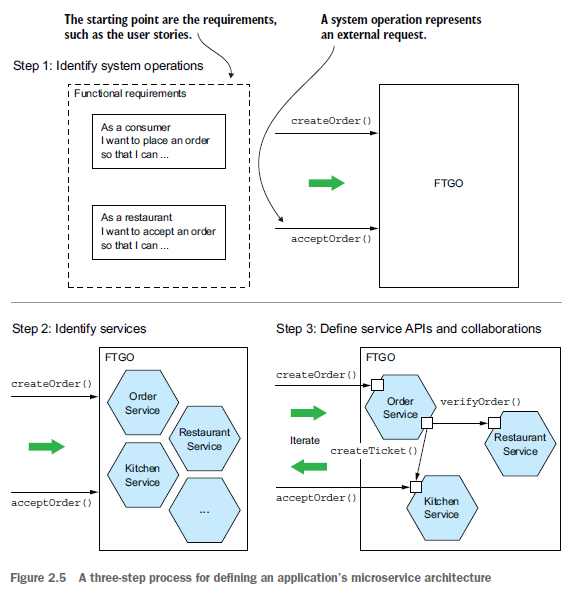
Defining an Application’s Microservices Architecture

This document describes a simple, three-step process, shown in the figure below, for defining an application’s architecture. It’s important to remember, though, that it’s not a process you can follow mechanically. **It’s likely to be iterative and involve a lot of creativity.**



1. An application exists to handle requests, so the first step in defining its architecture is to distill the application’s requirements into the key requests.

Instead of specific IPC(inter-process) technologies like REST or messaging, in this stage we use the more abstract notion of **System Operation**: an abstraction of a request that the application must handle. It’s either a command, which updates data, or a query, which retrieves data. The behavior of each command is defined in terms of an abstract domain model, which is also derived from the requirements. **The system operations become the architectural scenarios that illustrate how the services collaborate.**

1. The second step in the process is to determine the decomposition into services.

There are several strategies to choose from. One strategy, which has its origins in the discipline of business architecture, is to **define services corresponding to business capabilities**. Another strategy is to organize services **around domain-driven design subdomains**. The end result is **services that are organized around business concepts rather than technical concepts.**

1. The third step in defining the application’s architecture is to determine each service’s API.

To do that, **you assign each system operation identified in the first step to a service**. A service might implement an operation entirely by itself. Alternatively, it might need to collaborate with other services. In that case, you determine how the services collaborate, which typically requires services to support additional operations. You’ll also need to decide which of the IPC mechanisms I describe in another document to implement each service’s API.

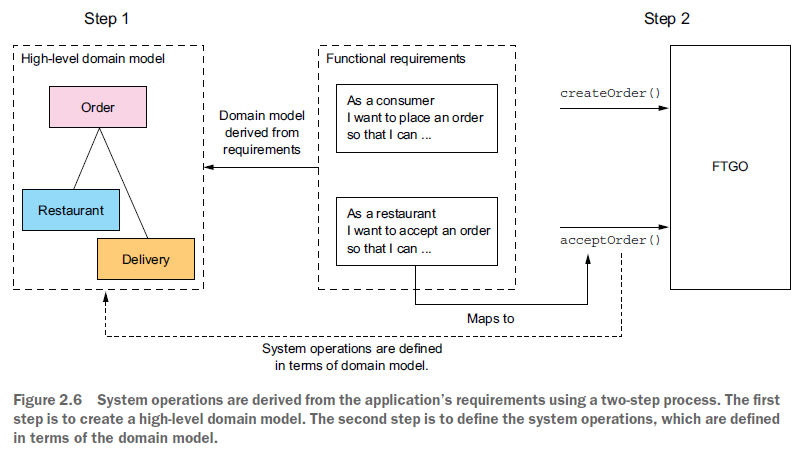
There are several **obstacles to decomposition:**

* **network latency:** You might discover that a particular decomposition would be impractical due to too many round-trips between services.
* **synchronous communication between services reduces availability**. You might need to use the concept of self-contained services, described in chapter 3.
* **the requirement to maintain data consistency across services**. You’ll typically need to use sagas, discussed in chapter 4.
* **god classes**, which are used throughout an application. Fortunately, you can use concepts from domain-driven design to eliminate god classes.

