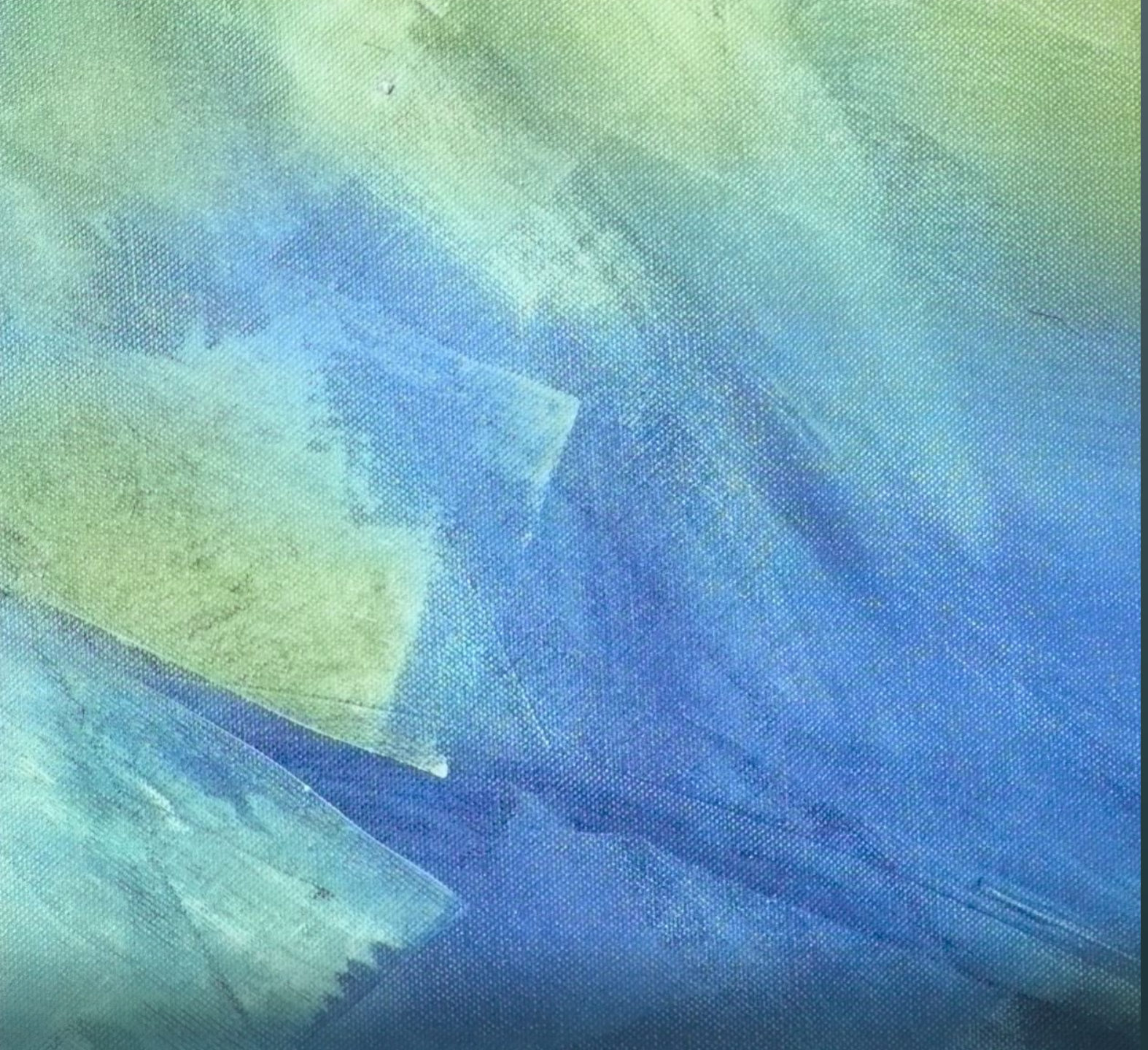


Decomposition Strategies

Wiwat Vatanawood

Duangdao Wichadakul

Nuengwong Tuaycharoen

Pittipol Kantavat

This chapter covers

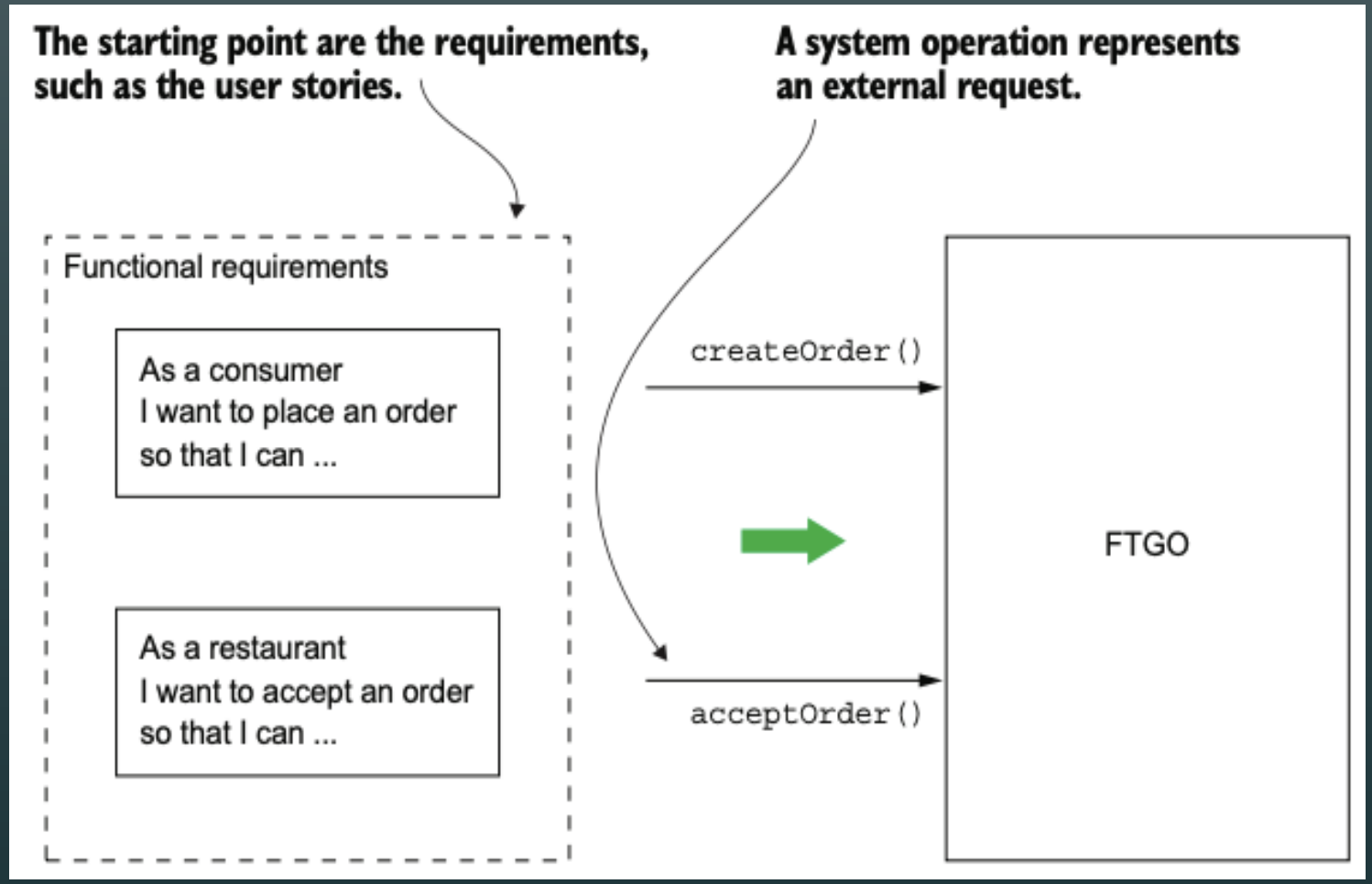
- Decomposing an application into services by applying the decomposition patterns Decompose by business capability and Decompose by subdomain
- Using the bounded context concept from domain-driven design (DDD) to untangle data and make decomposition easier

