**Decomposition Strategies**

* **What is microservice architecture exactly?**

The architecture of software application is its high-level structure, which consists of constituent parts and the dependencies between those parts.

As you will see in this section, an application’s architecture is multidimensional, so there are multiple ways to describe it. The reason architecture is important is because it determines the application’s software quality attributes or -ilities.

Traditionally, the goal of architecture has been scalability, reliability, and security. But today it is important that the architecture also enables the rapid and safe delivery of software. You will learn that microservice architecture is an architecture style that gives an application high maintainability, testability and deployability.

* **What is software architecture and why does it matter?**

Architecture is clearly important. There are at least two conferences dedicated to the topic.

O’Reilly software architecture conference <https://conferences.oreilly.com/software-architecture>

And the SATURN conference

<https://resources.sei.cmu.edu/news-events/events/saturn/>

Many developers have a goal of becoming architect. But what is architecture and why does it matter?

**A DEFINITION OF SOFTWARE ARCHITECTURE:**

There are numerous definitions about software architecture. But favourite definition comes from Len Bass and colleagues at

Software Engineering Institute. [www.sei.cmu.edu](http://www.sei.cmu.edu)

*The software architecture of a computing system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.*

That’s obviously a quiet abstract definition. But its essence is that an application’s architecture is its decomposition into parts

(the elements) and the relationship (the relations) between those parts. Decomposition is important for couple of reasons:

* + It facilitates the division of labour and knowledge. It enables multiple people (or multiple teams) with possibly specialized knowledge to work productively together an application.
  + It defines how the software elements interact.

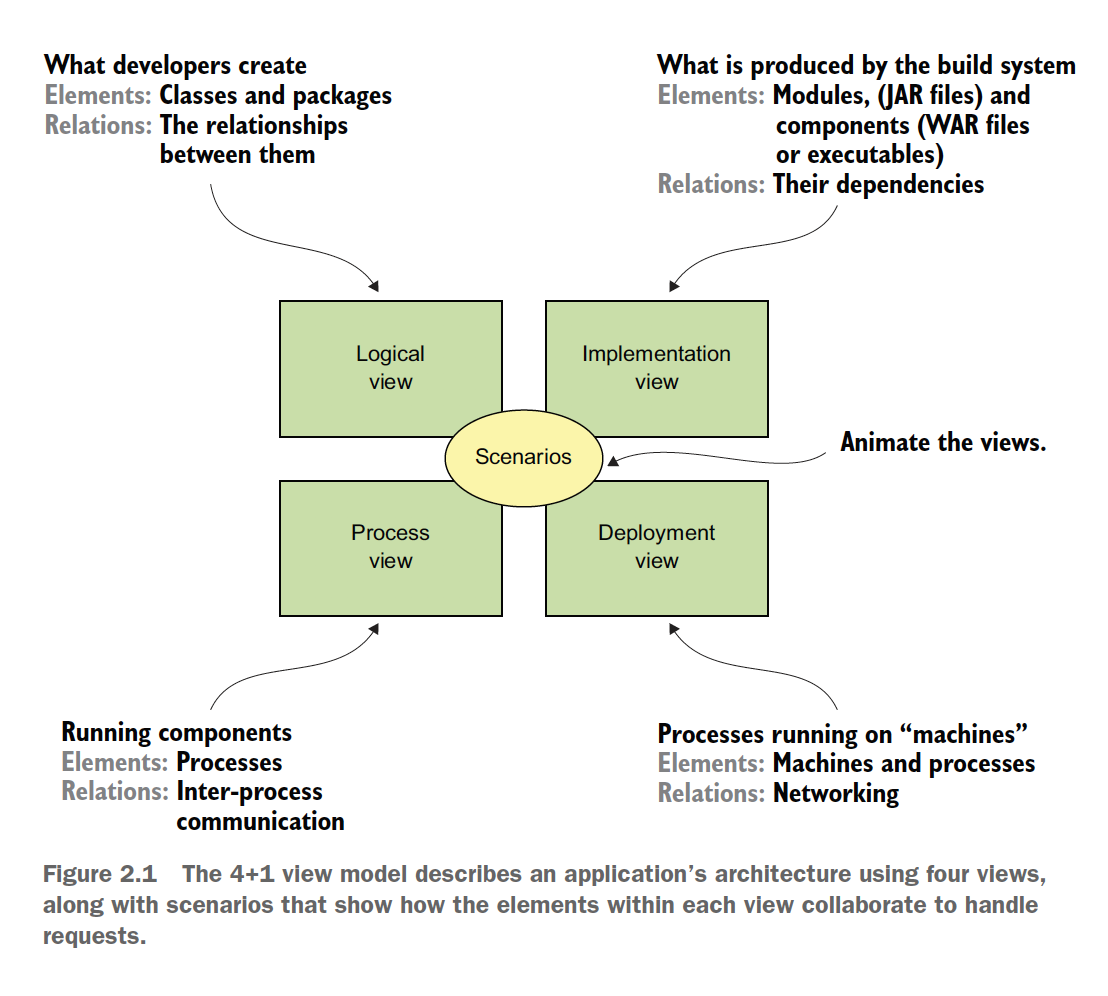
**THE 4+1 VIEW MODEL OF SOFTWARE ARCHITECTURE:**

More concretely, an application’s architecture can be viewed from multiple perspective, in the same way that a building’s architecture can be viewed from structural, plumbing, electrical and other perspectives.

Phillip Krutchen wrote a classic paper describing the 4+1 view model of software architecture.

[www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf](http://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf)

The 4+1 model defines four different views of a software architecture. Each describes the particular aspect of the architecture and consists of a particular set of software elements and relationship among them.



The purpose of each view is as follows:

***Logical view*** – The software elements that are created by developers. In object oriented languages, these elements are classes and packages. The relations between them are the relationship between classes and packages, including inheritance, associations, and depends-on.

***Implementation view*** – The output of the build system. This view consists of modules, which represent packaged code, and components, which are executable or deployable units consisting of one or more modules. In Java, a module is a JAR file, and a component is typically a WAR file or executable JAR file. The relations between them include dependency relationships between modules and composition relationships between components and modules.

***Process view*** – The components at runtime. Each element is a process, and the relations between processes represent inter-process communication.