# Identifying System Operations

**The first step** in defining an **application’s architecture** is to define **the system operations**. **The starting point is the application’s requirements**, including user stories and their associated user scenarios (note that these are different from the architectural scenarios).

The system operations are identified and defined using the two-step process shown in the figure below:



The first step creates the high-level domain model consisting of the **key classes that provide a vocabulary with which to describe the system operations.** The second step identifies the system operations and describes each one’s behavior **in terms of the domain model.**

* The domain model is derived primarily from the nouns of the user stories, and the system operations are derived mostly from the verbs. You could also define the domain model using a technique called Event Storming, which I talk about in chapter 5 of the book.
* The behavior of each system operation is described in terms of its **effect on one or more domain objects and the relationships between them.** A system operation can create, update, or delete domain objects, as well as create or destroy relationships between them.

## Creating a High-Level Domain Model

Note that this domain model is much simpler than what will ultimately be implemented. The application won’t even have a single domain model because, as you’ll soon learn, each service has its own domain model. Despite being a drastic simplification, **a high-level domain model is useful at this stage because it defines the vocabulary for describing the behavior of the system operations.**

domain model is created using standard techniques such as analyzing the nouns in the stories and scenarios and talking to the domain experts. Consider, for example, the **Place Order story**. **We can expand that story into numerous user scenarios including this one:**

Given a consumer

And a restaurant

And a delivery address/time that can be served by that restaurant

And an order total that meets the restaurant's order minimum

When the consumer places an order for the restaurant

Then consumer's credit card is authorized

And an order is created in the PENDING\_ACCEPTANCE state

And the order is associated with the consumer

And the order is associated with the restaurant

The nouns in this user scenario hint at the existence of various classes, including Consumer, Order, Restaurant, and CreditCard.

Similarly, the **Accept Order story** can be expanded into a scenario such as this one:

Given an order that is in the PENDING\_ACCEPTANCE state

and a courier that is available to deliver the order

When a restaurant accepts an order with a promise to prepare by a particular

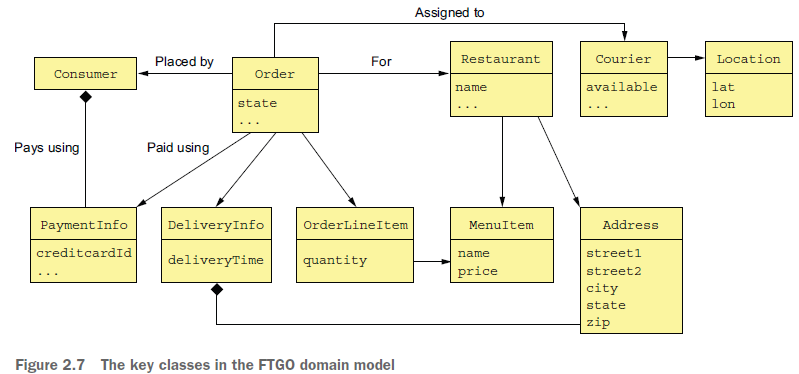
time

Then the state of the order is changed to ACCEPTED

And the order's promiseByTime is updated to the promised time

And the courier is assigned to deliver the order

This scenario suggests the existence of Courier and Delivery classes. The end result after a few iterations of analysis will be a domain model that consists, unsurprisingly, of those classes and others, such as MenuItem and Address. The class diagram below shows the key classes:



The responsibilities of each class are as follows:

 Consumer—A consumer who places orders.

 Order—An order placed by a consumer. It describes the order and tracks its status.

 OrderLineItem—A line item of an Order.

 DeliveryInfo—The time and place to deliver an order.

 Restaurant—A restaurant that prepares orders for delivery to consumers.

 MenuItem—An item on the restaurant’s menu.

 Courier—A courier who deliver orders to consumers. It tracks the availability of

the courier and their current location.

 Address—The address of a Consumer or a Restaurant.

 Location—The latitude and longitude of a Courier.

A class diagram such as the one in figure 2.7 **illustrates one aspect of an application’s architecture.** But it isn’t much more than a pretty **picture without the scenarios to animate it**. The next step is to define **the system operations, which correspond to architectural scenarios.**

## Defining System Operations

Once you’ve defined a high-level domain model, the next step is to identify the requests that the application must handle.