Defining an application's microservice architecture

- **Three-step process**
- Step 1: Identify system operations
- Step 2: Identify services
- Step 3: Define service APIs and collaborations

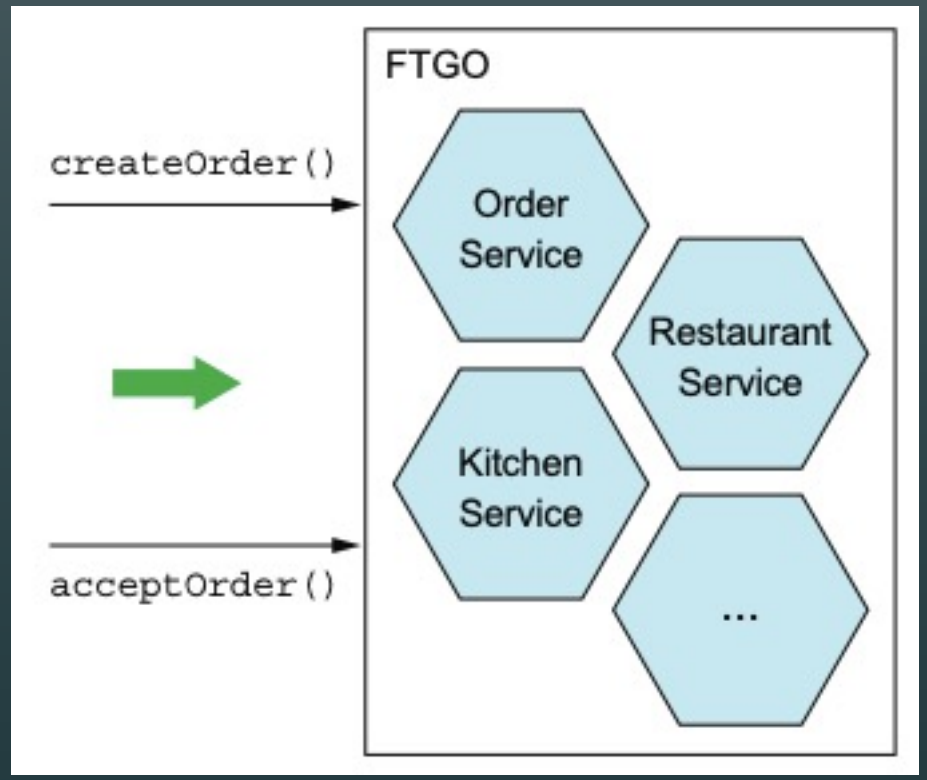
Step 1: Identify system operations

- A system operation is an abstraction of a request that the application must handle
- The system operations become the architectural scenarios that illustrate how the services collaborate.



