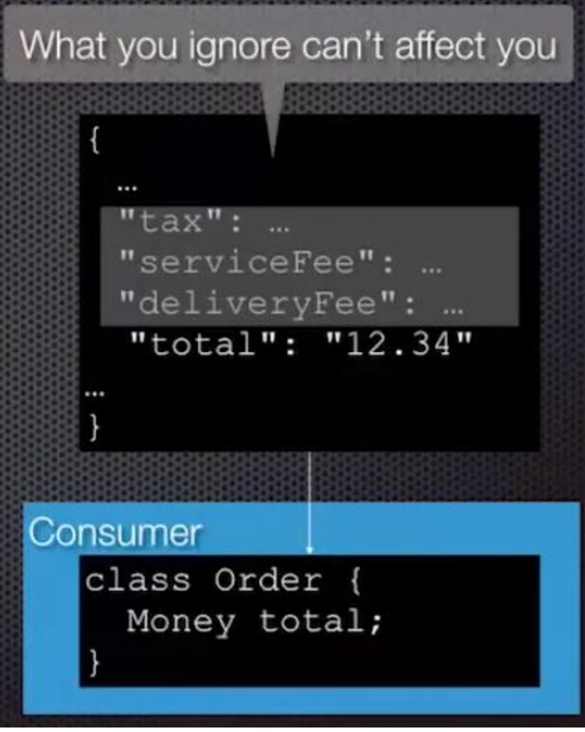
Received August 1971; revised November 1971

Ancient wisdom from Parnas!

@crichardson

Consume as little as possible

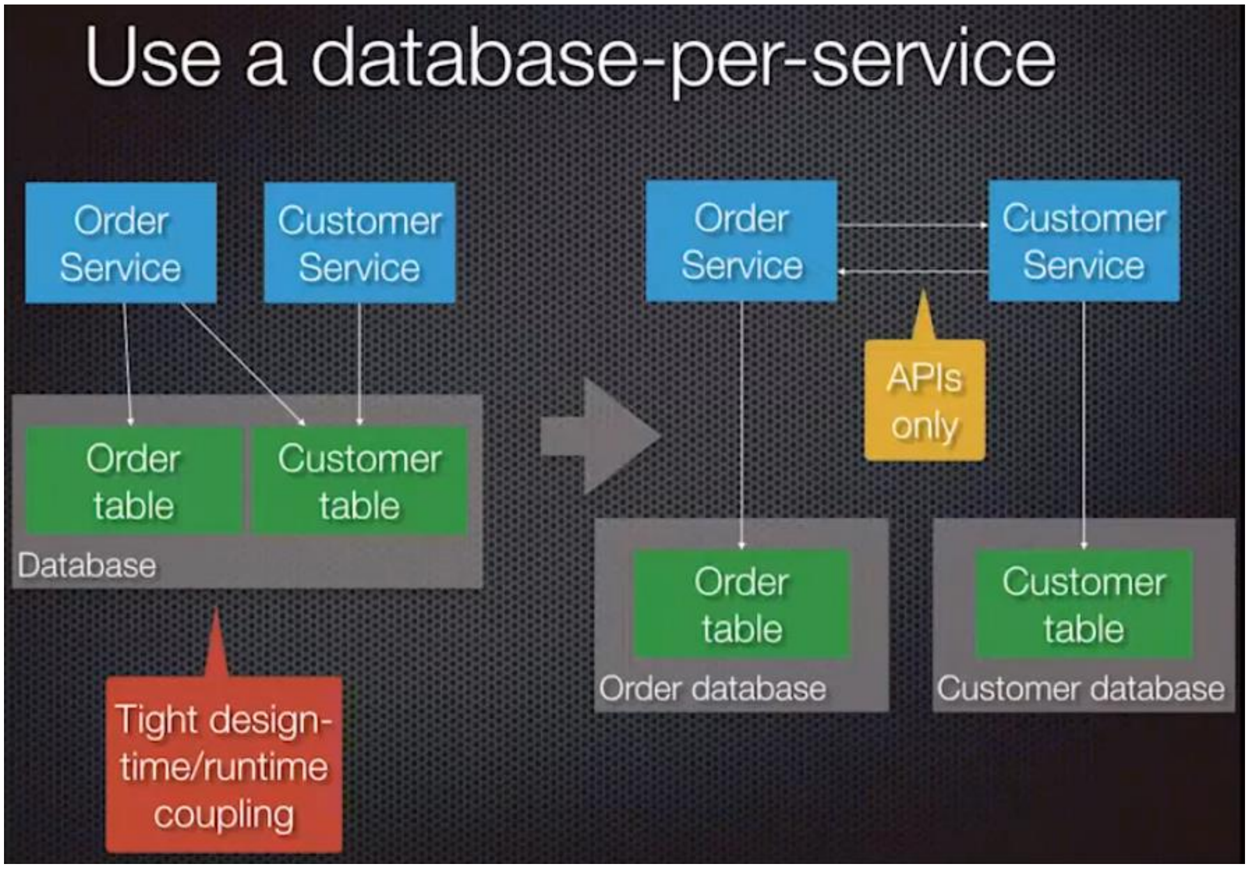
- Minimize
  - number of dependencies
  - what's consumed from each dependency
- Apply Postel's Robustness principle: https://en.wikipedia.org/wiki/Robustness_principle
- Consumer-driven contract tests verify compliance
- BTW: Swagger/Protobuf-generated stubs parse everything!