At this point, as different clients might use different communication protocols, we use a more abstract notion of System Operations as described before. Ultimately, these system operations will correspond to REST, RPC, or messaging endpoints, but for now thinking of them abstractly is useful.

Let’s first identify some commands.

A good starting point for identifying system commands is to **analyze the verbs** in the

user stories and scenarios. Consider, for example, the Place Order story. It clearly suggests

that the system must provide a Create Order operation. Many other stories individually

map directly to system commands:

|  |  |  |  |
| --- | --- | --- | --- |
| Actor | Story | Command | Description |
| Consumer | Create Order | createOrder() | 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 |

A command has a specification that defines its parameters, return value, and behavior in terms of the domain model classes. The behavior specification consists of preconditions that must be true when the operation is invoked and mirror the ***given*** part of the user scenario, and post-conditions that are true after the operation is invoked that show ***then*** in the user scenarios. When a system operation is invoked, it **will verify the preconditions** and perform the actions required **to make the post-conditions true**.

Here, for example, is the specification of the createOrder()and acceptOrder() system operations:

**Operation**: createOrder (consumer id, payment method, delivery address, delivery time, restaurant id, order line items)

**Returns**: orderId, …

**Preconditions**:

 The consumer exists and can place orders.

 The line items correspond to the restaurant’s menu items.

 The delivery address and time can be serviced by the restaurant.

**Post-conditions:**

 The consumer’s credit card was authorized for the order total.

 An order was created in the PENDING\_ACCEPTANCE state.

**Operation:** acceptOrder(restaurantId, orderId, readyByTime)

**Returns:** —

**Preconditions:**

 The order.status is PENDING\_ACCEPTANCE.

 A courier is available to deliver the order.

**Post-conditions:**

 The order.status was changed to ACCEPTED.

 The order.readyByTime was changed to the readyByTime.

 The courier was assigned to deliver the order.

Besides implementing commands, an application must also implement queries. The queries provide the UI with the information a user needs to make decisions. At this stage, we don’t have a particular UI design for FTGO application in mind, but consider, for example, the flow when a consumer places an order:

1 User enters delivery address and time.

2 System displays available restaurants.

3 User selects restaurant.

4 System displays menu.

5 User selects item and checks out.

6 System creates order.

This user scenario suggests the following queries:

 findAvailableRestaurants(deliveryAddress, deliveryTime)—Retrieves the restaurants that can deliver to the specified delivery address at the specified time

 findRestaurantMenu(id)—Retrieves information about a restaurant including the menu items

Of the two queries, findAvailableRestaurants() is probably the most architecturally significant:

* It’s a complex query involving geosearch. The geosearch component of the query consists of finding all points—restaurants—that are near a location—the delivery address.
* It also filters out those restaurants that are closed when the order needs to be prepared and picked up.
* **Moreover, performance is critical, because this query is executed whenever a consumer wants to place an order.**

The high-level domain model and the system operations capture what the application does. They help drive the definition of the application’s architecture. The behavior of each system operation is described in terms of the domain model. Each important system operation represents an architecturally significant scenario that’s part of the description of the architecture.

Once the system operations have been defined, the next step is to identify the application’s services. As mentioned earlier, **there isn’t a mechanical process to follow**. There are, however, various **decomposition strategies** that you can use. Each one attacks the problem from a different perspective and uses its own terminology. **But with all strategies, the end result is the same: an architecture consisting of services that are primarily organized around business rather than technical concepts.** Let’s look at the first strategy, which defines services corresponding to business capabilities.

# Identifying Services

## Defining Services by Applying the Decompose by business capability pattern

One strategy for creating a microservice architecture is to decompose by business capability. A concept from business architecture modeling, **a *business capability* is something that a business does in order to generate value**. The 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.**

### Business Capabilities Define What an Application Does

An organization’s business capabilities capture *what* an organization’s business is. **They’re generally stable, as opposed to *how* an organization conducts its business**, which changes over time, sometimes dramatically. That’s especially true today, with the rapidly growing use of technology to automate many business processes.