***Deployment view*** – How the processes are mapped to the machines and the processes. The relations between machines represent networking. This view also describes the relationship between processes and machines.

In addition to these four views, there are the scenarios – the +1 in the 4+1 model that animate views. Each scenario describes how the various architectural components within a particular view collaborate in order to handle a request.

A scenario in the logical view, for example shows how the classes collaborate. Similarly, a scenario in the process view shows how the processes collaborate.

“***The 4+1 view model is an excellent way to describe an applications architecture”***

**WHY ARCHITECTURE MATTERS:**

An application has two categories of requirements.

The first category includes the functional requirements, which define what the application must do. They are usually in the form or use cases or user stories. Architecture has very little to do with the functional requirements. You can implement functional requirements with almost any architecture, even a big ball of mud.

Architecture is important because it enables an application to satisfy the second category of requirements: it’s ***quality of service*** requirements. These are also known as quality attributes and are the so called -ilities. They also define development time qualities including maintainability, testability, and deployability. The architecture you choose for your application determines how well it meets these quality requirements.

* **Overview of architectural styles:**

In the physical world, a building’s architecture often follows a particular style, such as Victorian, American Craftsman, or Art Deco. Each style is a package of design decisions that constrains a building’s feature and building materials. The concept of architectural style applies to software.

David Garlan and Mary Shaw (An Introduction to Software Architecture) pioneers in the discipline of software architecture, define an architectural style as follows.

*An architectural style, then, defines a family of such systems in terms of patterns of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined.*

A particular architectural style provides a limited palette of elements (components) and relations (connectors) from which you can define a view of your application’s architecture. An application typically uses a combination of architectural styles. For example, later in this section I describe how the monolithic architecture is an architectural style that structures the implementation view as a single (executable/deployable) component. The microservice architecture structures an application as a set of loosely coupled services.

**THE LAYERED ARCHITECTURAL STYLE:**

The classic example of an architectural style is the layered architecture:

A layered architecture organizes software elements into layers.

Each layers has a well-defined set of responsibilities.

A layered architecture also constraints the dependencies between the layers.

A layer can only depend on either the layer immediately below it (if strict layering) or any of the layers below it.

You can apply the layered architecture to any of the four views discussed earlier. The popular three tier architecture is the layered architecture applied to logical view. It organizes the application classes into the following tiers or layers:

Presentation layer – Contains code that implements the user interface or external API’s.

Business logic layer – Contains the business logic.

Persistence layer – Implements the logic of interacting with the database.

The layered architecture is a great example of an architectural style, but it does have some significant drawbacks.

Single presentation layer – It doesn’t represent the fact that an application is likely to be invoked by more than just a single system.

Single persistence layer – It doesn’t represent the fact that an application is likely to interact with more than just a single database.

Defines the business logic layer as depending on the persistence layer:

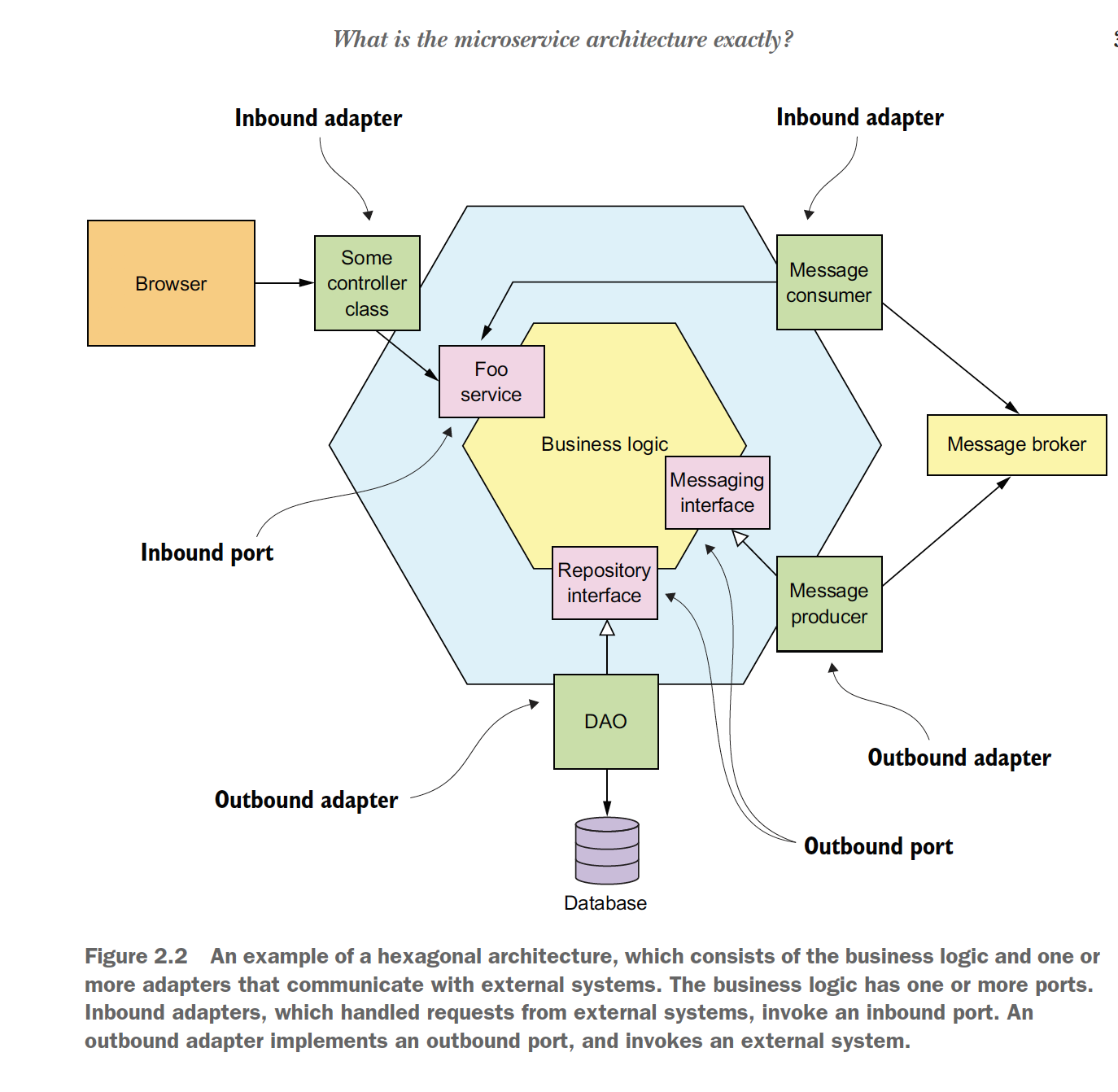
In theory, this dependency prevents you from testing the business logic without the database.