What you ignore can't affect you

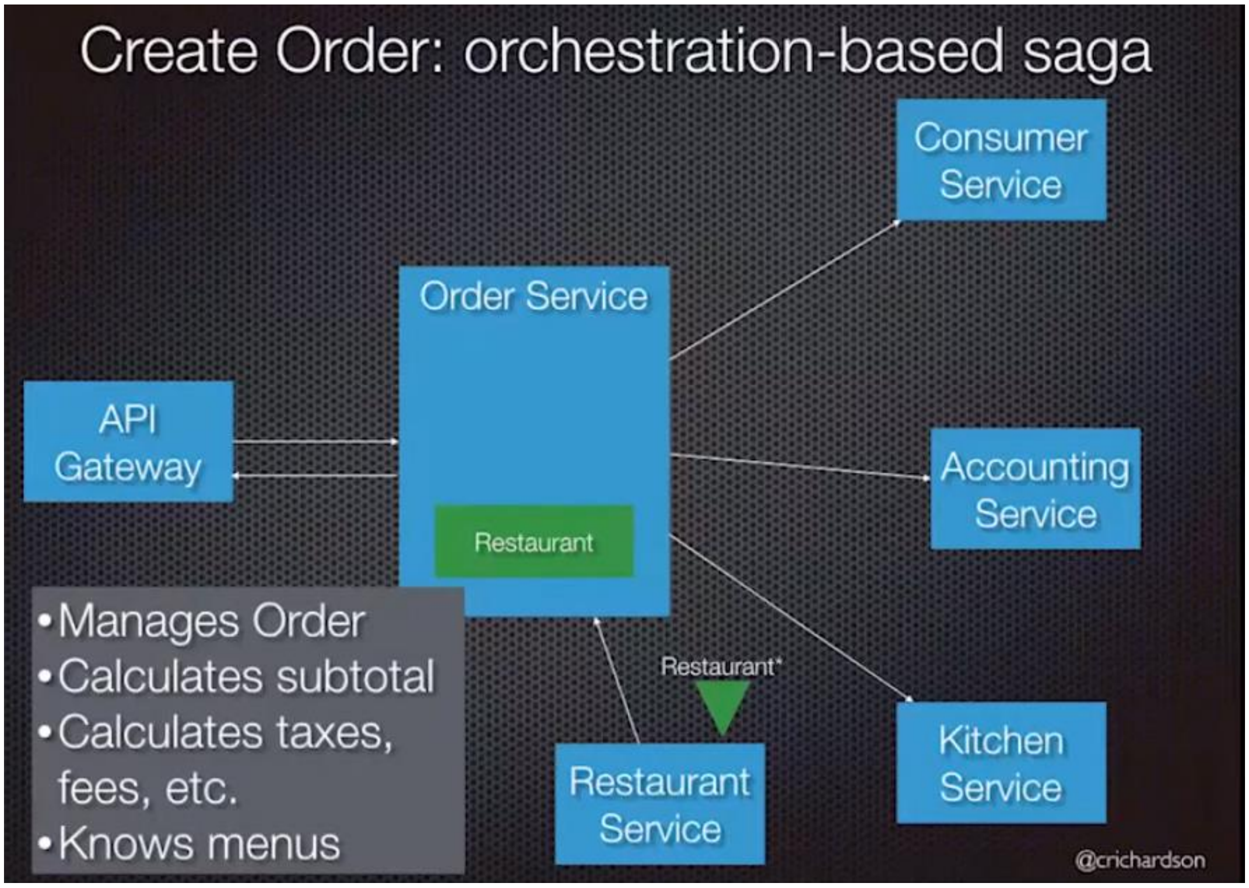
```
{
 ...
 "tax": ...
 "serviceFee": ...
 "deliveryFee": ...
  "total": "12.34"
...
}
```