For example, it wasn’t that long ago that you deposited checks at your bank by handing them to a teller. It then became possible to deposit checks using an ATM. Today you can conveniently deposit most checks using your smartphone. **As you can see, the Deposit check business capability has remained stable, but the manner in which it’s done has drastically changed.**

### Identifying Business Capabilities

An organization’s business capabilities are identified by analyzing the organization’s purpose, structure, and business processes. **Each business capability can be thought of as a service, except it’s business-oriented rather than technical.**

**A business capability is often focused on a particular business object**. For example, **the Claim business object** is the focus of the **Claim management capability**. A capability can often be decomposed **into sub-capabilities**. For example, **the Claim management capability has several sub-capabilities,** including Claim information management, Claim review, and Claim payment management.

the business capabilities for FTGO include the following:

 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

– Logistics

– *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

 …

The top-level capabilities include **Supplier management**, **Consumer management**, **Order taking and fulfillment**, **and Accounting**. There will likely be many other top-level capabilities, including marketing-related capabilities. Most top-level capabilities are decomposed into sub-capabilities. For example, Order taking and fulfillment is decomposed into five sub-capabilities.

On interesting aspect of this capability hierarchy is that there are three restaurant-related capabilities: Restaurant information management, Restaurant order management, and Restaurant accounting. That’s because they represent three very different aspects of restaurant operations.

### From Business capabilities to Services

Once you’ve identified the business capabilities, you then define a service for **each capability** **or** **group of related capabilities.**

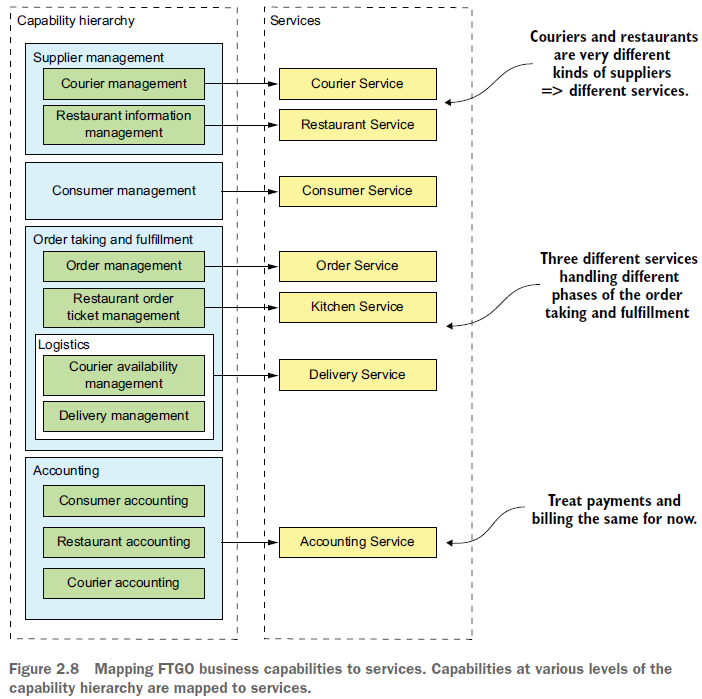
Figure 2.8 shows the mapping from capabilities to services for the FTGO application. **Some top-level capabilities**, such as the Accounting capability, **are mapped to services. In other cases, sub-capabilities are mapped to services.**

The decision of which level of the capability hierarchy to map to services is somewhat subjective. My justification for this particular mapping is as follows:

 I mapped the sub-capabilities of Supplier management to two services, because Restaurants and Couriers are very different types of suppliers.

 I mapped the Order taking and fulfillment capability to three services that are each responsible for different phases of the process. I combined the Courier availability management and Delivery management capabilities and mapped them to a single service because they’re deeply intertwined.

 I mapped the Accounting capability to its own service, because the different types of accounting seem similar.



Later on, it may make sense to separate payments (of Restaurants and Couriers) and billing (of Consumers)**. A key benefit of organizing services around capabilities is that because they’re stable, the resulting architecture will also be relatively stable. The individual components of the architecture may evolve as the *how* aspect of the business changes, but the architecture remains unchanged.**

Having said that, it’s important to remember that the services shown in figure 2.8 are merely the first attempt at defining the architecture. They may evolve over time as we learn more about the application domain. In particular, an important step in the architecture definition process is investigating how the services collaborate in each of the key architectural services.

You might, for example, discover that a particular decomposition is inefficient due to excessive inter-process communication and that you must combine services. Conversely, a service might grow in complexity to the point where it becomes worthwhile to split it into multiple services. What’s more, in section 2.2.5, I describe several obstacles to decomposition that might cause you to revisit your decision.