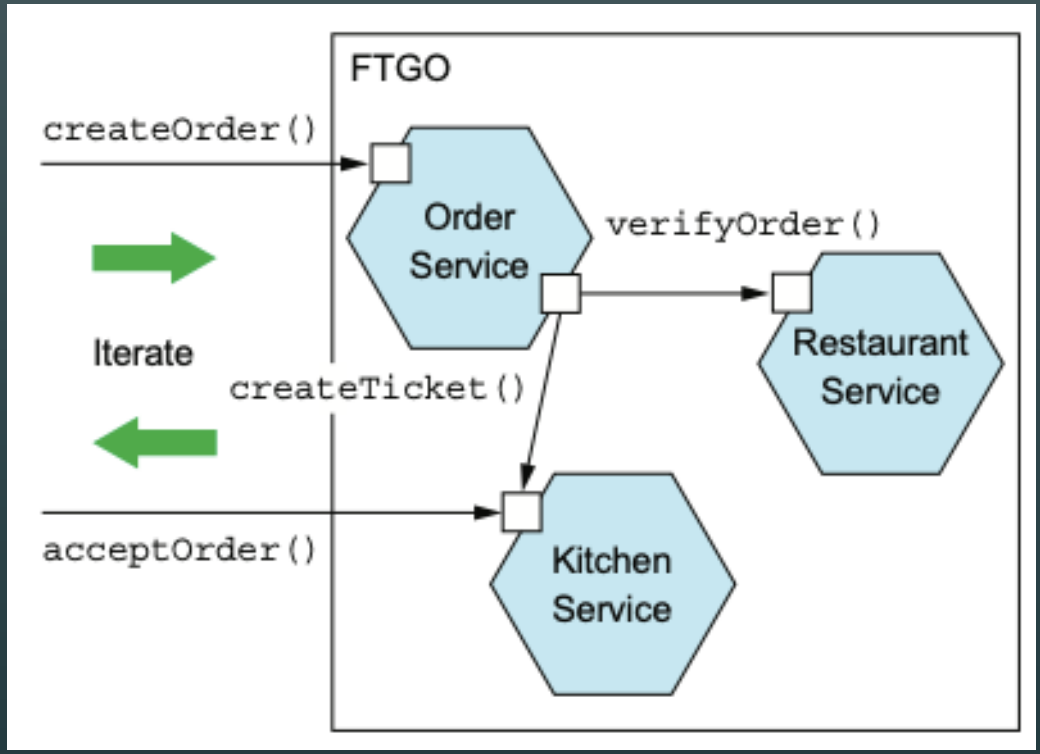
Step 2: Identify services

- Determine the decomposition into services
- There are several strategies
  1. Define services corresponding to business capabilities
  2. organize services around domain-driven design subdomains
- The end result is services that are organized around business concepts rather than technical concepts



Step 3: Define service APIs and collaborations

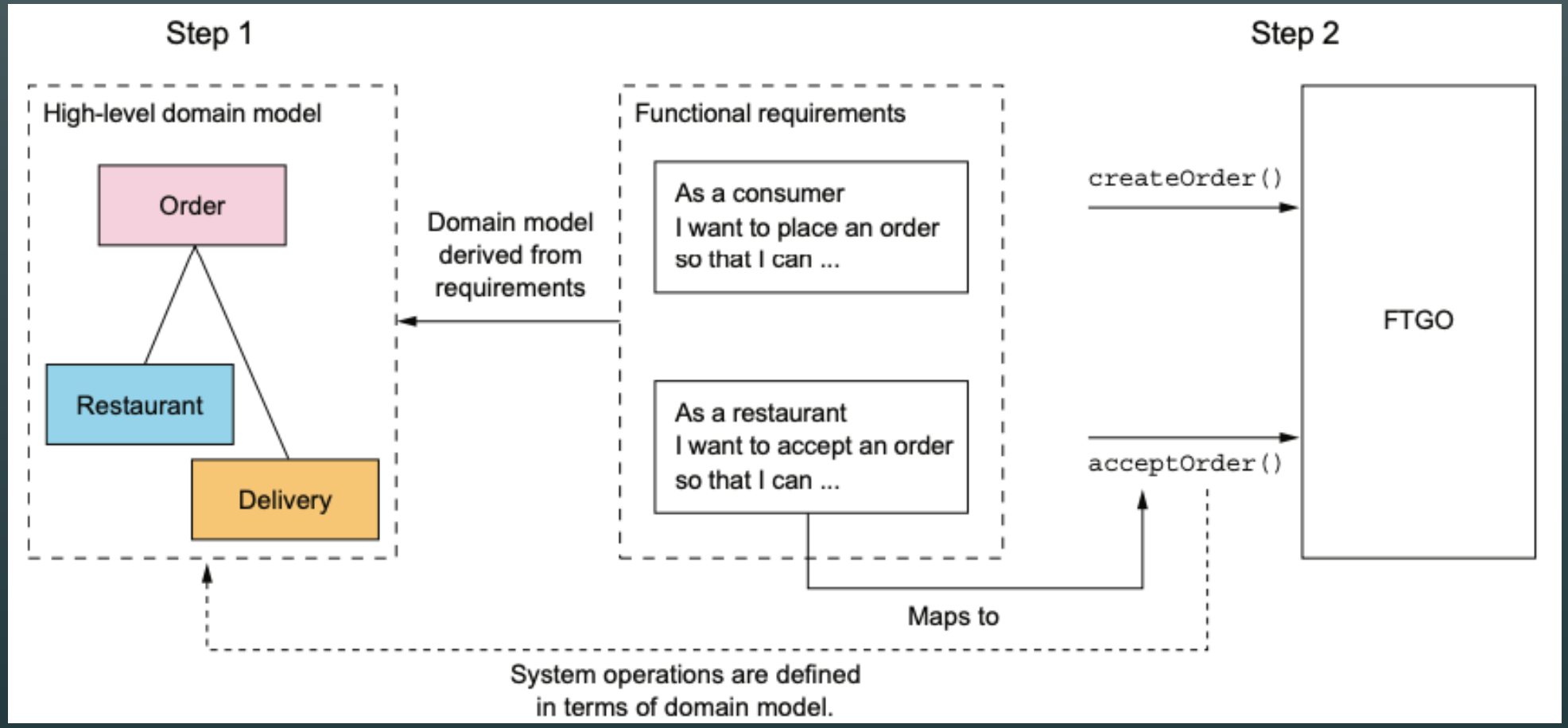
- Determine each service's API
- A service might :-
  1. Implement an operation entirely by itself
  2. Need to collaborate with other services
     (you must determine how the services collaborate)