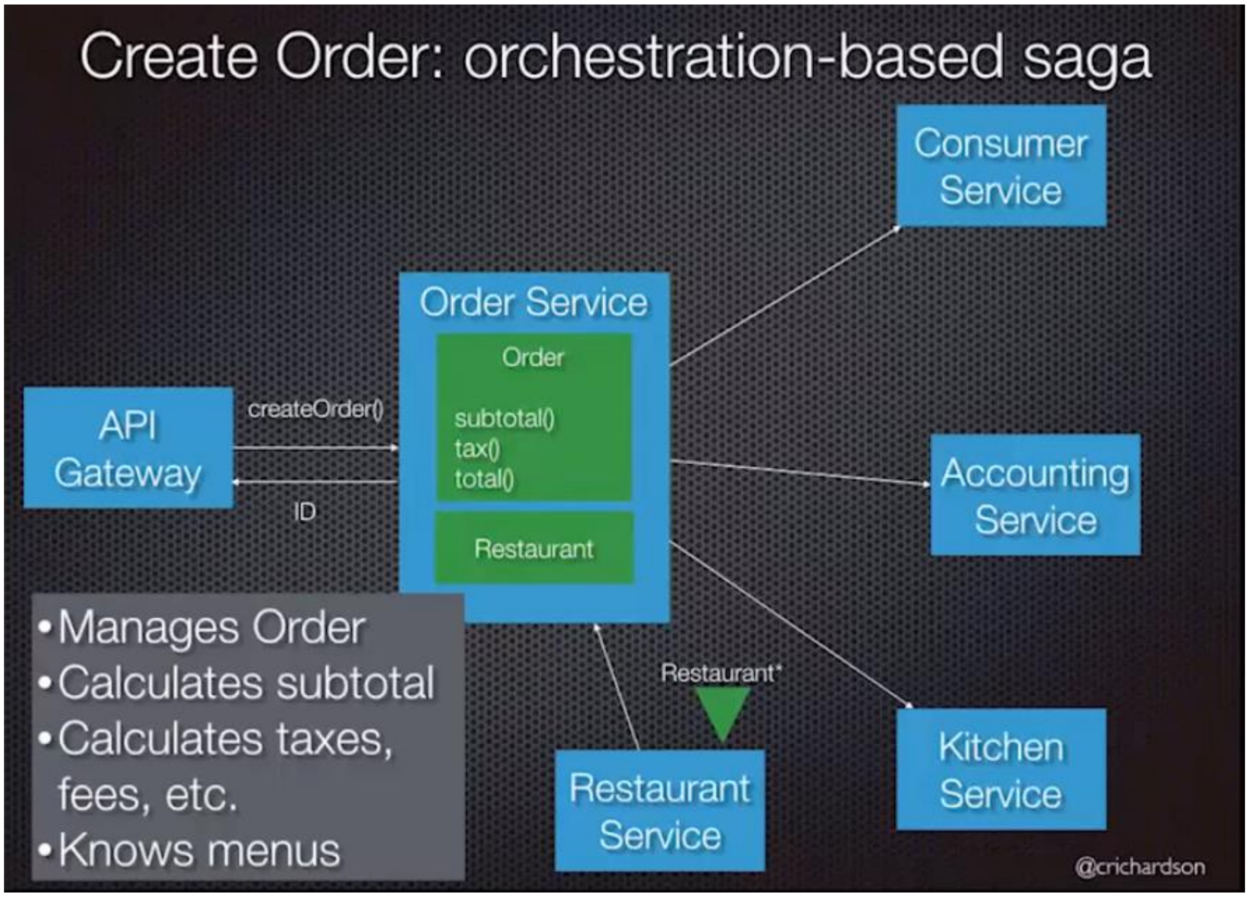
```
class Order {
  Money total;
}
```



• Takeout burritos: a case study in design-time coupling

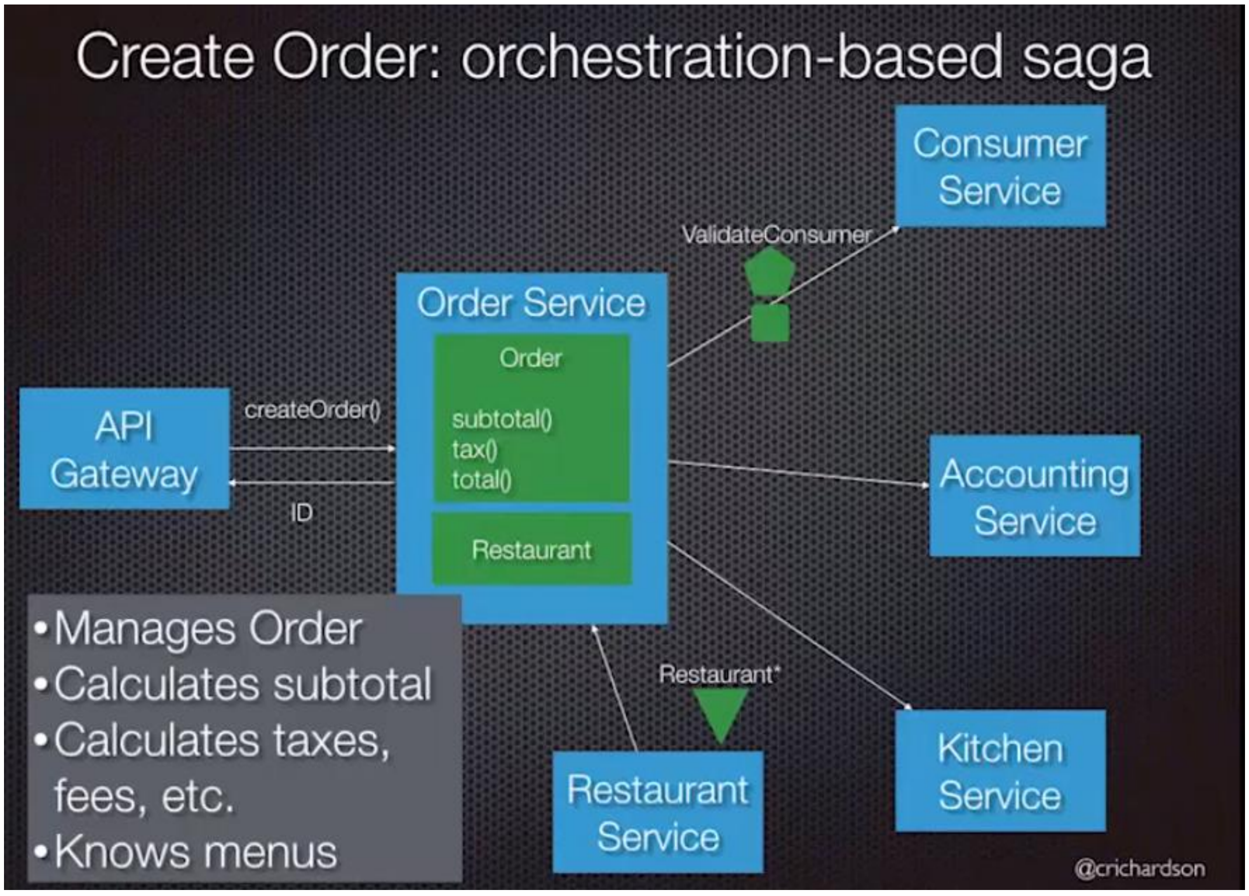
Let us explore how to improve an architecture in order to withstand changing requirements in future