Let’s take a look at another way to decompose an application that is based on

domain-driven design.

## Defining Services by Applying the Decompose by Sub-Domain Pattern

## Decomposition Guidelines

So far in this chapter, we’ve looked at the main ways to define a microservice architecture. We can also adapt and use a couple of principles from object-oriented design when applying the microservice architecture pattern. These principles were created by Robert C. Martin and described in his classic book *Designing Object Oriented C++* *Applications Using The Booch Method* (Prentice Hall, 1995). The first principle is the Single Responsibility Principle (SRP), for defining the responsibilities of a class. The second principle is the Common Closure Principle (CCP), for organizing classes into packages. Let’s take a look at these principles and see how they can be applied to the microservice architecture:

### Single Responsibility Principle

*“A class should have only one reason to change.”*

Each responsibility that a class has is a potential reason for that class to change. If a class has multiple responsibilities that change independently, the class won’t be stable. By following the SRP, you define classes that each have a single responsibility and hence a single reason for change.

We can apply SRP when defining a microservice architecture and create small, cohesive services that each have a single responsibility. **This will reduce the size of the services** and **increase their stability**. The new FTGO architecture is an example of SRP in action. Each aspect of getting food to a consumer—order taking, order preparation, and delivery—is the responsibility of a separate service.

### Common Closure Principle

*“The classes in a package should be closed together against the same kinds of changes. A change that affects a package affects all the classes in that package.”*

**The idea is that if two classes change in lockstep because of the same underlying reason, then they belong in the same package.** Perhaps, for example, those classes implement a different aspect of a particular business rule. The goal is that when that business rule changes, developers only need to change code in a small number of packages (ideally only one). **Adhering to the CCP significantly improves the maintainability of an application.**

We can apply CCP when creating a microservice architecture and **package components that change for the same reason into the same service.** Doing this will minimize the number of services that need to be changed and deployed when some requirement changes. Ideally, a change will only affect a single team and a single service**. CCP is the antidote to the distributed monolith anti-pattern.**

Decomposition by business capability and by subdomain along with SRP and CCP are good techniques for decomposing an application into services. **In order to apply them and successfully** develop a microservice architecture, **you must solve some transaction management** and **inter-process communication issues.**

## Obstacles of Decomposing an Application into Services

On the surface, the strategy of creating a microservice architecture by defining services corresponding to business capabilities or subdomains looks straightforward. You may, however, encounter several obstacles:

### Network Latency

*Network latency* is an ever-present concern in a distributed system. You might discover that a particular decomposition into services results in a large number of round-trips between two services. Sometimes, **you can reduce the latency to an acceptable amount by implementing a batch API for fetching multiple objects in a single round trip**. **But in other situations, the solution is to combine services, replacing expensive IPC with language-level method or function calls.**

### Synchronous Interprocess Communication Reduces Availability

Another problem is how to implement interservice communication in a way that doesn’t reduce availability. For example, the most straightforward way to implement the createOrder() operation is for the Order Service to synchronously invoke the other services using REST. The drawback of using a protocol like **REST** is that it reduces the availability of the Order Service**. It won’t be able to create an order if any of those other services are unavailable.** Sometimes this is a worthwhile trade-off, but in chapter 3 you’ll learn **that using asynchronous messaging, which eliminates tight coupling and improves availability, is often a better choice.**

### Maintaining Data Consistency Across Services

Another challenge is maintaining data consistency across services. **Some system operations need to update data in multiple services.** For example, when a restaurant accepts an order, updates must occur in both the Kitchen Service and the Delivery Service. The Kitchen Service changes the status of the Ticket. The Delivery Service schedules delivery of the order. Both of these updates must be done atomically. The traditional solution is to use a two-phase, commit-based, distributed transaction management mechanism. But as you’ll see in chapter 4, this is not a good choice for modern applications, and you must use a very different approach to transaction management, a saga**. A *saga* is a sequence of local transactions that are coordinated using messaging**. Sagas are more complex than traditional ACID transactions but they work well in many situations. **One limitation of sagas is that they are eventually consistent**. **If you need** to update some data **atomically, then it must reside within a single service, which can be an obstacle to decomposition.**