Defining an application's microservice architecture

- Three-step process
- **Step 1: Identify system operations**
- Step 2: Identify services
- Step 3: Define service APIs and collaborations

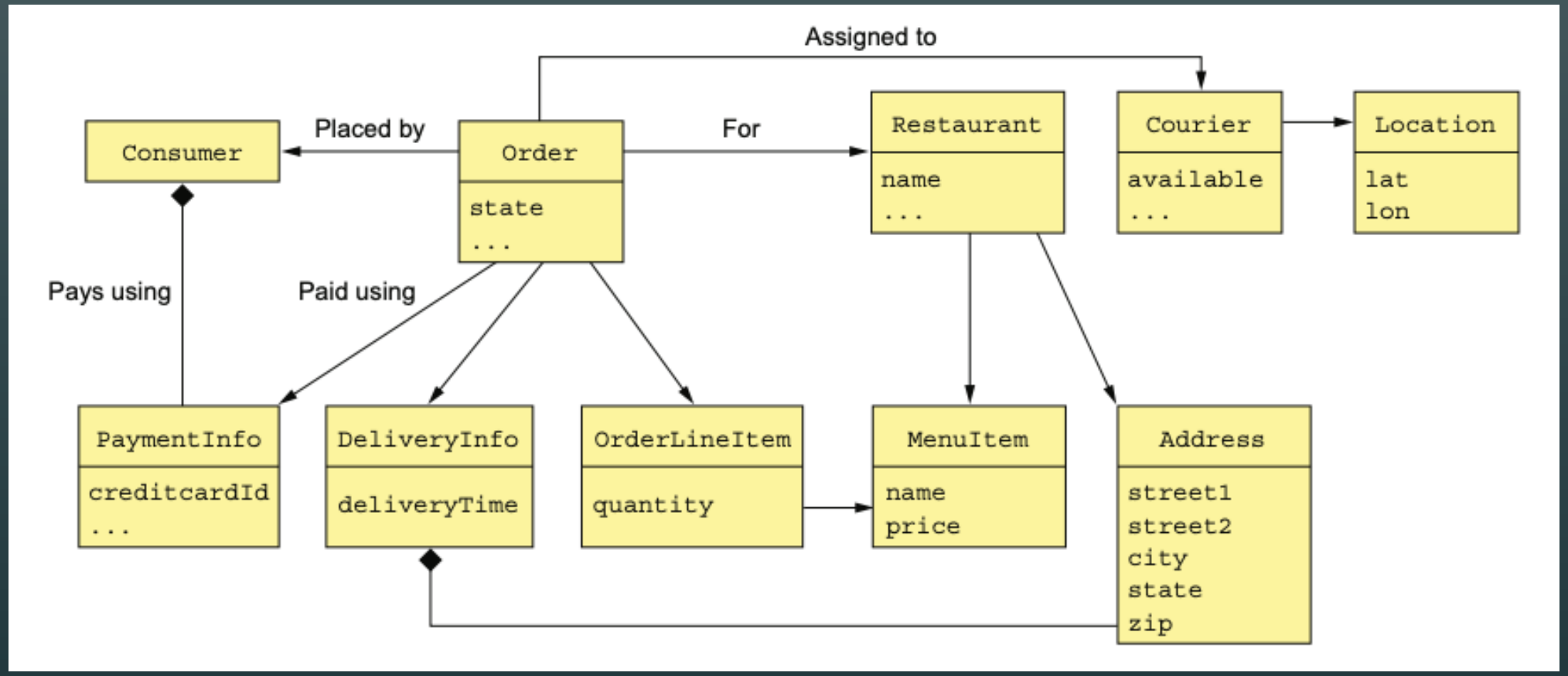
Identifying system operations (1)



Step 1 -- create a high-level domain model.

Step 2 -- define the system operations in terms of the domain model.

Identifying system operations (2)



Identifying system operations (3)

Actor	Story	Command	Description
Consumer	Creates Order	*createsOrder()*	Creates an order
Restaurant	Accept Order	*acceptOrder()*	Indicates that the restaurant has accepted the order and is committed to preparing it by the indicated time
Restaurant	Order Ready for Pickup	*noteOrderReadyForPickup()*	Indicates that the order is ready for pickup
Courier	Update Location	*noteUpdatedLocation()*	Updates the current location of the courier
Courier	Delivery picked up	*noteDeliveryPickedUp()*	Indicates that the courier has picked up the order
Courier	Delivery delivered	*noteDeliveryDelivered()*	Indicates that the courier has delivered the order

Defining an application's microservice architecture

- Three-step process
- Step 1: Identify system operations
- **Step 2: Identify services**
- Step 3: Define service APIs and collaborations

Identifying services by Business Capabilities

- Business capability is something that a business does to generate value.
- Set of capabilities for a given business depends on the kind of business.
- For example :-
  - The capabilities of an insurance company typically include Underwriting, Claims management, Billing, Compliance, and so on.
  - The capabilities of an online store include Order management, Inventory management, Shipping, and so on.