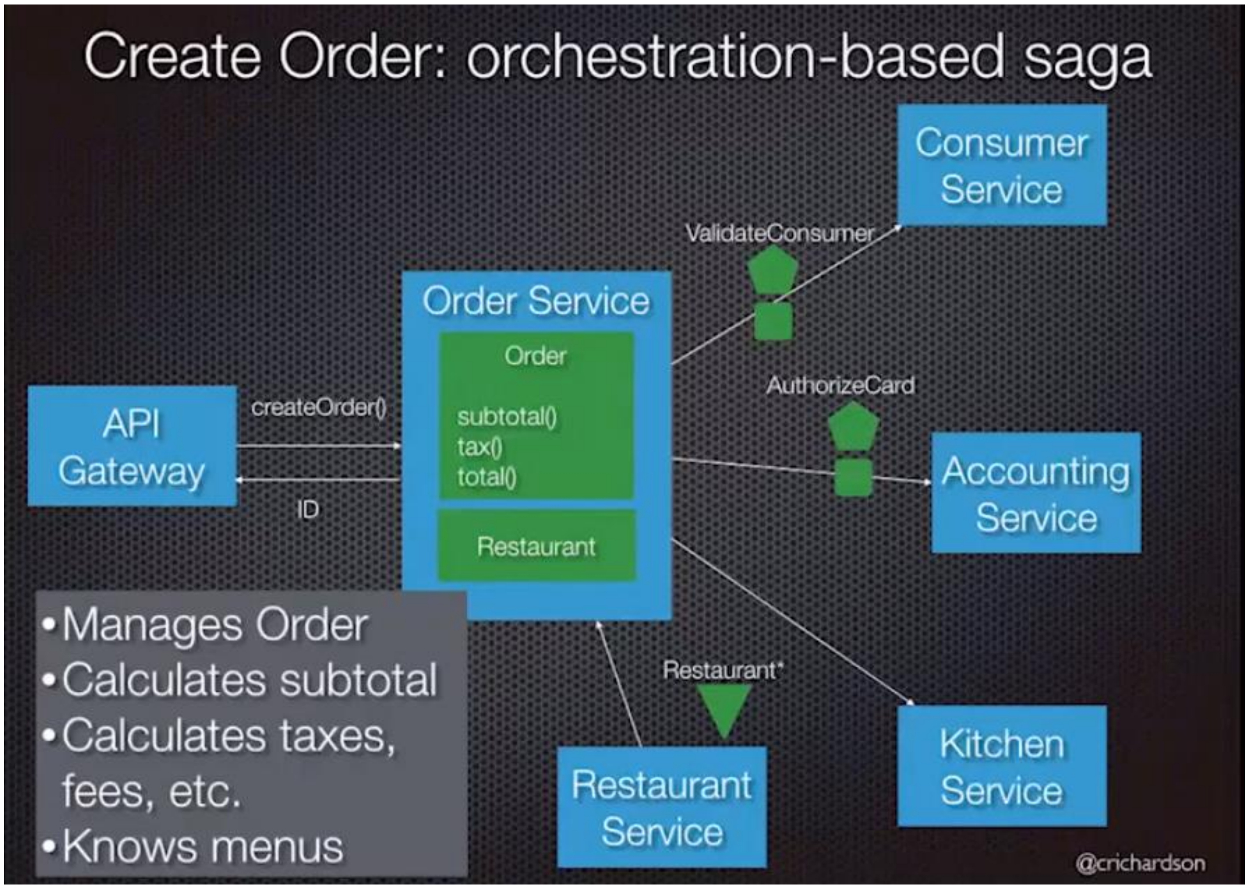
Create Order: orchestration-based saga

- Manages Order
- Calculates subtotal
- Calculates taxes, fees, etc.
- Knows menus



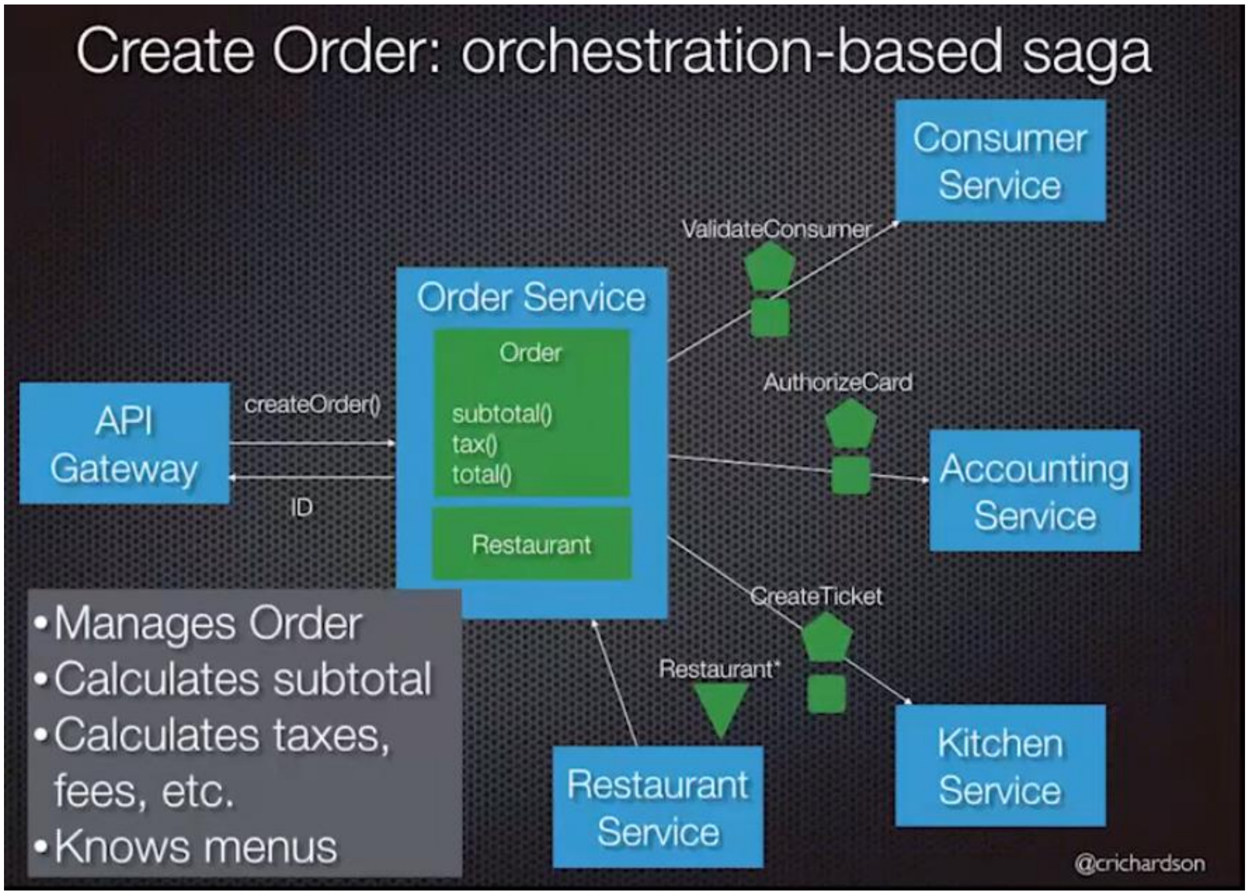
Create Order: orchestration-based saga

- Manages Order
- Calculates subtotal
- Calculates taxes, fees, etc.
- Knows menus