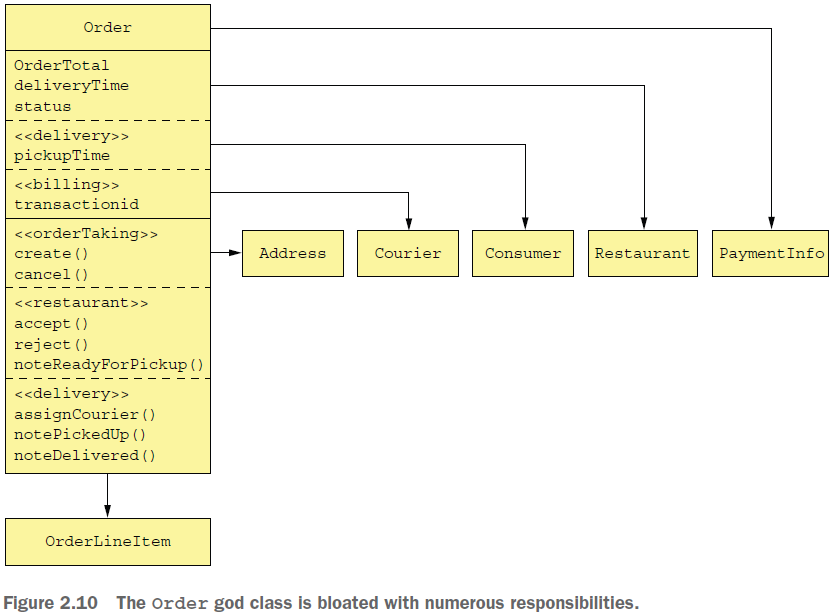
### Obtaining a Consistent View of the Data

**Another obstacle to decomposition is the inability to obtain a truly consistent view of data across multiple databases.** In a monolithic application, the properties of ACID transactions guarantee that a query will return a consistent view of the database. In contrast, in a microservice architecture, even though each service’s database is consistent, you can’t obtain a globally consistent view of the data. If you need a consistent view of some data, then it must reside in a single service, which can prevent decomposition. Fortunately, in practice this is rarely a problem.

### God Classes Prevent Decomposition

Another obstacle to decomposition is the existence of so-called god classes. *God classes* are the bloated classes that are used throughout an application (http://wiki.c2.com/?GodClass). **A god class typically implements business logic for many different aspectsof the application.** It normally has a large number of fields mapped to a databasetable with many columns. Most applications have at least one of these classes, eachrepresenting a concept that’s central to the domain: **accounts in banking, orders ine-commerce, policies in insurance, and so on**. Because a god class bundles **togetherstate and behavior for many different aspects of an application, it’s an insurmountableobstacle to splitting any business logic that uses it into services.**

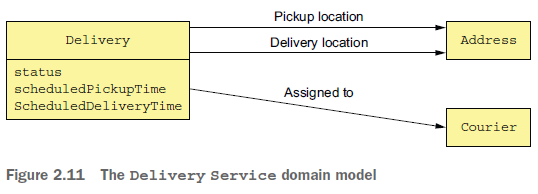
The Order class is a great example of a god class in the FTGO application. That’s not surprising—after all, the purpose of FTGO is to deliver food orders to customers. Most parts of the system involve orders. If the FTGO application had a single domain model, the Order class would be a very large class. It would have state and behavior corresponding to many different parts of the application. Figure 2.10 shows the structure of this class that would be created using traditional modeling techniques. As you can see, the Order class has fields and methods corresponding to order processing, restaurant order management, delivery, and payments. **This class also has a complex state model, due to the fact that one model has to describe state transitions from disparate parts of the application. In its current form, this class makes it extremely difficult to split code into services:**



* One solution is to package the Order class into a library and create a central Order database. All services that process orders use this library and access the access database. The trouble with this approach is that it violates one of the key principles of the microservice architecture and results in undesirable, tight coupling. For example, any change to the Order schema requires the teams to update their code in lockstep.
* Another solution is to encapsulate the Order database in an Order Service, which is invoked by the other services to retrieve and update orders. The problem with that design is that the Order Service would be a data service with an anemic domain model containing little or no business logic.