Identifying Business Capabilities

- Business capabilities are identified by analyzing the organization's purpose, structure, and business processes.
- Business-oriented rather than technical
- Often focused on a particular business object.
  - For example, the Claim business object is the Claim management capability.
- A capability can often be decomposed into sub-capabilities.
  - For example, the Claim management capability includes Claim information management, Claim review, and Claim payment management.

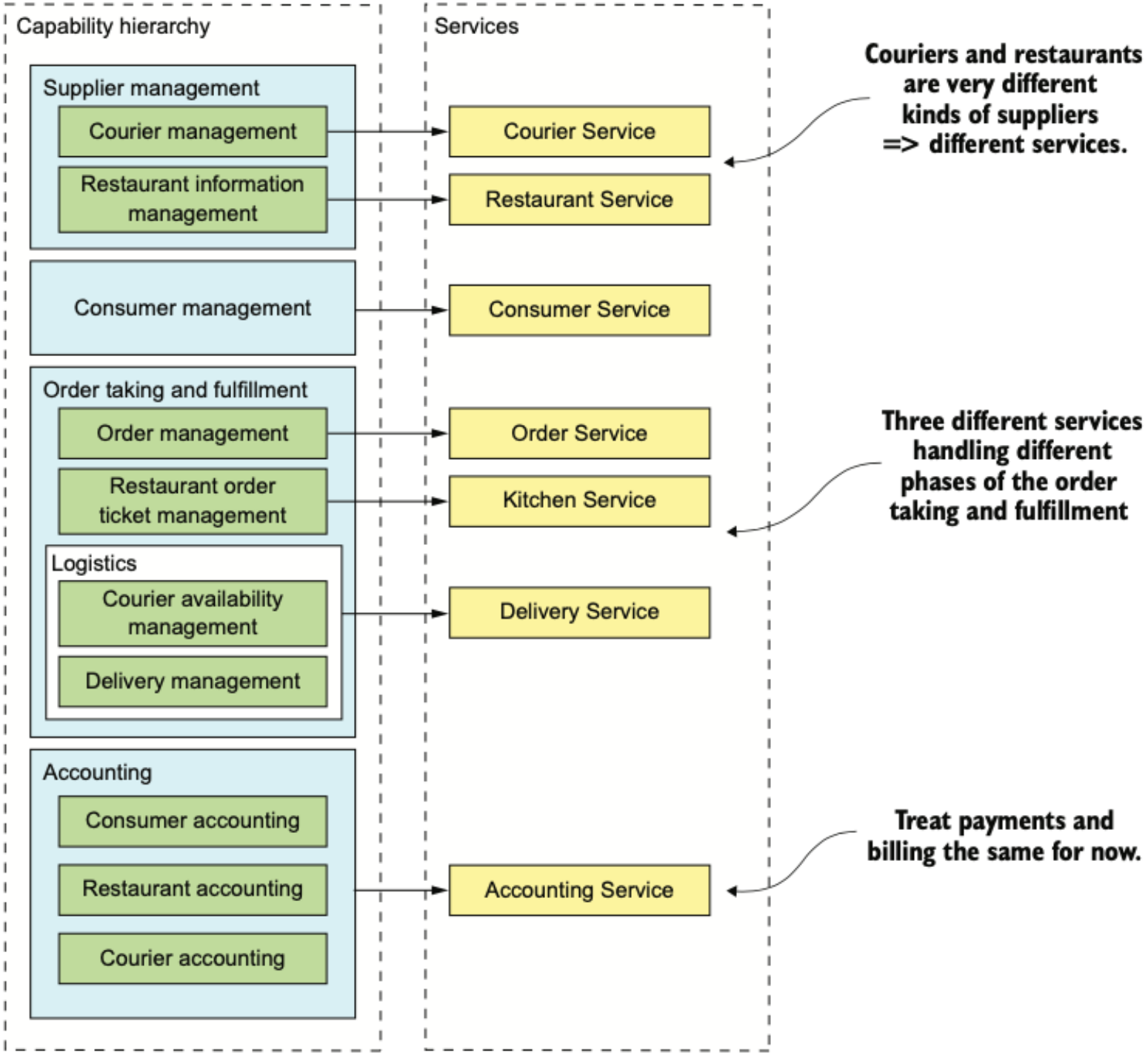
Identifying FTGO Business Capabilities (1)

- Supplier management
  - Courier management —Managing courier information
  - Restaurant information management—Managing restaurant menus and other information, including location and open hours
- Consumer management—Managing information about consumers
- Order taking and fulfillment
  - Order management—Enabling consumers to create and manage orders
  - Restaurant order management—Managing the preparation of orders at a restaurant

Identifying FTGO Business Capabilities (2)

- Logistic
  - Courier availability management—Managing the real-time availability of couriers to delivery orders
  - Delivery management —Delivering orders to consumers
- Accounting
  - Consumer accounting —Managing billing of consumers
  - Restaurant accounting —Managing payments to restaurants
  - Courier accounting —Managing payments to couriers
- …

Mapping FTGO business capabilities to services



Benefits and drawbacks (of business capabilities)

- The resulting architecture are relatively stable.
  - The individual components may evolve as the *how* aspect of the business changes, but the architecture remains unchanged.
- However, in defining service collaboration,
  - Some decompositions are inefficient due to excessive in interprocess communication
  - Some services might grow in complexity to the point where it becomes worthwhile to split it into multiple services

Identifying services by domain-driven design subdomains (1)

- Domain-driven design (DDD) is an approach for building complex software applications that is centered on the development of an object-oriented domain model
  - The domain model is mirrored in the design and implementation of the application
- DDD has two concepts that are incredibly useful when applying the microservice architecture: (1) subdomains and (2) bounded contexts