Also, the layered architecture mispresents the dependencies in a well-designed applications. The business logic typically defines an interface or a repository of interfaces that define data access methods. The persistence tier defines DAO classes that implement the repository interfaces. In other words, the dependencies are the reverse of what’s depicted by a layered architecture.

Let’s look at an alternative architecture that overcomes these drawbacks: the hexagonal architecture.

**ABOUT THE HEXAGONAL ARCHITECTURE STYLE:**

Hexagonal architecture is an alternative to the layered architectural style.



The hexagonal architecture style organizes the logical view in a way that places the business logic at the center.

Instead of the presentation layer, the application has one or more inbound adapters that handle requests from the outside by invoking the business logic.

Similarly, instead of data persistence tier, the application has one or more outbound adapters that are invoked by business logic and invoke external applications.

A key characteristic and benefit of this architecture is that the business logic doesn’t depend on the adapters. Instead, they depend on it.

The business logic has one or more ports. A port defines a set of operations and is how the business logic interacts with what’s the outside of it. In Java, for example, a port is often a Java interface. There are two kinds of ports: inbound and outbound ports.

An inbound port is an API exposed by the business logic, which enables which enables it to be invoked by external applications.

An example of an inbound port is a service interface, which defines a service’s public methods.

An outbound port is how the business logic invokes external systems. An example of outbound port is a repository interface, which defines a collection of data access operations.

Surrounding the business logic are adapters. As with ports, there are two types of adapters: inbound and outbound. An inbound adapter handles requests from the outside world by invoking an inbound port.

An example of an inbound adapter is a spring MVC Controller that implements either a set of rest endpoints or a set of web pages.

Another example is a message broker client that subscribes to messages. Multiple inbound adapters can invoke the same inbound port.

An outbound adapter implements an outbound port and handles the request from the business logic by invoking an external application or service. An example of outbound adapter is a data access object class that implements operation for accessing database.

Another example would be a proxy class that invokes a remote service. Outbound adapters can also publish events.

An important benefit of the hexagonal architectural style is that it decouples the business logic from the presentation and data access logic in adapters. The business logic doesn’t depend on either the presentation logic or the data access logic.

Because of this decoupling, it’s much easier to test the business logic in isolation. Another benefit is that it more accurately reflects the architecture of modern application. The business logic can be invoked via multiple adapters, each of which implements a particular API or UI.

The business logic can also invoke multiple adapters, each one of which invokes a different external system. Hexagonal architecture is a great way to describe the architecture of each service in a microservice architecture.

* **The microservice architecture is an architectural style:**

I’ve discussed the 4+1 view model and architecture styles, so I can now define monolithic and microservice architecture. They are both architectural styles. Monolithic architecture is an architectural style that structures the implementation view as a single component: a single executable or WAR file. This definition says nothing about the other views. A monolithic application can, for example, have a logical view that’s organized along the lines of hexagonal architecture.

http://microservices.io/patterns/ monolithic.html

The microservice architecture is also an architectural style. It structures the implementation view as a set of components: executables or WAR files.

The components are services, and the connectors are communication protocols that enable those services to collaborate. Each service has its own logical view architecture, which is typically a hexagonal architecture. Below figure shows a possible architecture for the FTGO application. The services in this architecture correspond to business capabilities such as Order management and Restaurant management.

<http://microservices.io/patterns/microservices.html>

Later in this chapter we will see what is ***business capability***. The connectors between these services are implemented using inter-process communication mechanisms such as REST and asynchronous messaging.

Diagram

Description automatically generated

A key constraint imposed by the microservice architecture is that the services are loosely coupled. Consequently, there are restrictions on how the services collaborate. In order to explain those restrictions. I will attempt to define the term ***service*** describe what it means to be loosely coupled, and tell you why this matters.

**WHAT IS A SERVICE?**

A service is a standalone, independently deployable software component that implements some useful functionality. Below figure shows an external view of service, which in this example is the order service.

Diagram

Description automatically generated

A service has an API that provides its clients access to its functionality. There are two kinds of operation: ***commands*** and ***queries.*** The API consists of commands, queries, and events.

A command such as createOrder() performs actions and updates data.

A query, such as findOrderById(), retrieves data.

A service also publishes events, such as OrderCreated, which are consumed by its clients.

A service’s API encapsulates it internal implementation. Unlike in monolith, a developer can’t write code that bypasses its API. As a result, the microservice architecture enforces the application’s modularity.

Each service in a microservice architecture has its own architecture and, potentially technology stack. But a typical service has a hexagonal architecture. Its API is implemented by adapters that interact with the service’s business logic. The operations adapter invokes the business logic, and the events adapter publishes event emitted by the business logic.

**WHAT IS LOOSE COUPLING?**

An important characteristics of the microservice architecture is that the services are loosely coupled. All interactions with a service happens via its API, which encapsulates its implementation details. This enables the implementation of the service to change without impacting its clients.

Loosely coupled services are key to improving an application’s development time attributes, including its maintainability and testability. They are much easier to understand, change and test.

**THE ROLE OF SHARED LIBRARIES:**

Developers often package functionality in a library (module) so that it can be reused by multiple applications without duplicating code. After all, where would we be today without Maven or npm repositories? You might be tempted to also use shared libraries in microservice architecture. On the surface, it looks like a good way to reduce code duplication in your services. But you need to ensure that you don’t accidently introduce coupling between your services.

Imagine, for example, that multiple services need to update the order business object. One approach is to package that functionality as a library that’s used by multiple services. On one hand using a library eliminates code duplication. On the other hand, consider what happens when the requirements change in a way that affects the order business object. You would need to simultaneously rebuild and redeploy those services. A much better approach would be to implement functionality that’s likely to change, such as Order management, as a service.

You should strive to use libraries for functionality that’s unlikely to change. For example, in a typical application it makes no sense for every service to implement a generic Money class. Instead, you should create a library that’s used by the services.