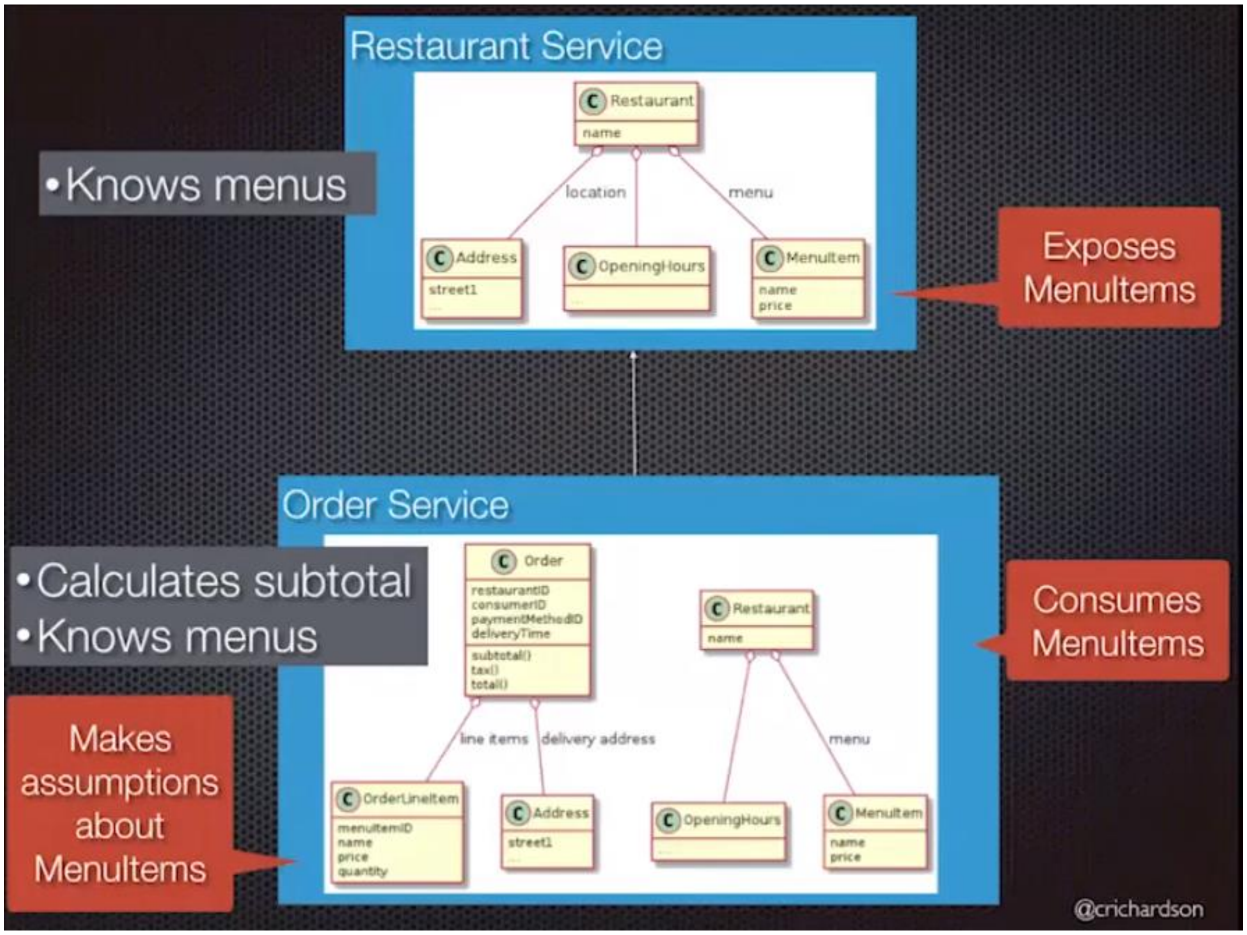
Create Order: orchestration-based saga

- Manages Order
- Calculates subtotal
- Calculates taxes, fees, etc.
- Knows menus