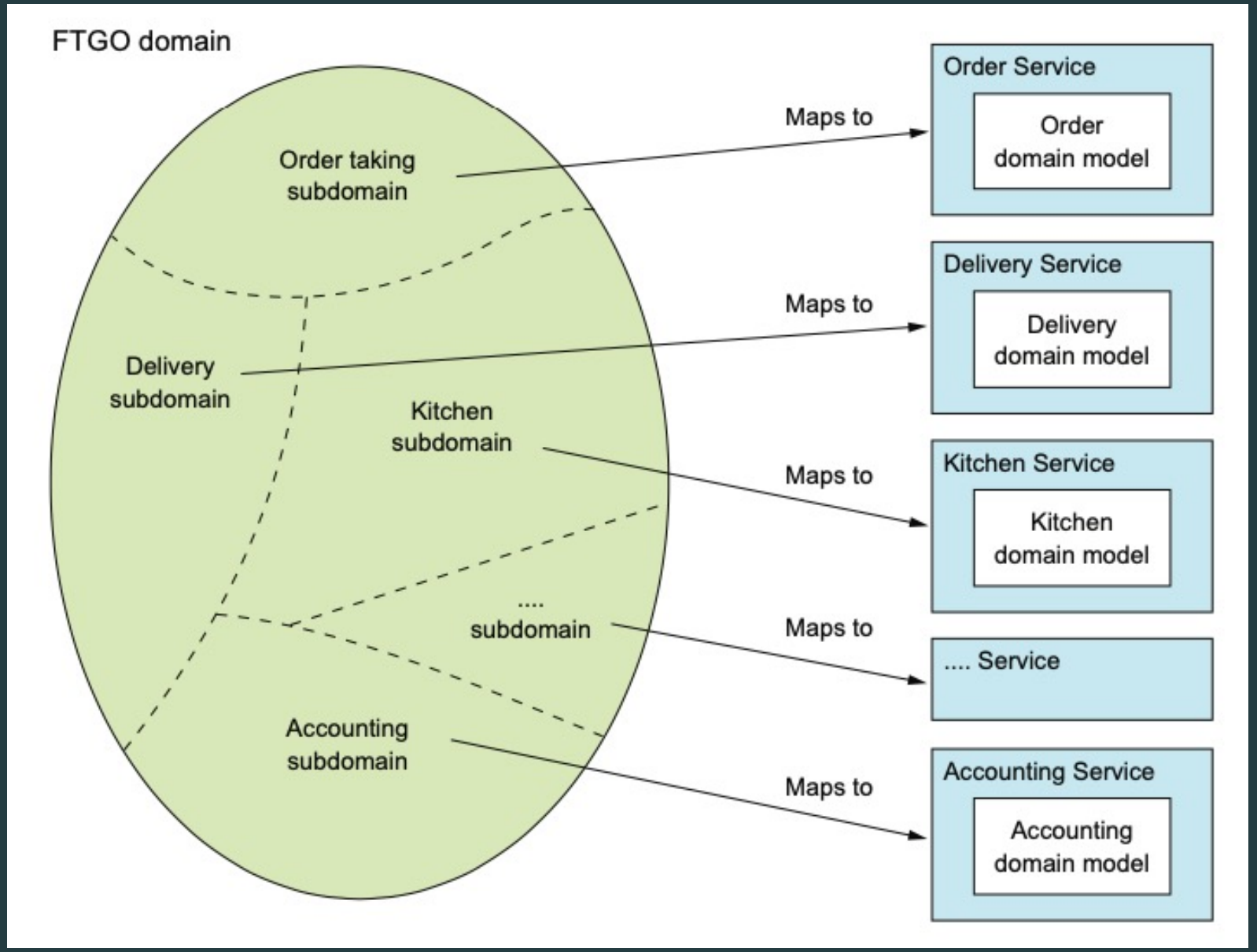
Identifying services by domain-driven design subdomains (2)

- DDD defines a separate domain model for each subdomain.
  - Subdomains are identified using the same approach as identifying business capabilities:
  - Analyze the business and identify the different areas of expertise
- DDD calls the scope of a domain model a bounded context.
  - When using the microservice architecture, each bounded context is a service or possibly a set of services

From subdomains to services:

each subdomain of the FTGO application domain is mapped to a service, which has its own domain model.

- DDD and the microservice architecture are in almost perfect alignment.



Obstacles to decomposing an application into services (1)

- Network latency
  - Can be reduced by implementing a batch API for fetching multiple objects in a single round trip
- Reduced availability due to synchronous communication
  - The most straightforward way to implement is using REST but it reduces the availability
  - Using asynchronous messaging, which eliminates tight coupling and improves availability

Obstacles to decomposing an application into services (2)