**THE SIZE OF A SERVICE IS MOSTLY UNIMPORTANT:**

One problem with the term microservice is that the first thing you hear is micro. This suggests that a service should be very small. This is also true of other size-based terms such as miniservice and nano-service. In reality, size isn’t a useful metric.

A much better goal is to define a well-designed service to be a service capable of being developed by a small team with minimal lead time and with minimal collaborations with other teams.

In theory, a team might only be responsible for a single service, so that service is no means micro. Conversely, if a service requires a large team or takes long time to test, it probably makes sense to split the team and the service.

Or if you constantly need to change a service because of changes to other services or if it’s triggering changes in other services, that’s a sign that it is not loosely coupled. You might even have built a distributed monolith.

* **Defining an application’s microservice architecture:**

How should we define a microservice architecture? As with any software development effort, the starting points are the written requirements, hopefully domain experts, and perhaps an existing application. Like much of software development, defining an architecture is more art than science. This section describes a simple, three step process, shown in below figure.

Diagram

Description automatically generated

An application exists to handle requests, so the first step is defining its architecture is to distil the application’s requirements into a key requests. But instead of describing the requests in terms of specific IPC technologies such as REST or messaging. I use the more abstract notion of system operation.

A ***system operation*** is 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 behaviour 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.

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 sub-domains. The end result is services that are organized around business concepts rather than technical concepts.

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 will also need to decide which of the IPC mechanism to implement each services API.

There are several obstacles to decomposition:

The first is network latency. You might discover that a particular decomposition would be impractical due to too many round-trips between services.

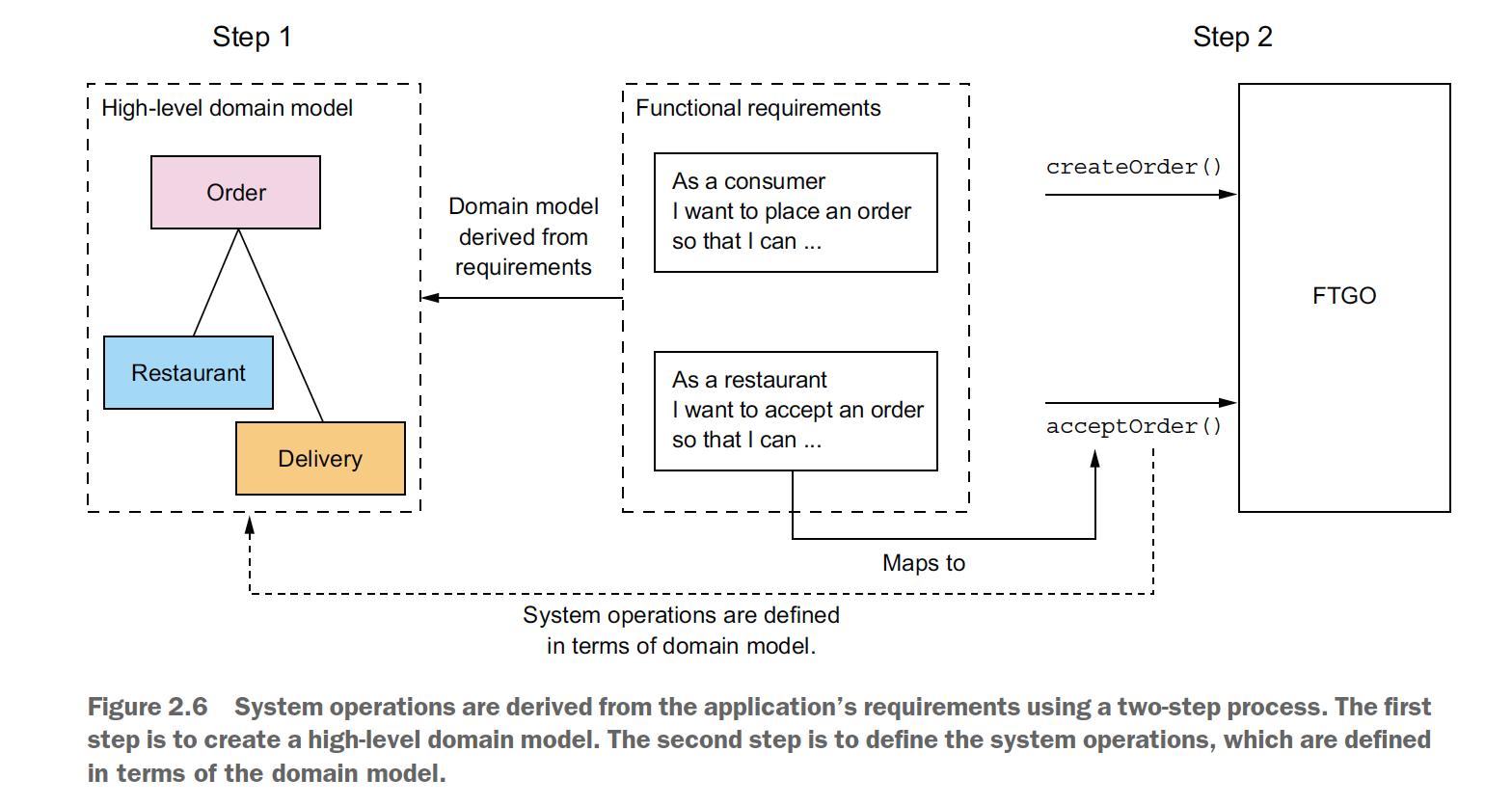
Another obstacle to decomposition is that synchronous communication between services reduces availability. We might need to use the concept of self-contained services.

The third obstacle is the requirement to maintain data consistency across services. You’ll typically need to use sagas.

The fourth and final obstacle to decomposition is so called god classes which are used throughout an application. Fortunately, we can use concepts from domain-driven design to eliminate god classes.

* **Identifying the 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 show in figure below.



This process is inspired by the object-oriented design process covered in Craig Larman’s book Applying UML and patterns. See below link for details.

[www.craiglarman.com/wiki/index.php?title=Book\_Applying\_UML\_and\_Patterns](http://www.craiglarman.com/wiki/index.php?title=Book_Applying_UML_and_Patterns)

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 behaviour in terms of 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 we will cover in chapter-5.