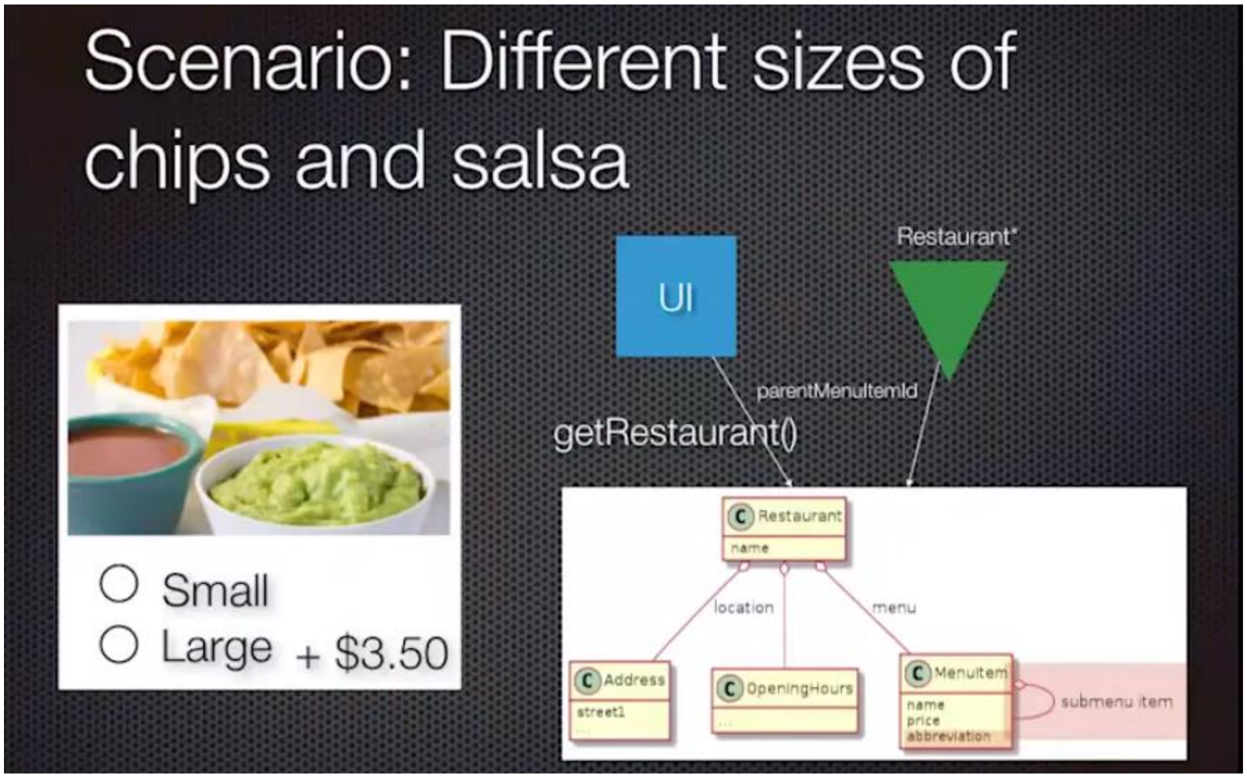


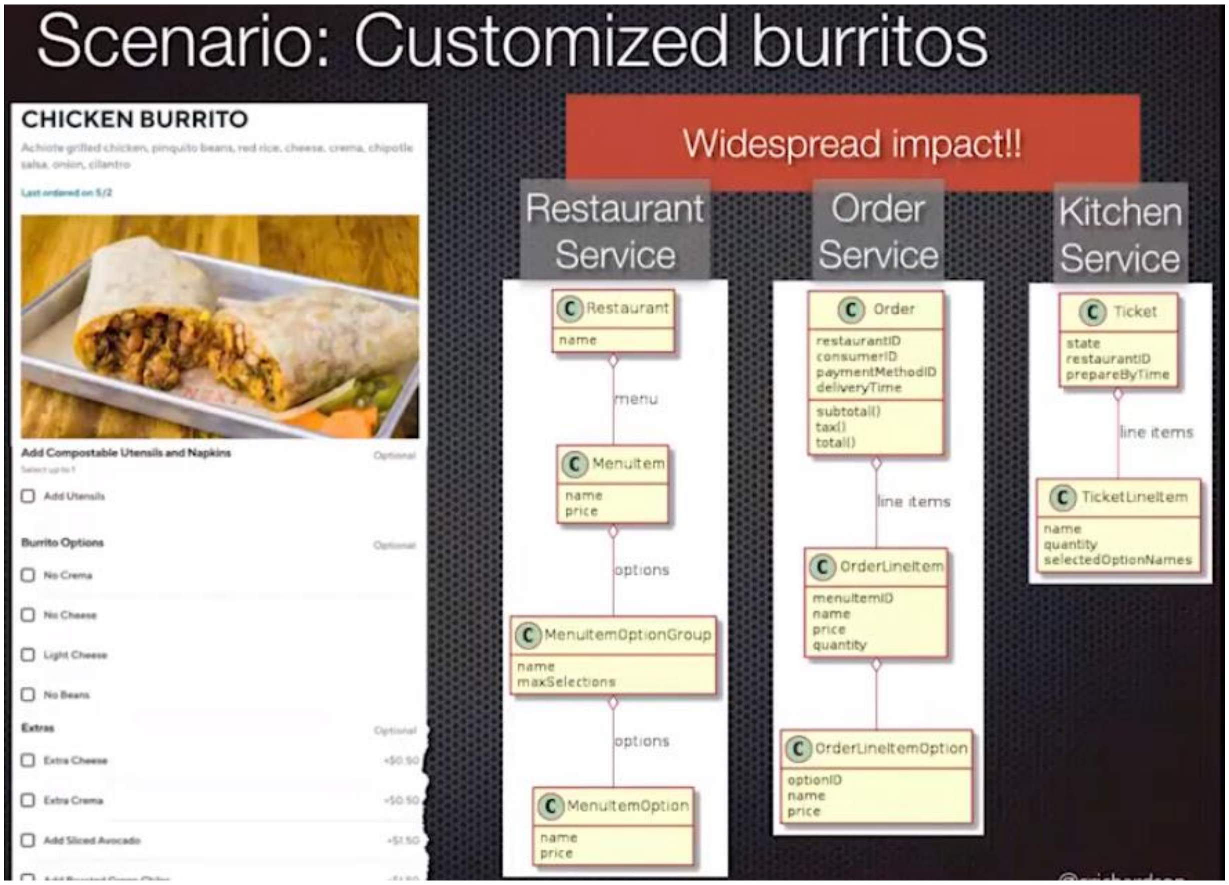
Create Order: orchestration-based saga

- Manages Order
- Calculates subtotal
- Calculates taxes, fees, etc.
- Knows menus