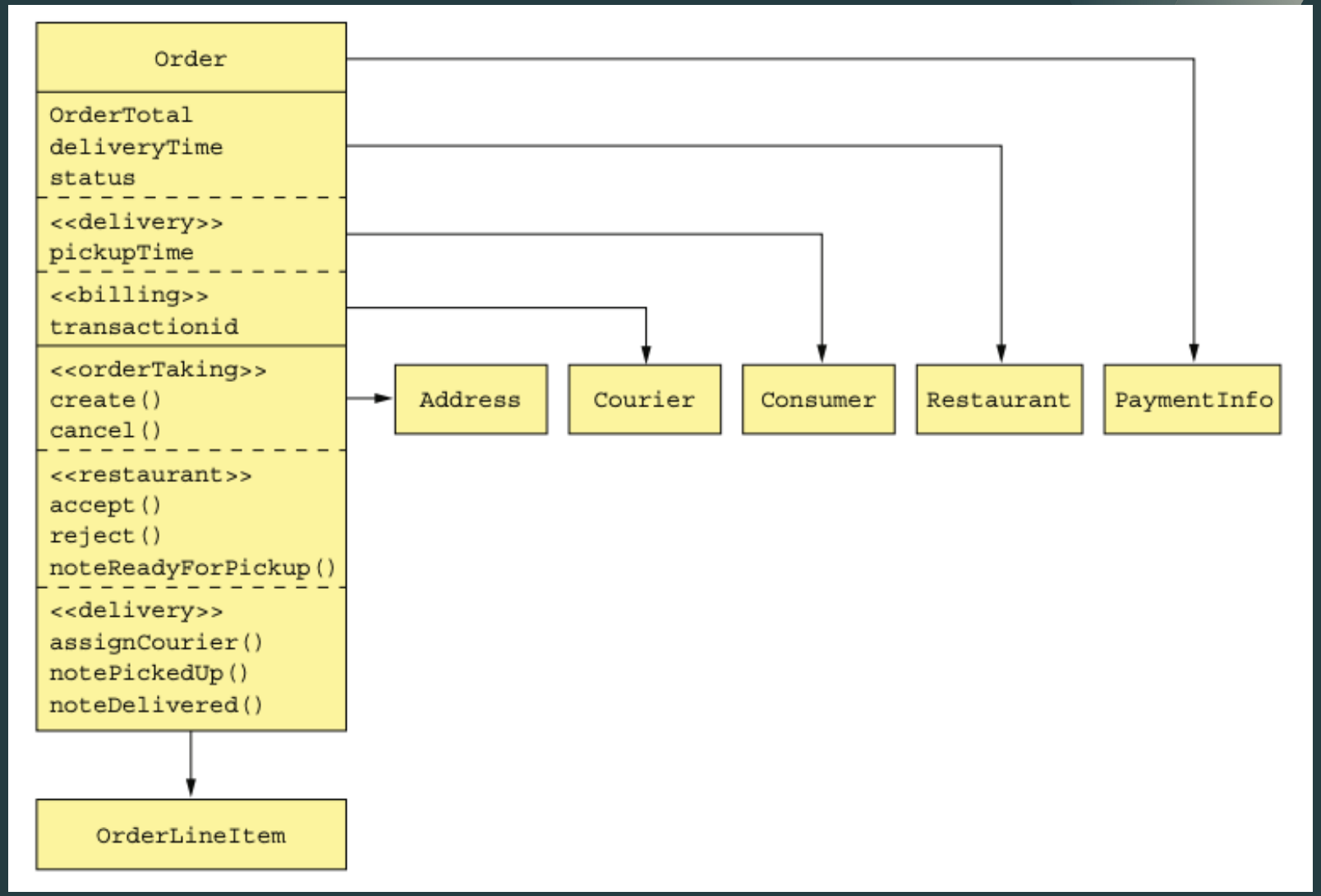
- Maintaining data consistency across services
  - Sagas are more complex than traditional ACID transaction, but they work well in many situations
- Obtaining a consistent view of the data
  - Inability to obtain a consistent view of data across multiple databases
  - Fortunately, in practice this is rarely a problem.

Obstacles to decomposing an application into services (3)

- God classes preventing decomposition
  - God classes are the bloated classes used throughout an application
  - Typically implements business logic for many aspects of the application
  - Normally has a large number of fields mapped to a database table with many columns
  - Most applications have at least one of these classes
  - For example, orders in e-commerce, policies in insurance, and so on.

The Order, a god class in FTGO

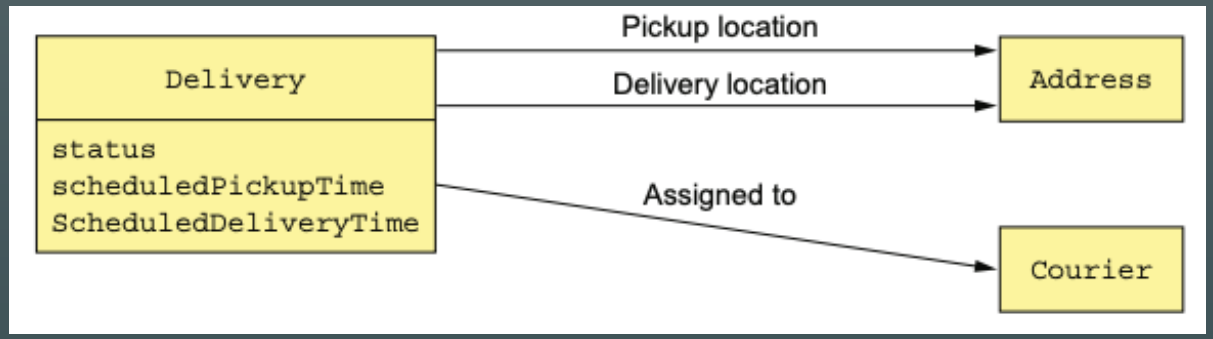
- The purpose of FTGO is to deliver food orders to customers.
- If the FTGO application had a single domain model, the Order class would be a very large class



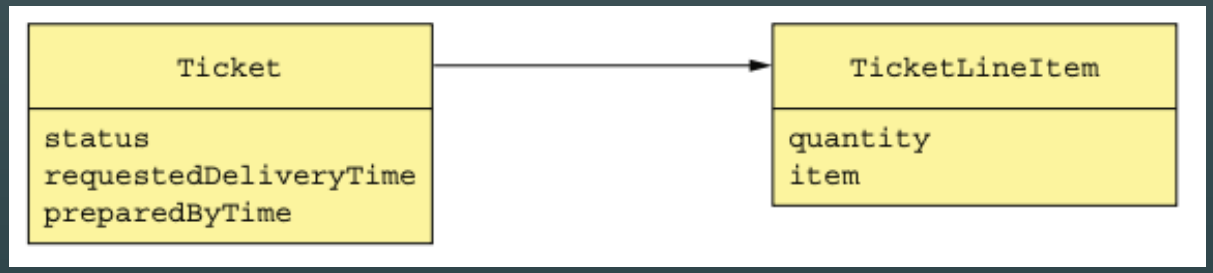
The Order god class is bloated with numerous responsibilities

Applying DDD is a solution.