The behaviour 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:**

The first step in the process of defining the system operations is to sketch a high level domain model for the application. 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, 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 behaviour of the system operations.

A domain model is created using standard techniques such as analyzing the nouns in the stories and scenarios and talking to the domain experts. 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 Credit Card.

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***

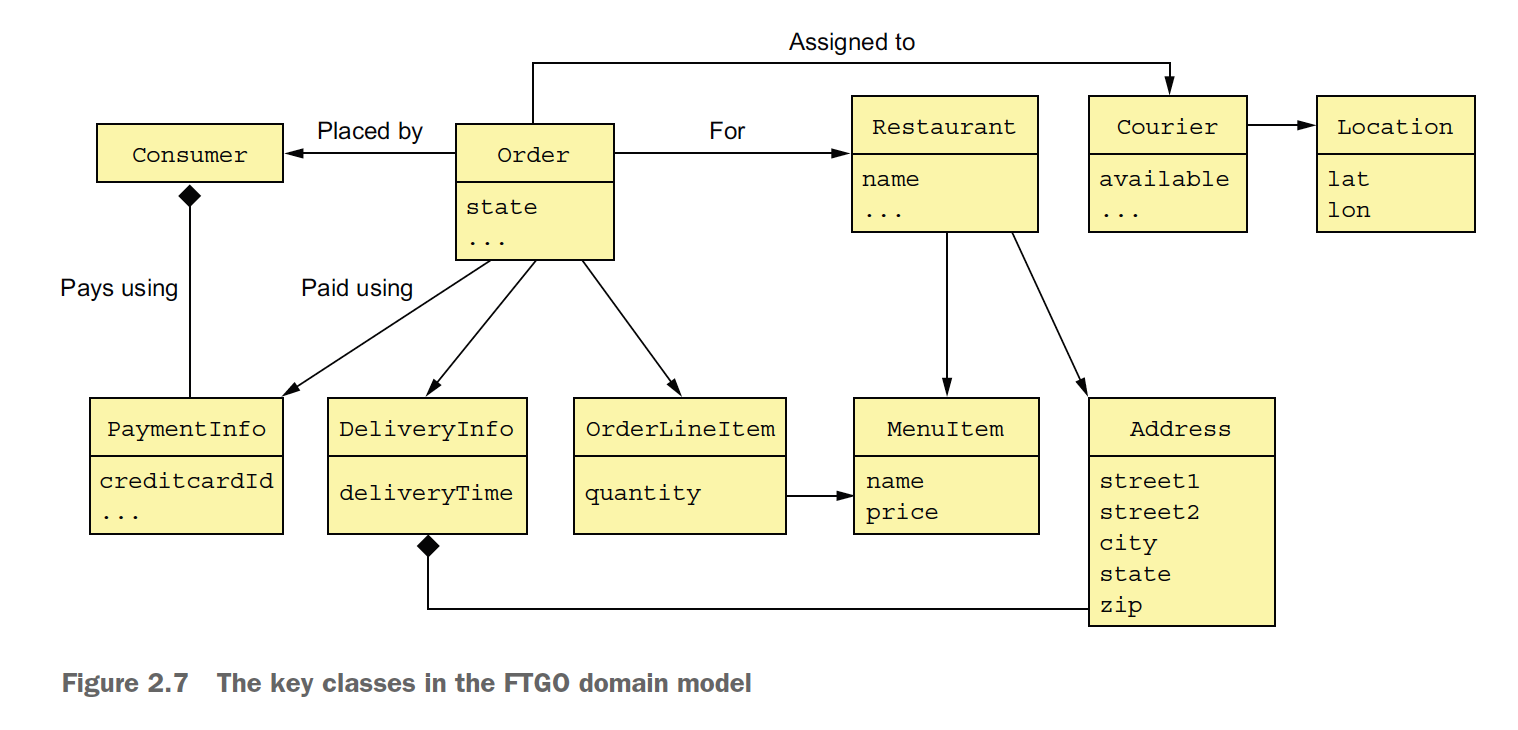
***a particular time***

***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 the few iterations of analysis will be a domain model that consists, unsurprisingly, of those classes and others, such as MenuItem and Address. Below class diagram show the key classes.



The responsibilities of each class are as follows.

Consumer – A consumer who places an order.

Order – An order placed by an 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 delivers 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.

**DEFINING SYSTEM OPERATIONS:**

Once you have defined a high-level domain model, the next step is to identify the requests that the application must handle. The details of the UI are beyond the scope of the book, but you can imagine that in each user scenario, the UI will make requests to the backend business logic to retrieve and update data. FTGO is primarily a web application, which means that most requests are HTTP-based, but it’s possible that some clients might use messaging. Instead of committing to a specific protocol, therefore, it makes sense to use the more abstract notation of a system operation to represent requests.

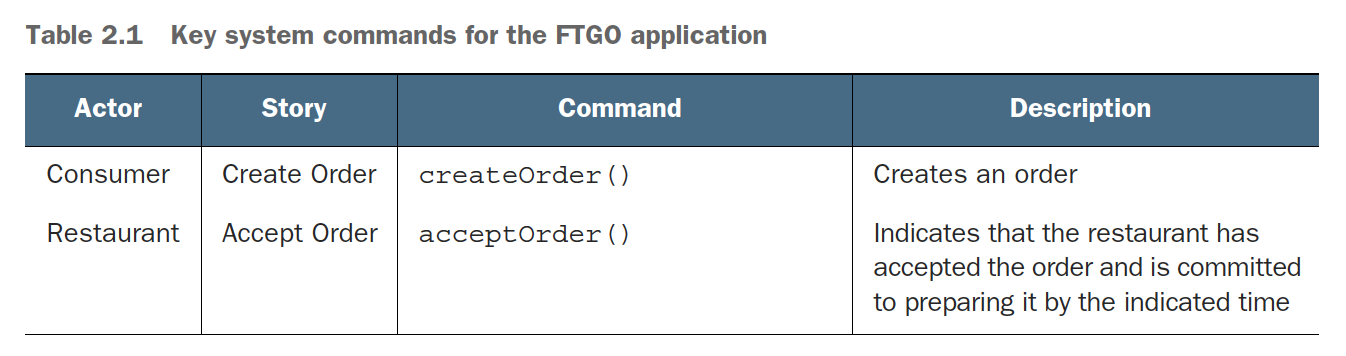
There are two types of system operations:

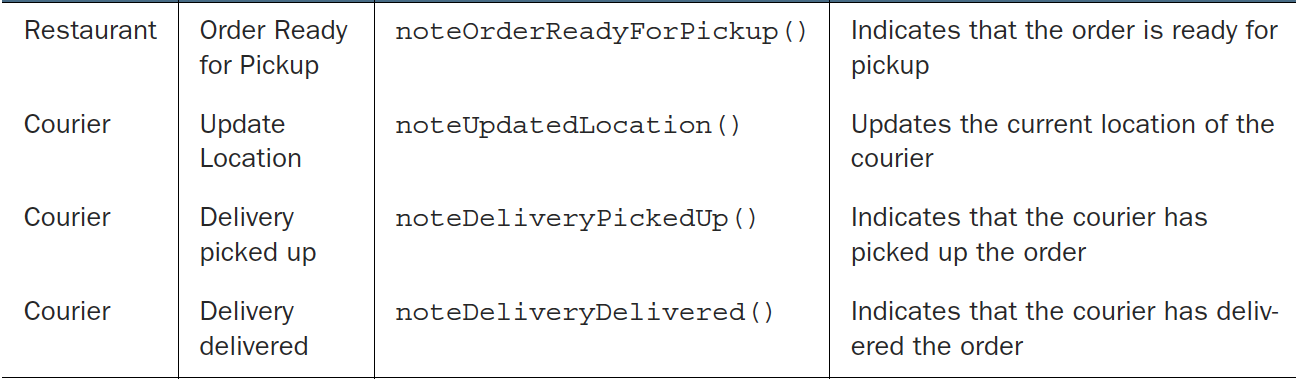
Commands – System operations that create, update, and delete data.

Queries – System operations that read (query) data.

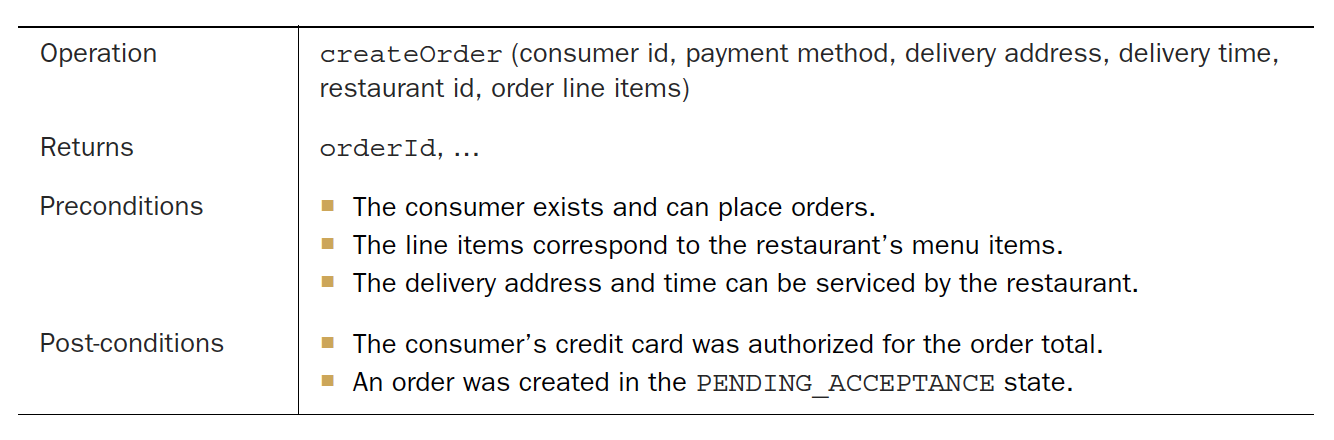
Ultimately, these system operations will corresponds to REST, RPC or messaging endpoints, but for now thinking of them abstractly is useful. Let’s first identify some commands.

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. Below table shows key commands.



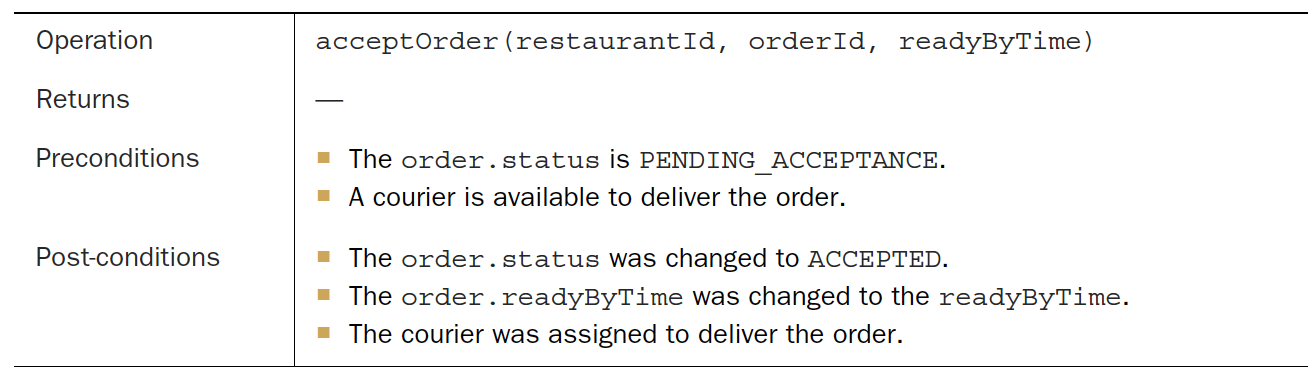


A command has a specification that defines its parameters, return values, and behaviour in terms of the domain model classes. The behaviour specification consists of preconditions that must be true when the operation is invoked, and the post-conditions that are true after the operation is invoked. Here, for example, is the specification of the createOrder() system operation.



The pre-conditions mirrors the *givens* in the place order user scenario described earlier. The post-conditions mirrors the *then* from the scenario. When the system operation is invoked it will verify the preconditions and perform the actions required to make the postconditions true.

Here is the specification of the acceptOrder() system operation.



It’s pre and post conditions mirror the user scenario from earlier.

Most of the architecturally relevant systems operations are commands. Sometimes, though, queries, which retrieves data, are also important.