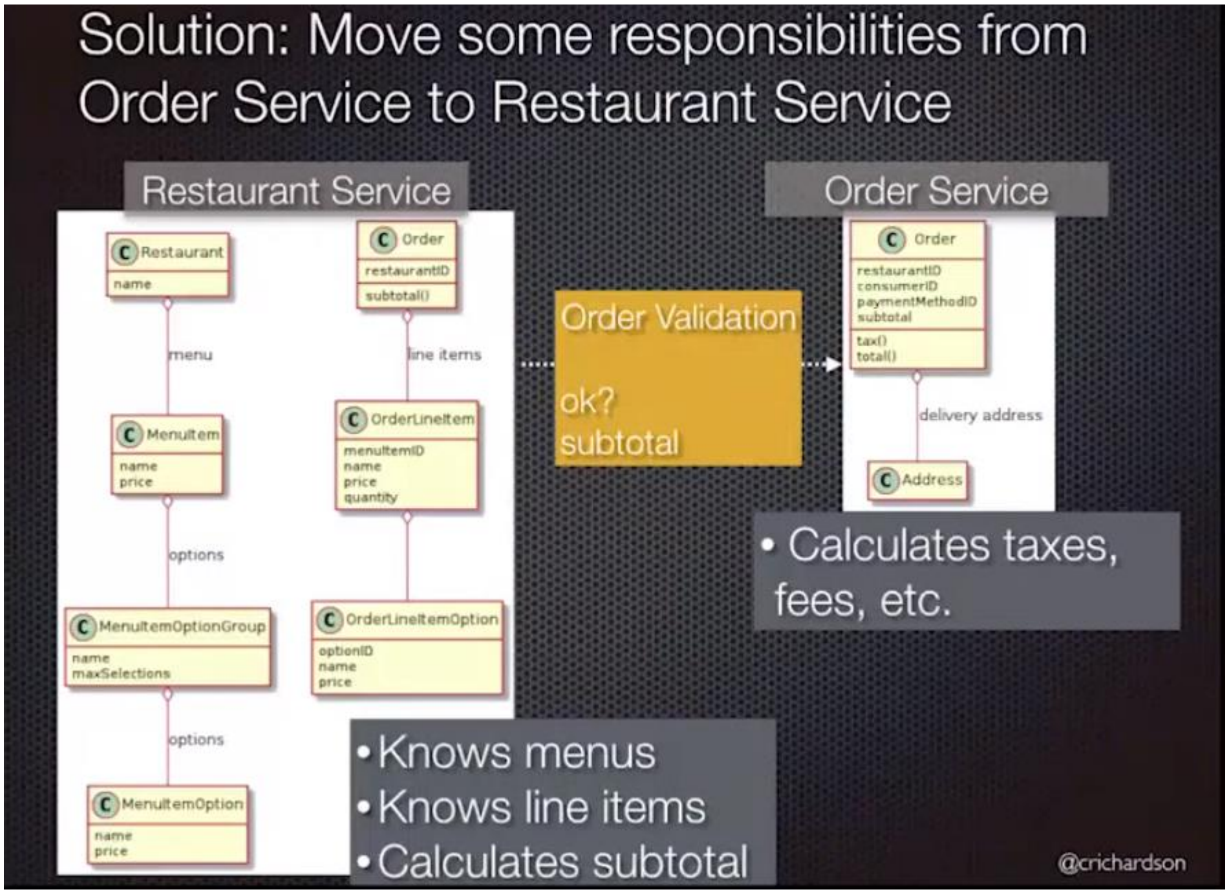
Let us study the design time coupling of the Order service and the Restaurant service.


Scenario: Different sizes of chips and salsa


Scenario: Customized burritos

Widespread impact!!

Concepts that are hidden to the service user can be changed easily, let's see how we can do that for a complex scenario