Neither of these options is appealing, but fortunately, **DDD provides a solution:**

* **A much better approach is to apply DDD and treat each service as a separate subdomain with its own domain model.** This means that **each of the services** in the FTGO application that has anything to do with orders **has its own domain model** **with its version of the Order class**. A great example of the benefit of multiple domain models is the Delivery Service. Its view of an Order, shown in figure 2.11, is extremely simple: pickup address, pickup time, delivery address, and delivery time. Moreover, rather than call it an Order, the Delivery Service uses the more appropriate name of Delivery:

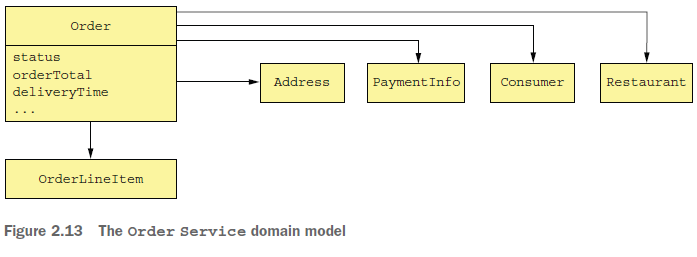


**The Delivery Service isn’t interested in any of the other attributes of an order.**

* The Kitchen Service also has a much simpler view of an order. **Its version of an Order is called a Ticket**. As figure 2.12 shows, a Ticket simply consist of a status, the requestedDeliveryTime, a prepareByTime, and a list of line items that tell the restaurant what to prepare. It’s unconcerned with the consumer, payment, delivery, and so on:



* The Order service has the most complex view of an order, shown in figure 2.13. Even though it has quite a few fields and methods, it’s still much simpler than the original version:



**The Order class in each domain model represents different aspects of the same Order business entity**. **The FTGO application must maintain consistency between these different objects in different services**. For example, once the Order Service has authorized the consumer’s credit card, it must trigger the creation of the Ticket in the Kitchen Service. Similarly, if the restaurant rejects the order via the Kitchen Service, it must be cancelled in the Order Service, and the customer credited in the billing service.

In chapter 4, you’ll learn how to maintain consistency between services, using the previously mentioned event-driven mechanism sagas.

As well as creating technical challenges, having multiple domain models also impacts the implementation of the user experience. **An application must translate between the user experience, which is its own domain model, and the domain models of each of the services.** In the FTGO application, for example, the Order status displayed to a consumer is derived from Order information stored in multiple services. **This translation is often handled by the API gateway, discussed in chapter 8.**

**Despite these challenges, it’s essential that you identify and eliminate god classes when defining a microservice architecture.**

We’ll now look at how to define the service APIs.

# Defining Service APIs

So far, we have a list of system operations and a list of a potential services. The next step is to define each **service’s API: its operations** **and events**. A service API operation exists for one of two reasons: some operations correspond to system operations. They are invoked by external clients and perhaps by other services. The other operations exist to support collaboration between services. These operations are only invoked by other services.

A service publishes events primarily to enable it to collaborate with other services. Chapter 4 describes how **events can be used to implement sagas, which maintain data consistency** across services. And chapter 7 discusses how **events can be used to update CQRS views, which support efficient querying.** An application **can also use events to notify external clients**. For example, it could use WebSockets to deliver events to a browser.

**The starting point for defining the service APIs is to map each system operation to a service**. **After that, we decide whether a service needs to collaborate with others** to implement a system operation**. If collaboration is required, we then determine what APIs those other services must provide in order to support the collaboration**. Let’s begin by looking at how to assign system operations to services.

## Assigning System Operations to Services

**The first step is to decide which service is the initial entry point for a request.** Many system operations neatly map to a service, but sometimes the mapping is less obvious.

Consider, for example, the noteUpdatedLocation() operation, which updates the courier location. On one hand, because it’s related to couriers, this operation should be assigned to the Courier service. On the other hand, it’s the Delivery Service that needs the courier location. In this case, assigning an operation to a service that needs the information provided by the operation is a better choice. In other situations, it might make sense to assign an operation to the service that has the information necessary to handle it.

An example of mapping between services and system operations:

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

After having assigned operations to services, the next step is to decide how the services collaborate in order to handle each system operation.