Besides, implementing commands, an application must also implement queries. The queries provide the UI with the information a user needs to make a 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 an 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 the 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 application does. They help drive the definition of the application’s architecture. The behaviour of each system operation is described in terms of the domain model. Each important system operation represents an architecturally significant scenario that’s the 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 various decomposition strategies we 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.

* **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 modelling, a business capability is something that a business does in order to generate value. 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.

<http://microservices.io/patterns/decomposition/decompose-by-business-capability.html>

**BUSINESS CAPABILITIES DEFINE WHAT AN ORGANIZATION DOES:**

An organization’s business capabilities capture what an organization’s business is. They are generally stable, as opposed to how an organization conducts its business, which changes over time, sometimes drastically. 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 become possible to deposit checks using 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. It’s specification consists of various components, including inputs, outputs, and service-level agreements.

For example, the input to an Insurance underwriting capability is the consumer’s application, and the outputs include approval and price.

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 decomposed into sub-capabilities. For example, the Claim management capability has several sub-capabilities, including Claim information management, Claim review, and Claim payment management.

It is not difficult to imagine that 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 consumer.

Order taking and fulfilment:

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 courier.

The top-level capabilities include supplier management, consumer management, order taking and fulfilment, 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 fulfilment is decomposed into five sub-capabilities.

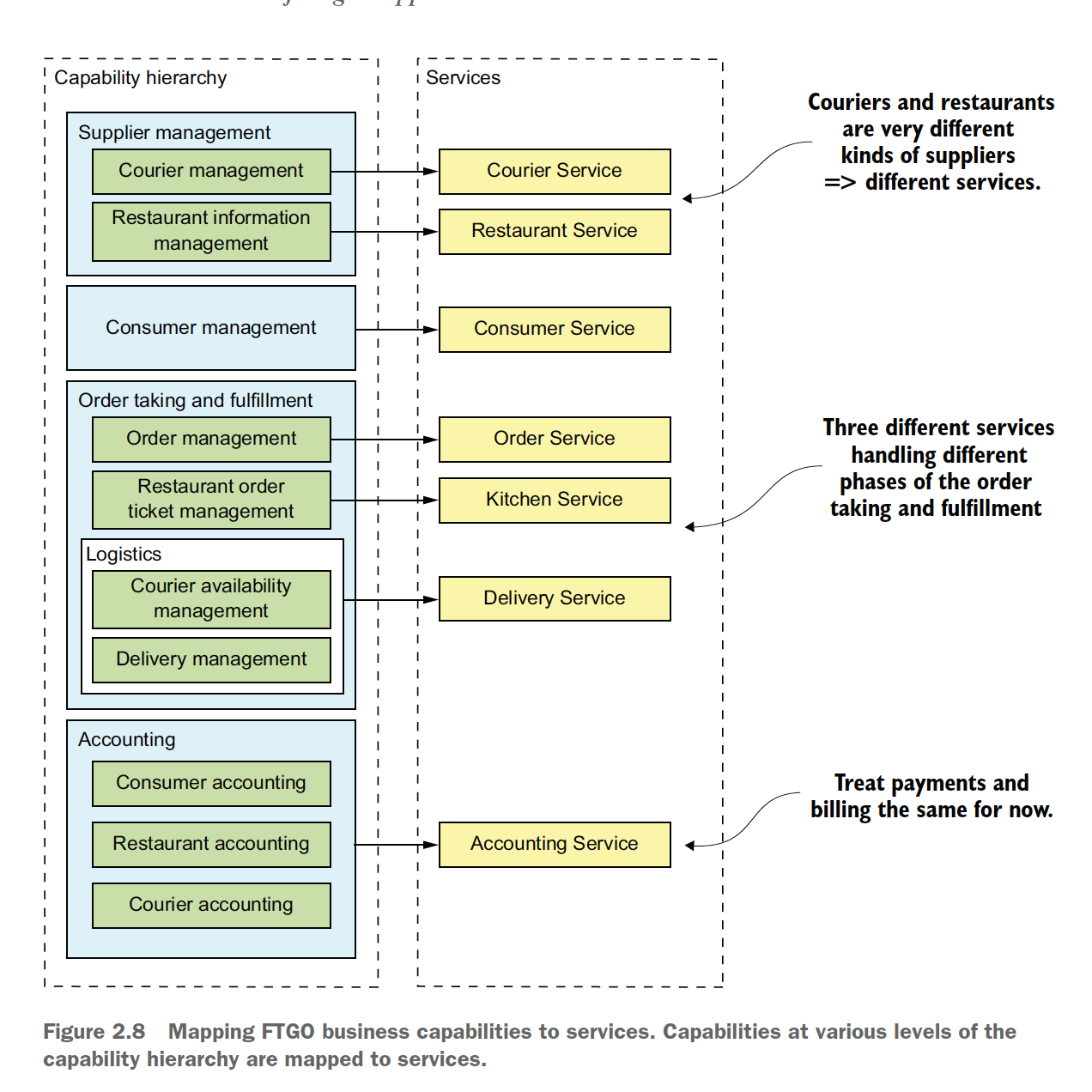
On interesting aspect of this capability hierarchy is that there are three restaurant related capabilities. Restaurant information management, Restaurant order management, and Restaurant accounting. That’s because they represent three very different aspects of restaurant operations.

**FROM BUSINESS CAPABILITIES TO SERVICES:**

Once you have defined the business capabilities, you then define a service for each capability or group of related capabilities. Below figure 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, because 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 order taking and fulfilment capabilities 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 are 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 are 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 above 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.

* **Defining Services by applying the Decompose by sub-domain pattern:**

DDD, as described in the excellent book Domain-driven design by Eric Evans, is an approach for building complex software applications that is centered on the development of an object-oriented domain model.