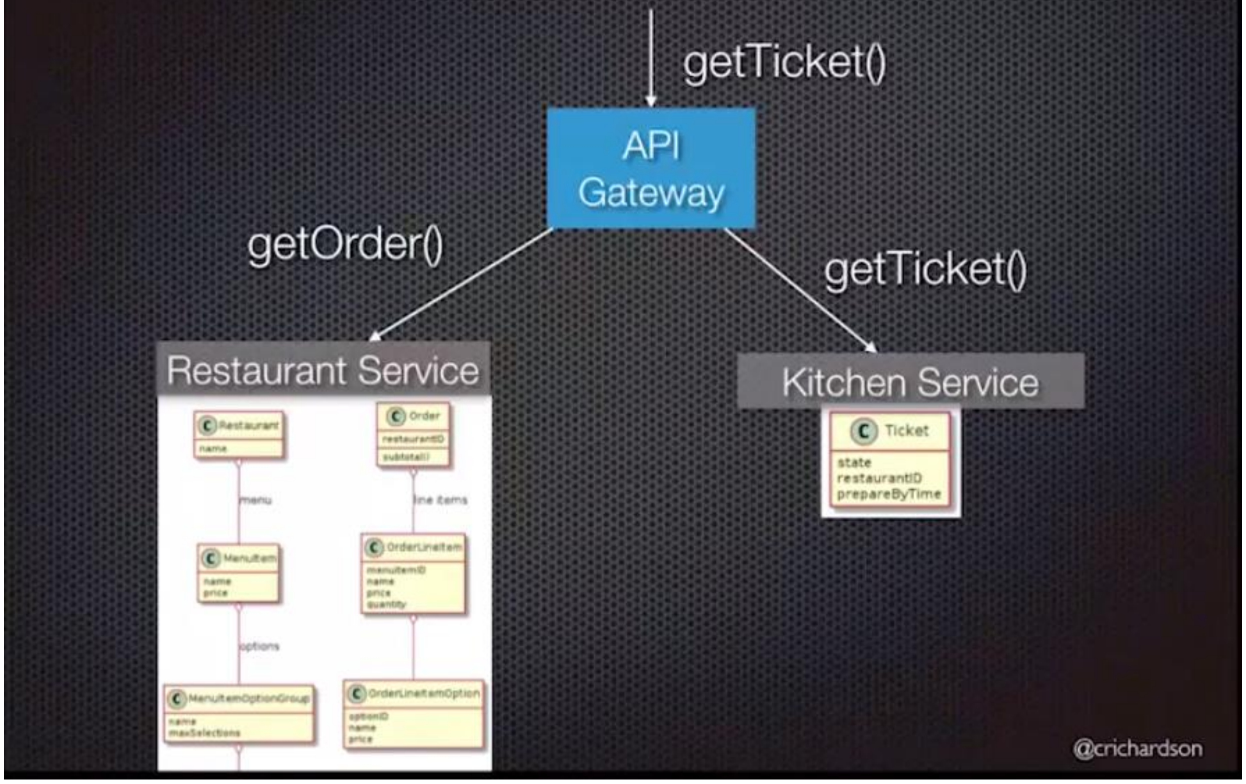

Solution: Move some responsibilities from Order Service to Restaurant Service

- Knows menus
- Knows line items
- Calculates subtotal

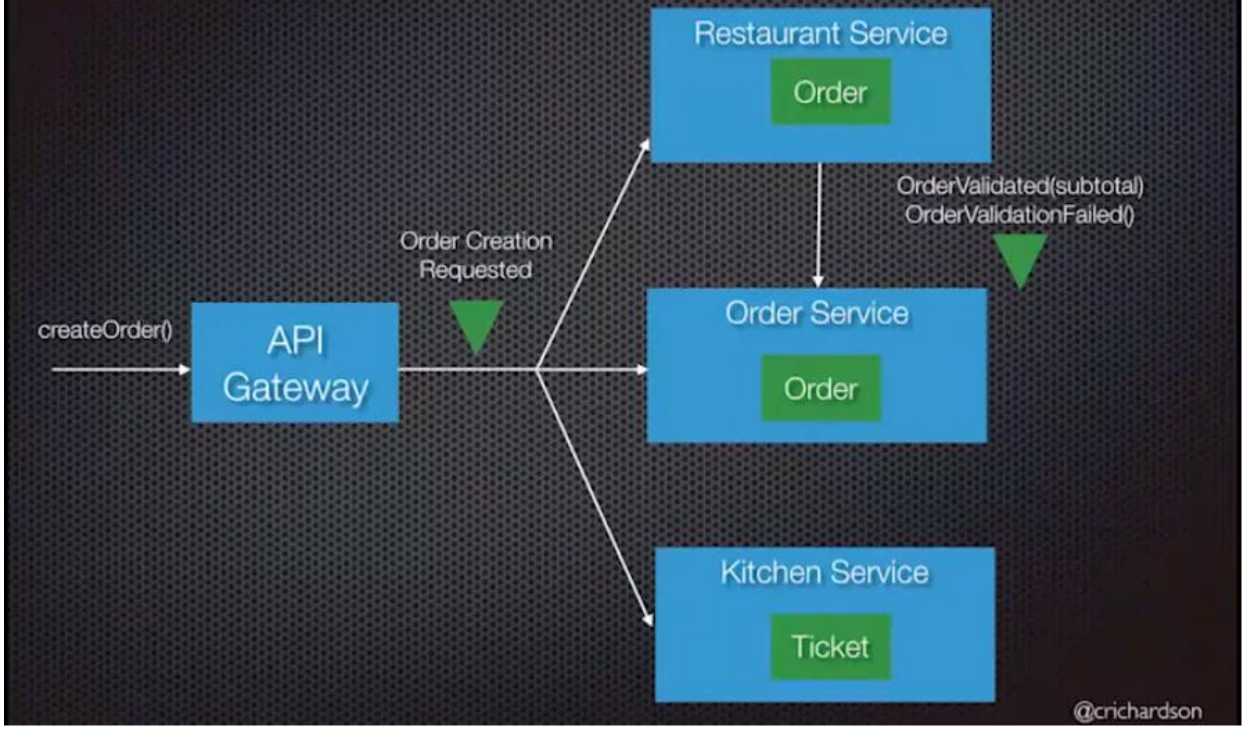
- Calculates taxes, fees, etc.

We have now reduced design time coupled but increased the runtime coupling

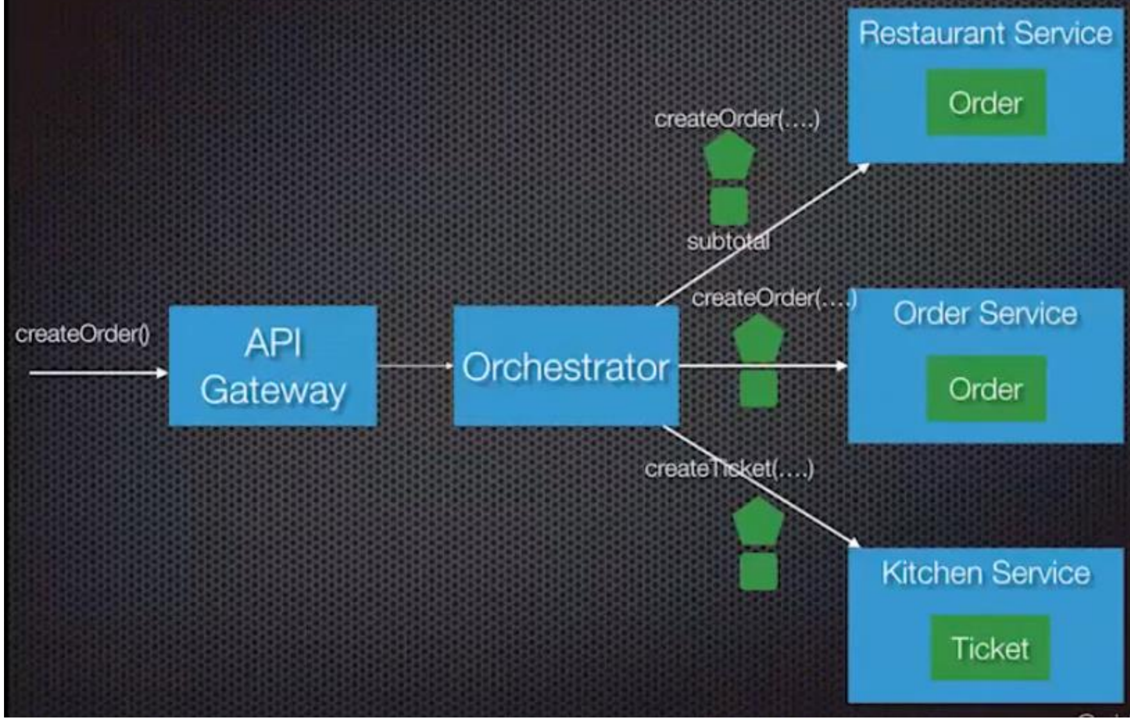
Use API Composition to display a Ticket



Choreography-based coordination



Orchestration-based coordination



Summary

- Rapid and frequent development requires loose design-time coupling
- You must carefully define your services to achieve loose coupling
- Apply the DRY principle
- Design services to be icebergs
- Carefully design service dependencies
- Avoid sharing database tables