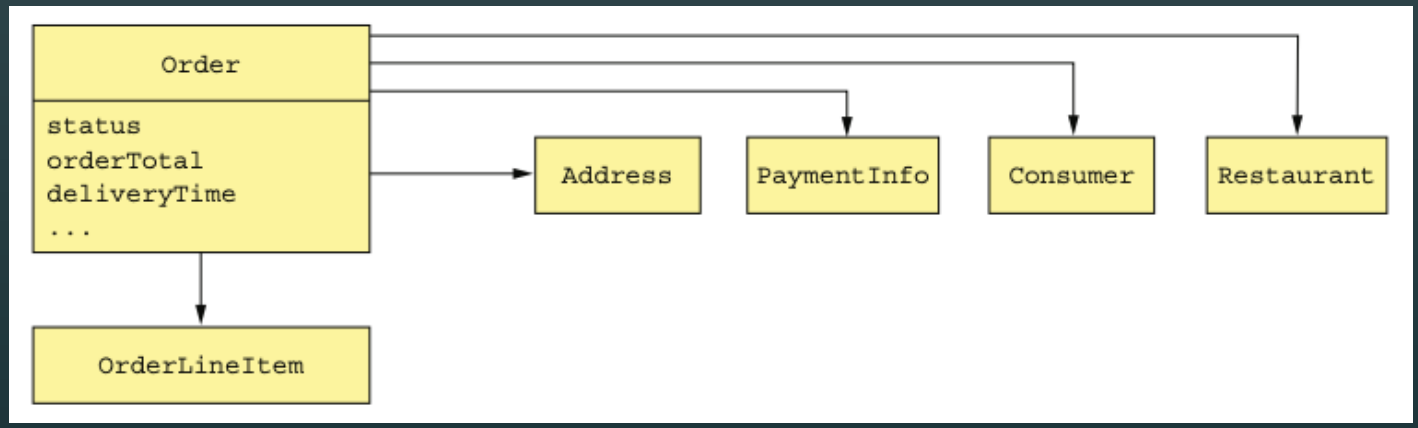
- A much better approach is to apply DDD.
- Treat each service as a separate sub- domain with its own domain model.
- The Order class in each domain model represents different aspects of the same Order business entity



The *Delivery Service* domain model



The *Kitchen Service* domain model



The *Order Service* domain model

Defining an application's microservice architecture

- Three-step process
- Step 1: Identify system operations
- Step 2: Identify services
- **Step 3: Define service APIs and collaborations**

Defining service APIs

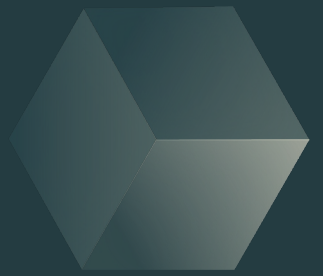
- Defining the service APIs is to
  1. Map each system operation to a service.
  2. Then, decide whether a service needs to collaborate with others to implement a system operation.
  3. If collaboration is required, then determine what APIs those other services must provide in order to support the collaboration

1 >> Map each system operation to a service

- The first step is to decide which service is the initial entry point for a request
- Table shows which services in the FTGO application are responsible for which operations

Service	Operations
Consumer Service	createConsumer()
Order Service	createOrder()
Restaurant Service	findAvailableRestaurants()
Kitchen Service	▪ acceptOrder() ▪ noteOrderReadyForPickup()
Delivery Service	▪ noteUpdatedLocation() ▪ noteDeliveryPickedUp() ▪ noteDeliveryDelivered()

2 and 3 >> Determine the APIs required to support collaboration between services



- Some system operations are handled entirely by a single service
  - For example, the Consumer Service handles the *createConsumer()*
- But other system operations span multiple services.
- The data might be scattered around multiple services
  - For example, in order to implement the *createOrder()* operation, the Order Service must invoke some other services

Service	Operations	Collaborators
Consumer Service	verifyConsumerDetails()	—
Order Service	createOrder()	■ Consumer Service verifyConsumerDetails() ■ Restaurant Service verifyOrderDetails() ■ Kitchen Service createTicket() ■ Accounting Service authorizeCard()
Restaurant Service	■ findAvailableRestaurants() ■ verifyOrderDetails()	—
Kitchen Service	■ createTicket() ■ acceptOrder() ■ noteOrderReadyForPickup()	■ Delivery Service scheduleDelivery()
Delivery Service	■ scheduleDelivery() ■ noteUpdatedLocation() ■ noteDeliveryPickedUp() ■ noteDeliveryDelivered()	—
Accounting Service	■ authorizeCard()	—

What's next?

- Selected any specific interprocess communication (IPC) technology
  - synchronous communication mechanisms such as REST
  - asynchronous messaging using a message broker
  - self-contained service using the CQRS pattern
- Apply the saga concept to uses asynchronous messaging for coordinating the services
- Implement a query operation using the API composition pattern

Summary

- Services in a microservice architecture are organized around business concerns— business capabilities or subdomains—rather than technical concerns.
- There are two patterns for decomposition:
  - Decompose by business capability, which has its origins in business architecture
  - Decompose by subdomain, based on concepts from domain-driven design
- You can eliminate god classes by applying DDD and defining a separate domain model for each service.