## Defining the APIs Required to Support Collaboration between Services

Some system operations are handled entirely by a single service. For example, in the FTGO application, the Consumer Service handles the createConsumer() operation entirely by itself. But other system operations span multiple services. The data needed to handle one of these requests might, for instance, be scattered around multiple services. For example, in order to implement the createOrder() operation, the Order Service must invoke the following services in order to verify its preconditions and make the post-conditions become true:

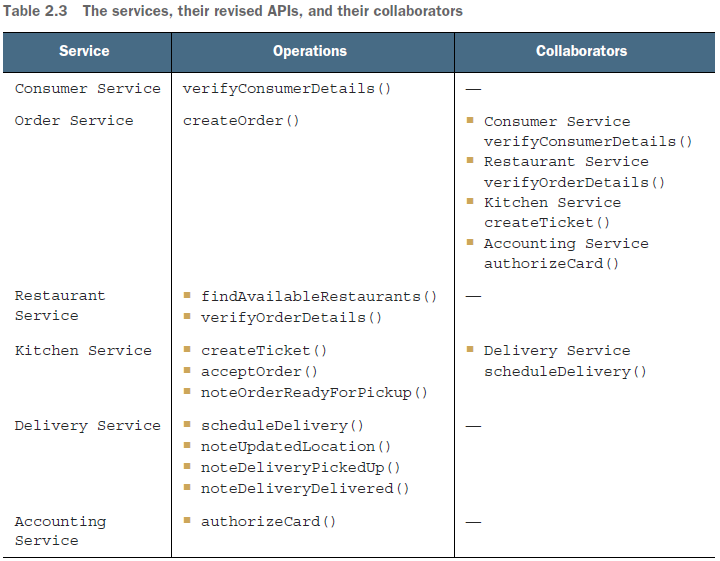
 Consumer Service—Verify that the consumer can place an order and obtain their payment information.

 Restaurant Service—Validate the order line items, verify that the delivery address/time is within the restaurant’s service area, verify order minimum is met, and obtain prices for the order line items.

 Kitchen Service—Create the Ticket.

 Accounting Service—Authorize the consumer’s credit card.

Similarly, in order to implement the acceptOrder() system operation, the Kitchen Service must invoke the Delivery Service to schedule a courier to deliver the order. The table below shows the services, their revised APIs, and their collaborators. **In order to fully define the service APIs, you need to analyze each system operation and determine what collaboration is required.**



So far, we’ve identified the services and the operations that each service implements. But it’s important to remember that the architecture we’ve sketched out is very abstract. We’ve not selected any specific IPC technology. Moreover, even though the term *operation* suggests some kind of synchronous request/response-based IPC mechanism, you’ll see that asynchronous messaging plays a significant role. Throughout the book he describes architecture and design concepts that influence how these services collaborate.

Chapter 3 describes specific IPC technologies, including synchronous communication mechanisms such as REST, and asynchronous messaging using a message broker. **I discuss how synchronous communication can impact availability** and **introduce the concept of a self-contained service, which doesn’t invoke other services synchronously.**

**One way to implement a self-contained service is to use the CQRS pattern**, covered in chapter 7. The Order Service could, for example, maintain a replica of the data owned by the Restaurant Service in order to eliminate the need for it to synchronously invoke the Restaurant Service to validate an order. It keeps the replica up-to-date by subscribing to events published by the Restaurant Service whenever it updates its data.

Chapter 4 introduces the saga concept and how it uses asynchronous messaging for coordinating the services that participate in the saga. As well as reliably updating data scattered across multiple services, **a saga is also a way to implement a self-contained service**. For example, I describe how the createOrder() operation is implemented using a saga, which invokes services such as the Consumer Service, Kitchen Service, and Accounting Service using asynchronous messaging.

Chapter 8 describes the concept of an API gateway, which exposes an API to external clients. An API gateway might implement a query operation using the API composition pattern, described in chapter 7, rather than simply route it to the service. Logic in the API gateway gathers the data needed by the query by calling multiple services and combining the results. In this situation, the system operation is assigned to the API gateway rather than a service. The services need to implement the query operations needed by the API gateway.

# Summary

 Architecture determines your application’s *-ilities*, including **maintainability, testability, and deployability**, which directly impact **development velocity.**

 The microservice architecture is an architecture style that **gives an application high** **maintainability**, **testability**, and **deployability.**

 You can eliminate god classes, which cause tangled dependencies that prevent decomposition, by applying DDD and **defining a separate domain model foreach service.**