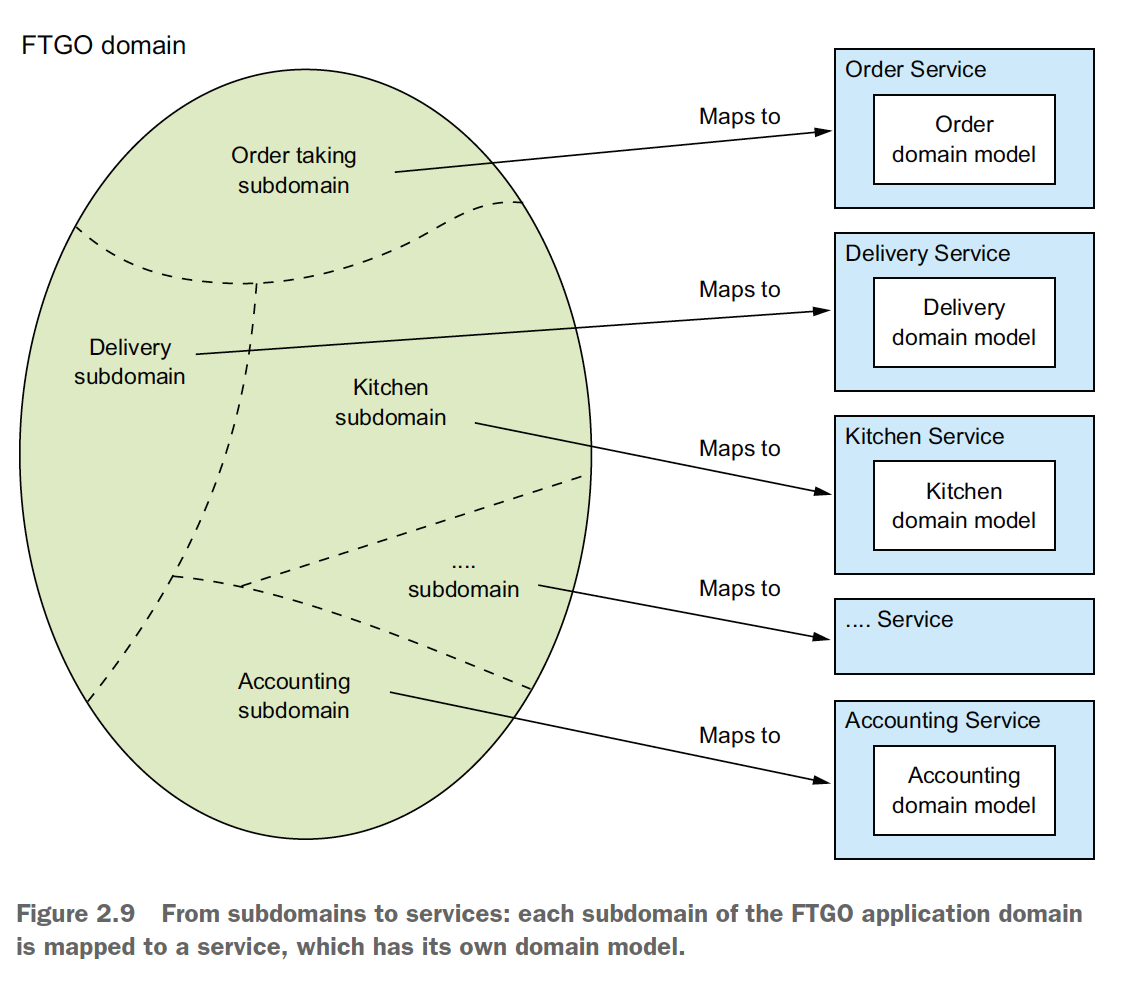
A ***domain model*** captures knowledge about a domain in a form that can be used to solve problems within in that domain. It defines the vocabulary used by the team, what DDD calls the ***Ubiquitous Language***. The domain model is closely mirrored in the design and implementation of the application. DDD has two concepts that are incredibly useful when applying the microservice architecture: ***subdomains and bounded context.***

<http://microservices.io/patterns/decomposition/decompose-by-subdomain.html>

DDD is quite different than the traditional approach to enterprise modelling, which creates a single model for the entire enterprise. In such a model there would be, for example, a single definition of each business entity, such as customer, order and so on. The problem with this kind of modelling is that getting different parts of an organization to agree on a single model is a monumental task. Also, it means that from the perspective of a given part of the organization, the model is overly complex for their needs. Moreover, the domain model can be confusing because different parts of the organization might use either the same term for different concepts or different terms for the same concept. DDD avoids these problems by defining multiple domain models, each with an explicit scope.

DDD defines a separate domain model for each sub-domain. A sub-domain is a part of the domain, DDD’s term for the application’s problem space. Subdomains are identified using the same approach as identifying business capabilities: analyze the business and identify the different areas of expertise. The end result is very likely to be subdomains that are similar to the other business capabilities. The examples of subdomains in FTGO include Order taking, Order Management, Kitchen management, Delivery and Financials. As you can see, these sub-domains are very similar to the business capabilities described earlier.

DDD calls the scope of a domain model a ***bounded context***. A bounded context includes the code artifacts that implements the model. When using microservice architecture, each bounded context is a service or possibly a set of services. We can create microservice architecture by applying the DDD and defining service for each subdomain. Below figure shows how the subdomains map to services, each with its own domain model.



DDD and the microservice architecture are in almost perfect alignment. The DDD concept of subdomains and bounded context maps nicely to services within a microservice architecture. Also, the microservice architecture’s concept of autonomous team owing services is completely aligned with the DDD’s scope of each domain model being owned and developed by a single team. Even better, as I described later in this section, the concept of subdomain with its own domain model is a great way to eliminate god classes and thereby make the decomposition easier.

Decompose by subdomain and Decompose by business capability are the two main patterns for defining an application microservice architecture. There are, however, some useful guidelines for decomposition that have their roots in object-oriented design.

* **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.

The first principle is the Single Responsibility Principle (SRP), for defining the responsibilities of the class.

The second principle is the Common Closure Principle (CCP), for organizing classes into packages.

**SINGLE RESPONSIBILITY PRINCIPLE:**

One of the main goals of software architecture and design is determining the responsibility of each software element. The SRP is as follows.

*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 SRP, you define the 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 consumer – order taking, order preparation, and delivery – is the responsibility of a separate service.

**COMMON CLOSURE PRINCIPLE:**

*The classes in the 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 particular business rule. The goal is that when that business rule changes, developers only need to change code in 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 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.

SRP and CCP are 2 of the 11 principles developed by Bob Martin. They are particularly useful when developing a microservice architecture. The remaining nine principles are used when designing classes and packages.

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

* **Obstacles to decomposing an application into services:**

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

* + Network latency.
  + Reduced availability due to synchronous communication.
  + Maintaining data consistency across services.
  + Obtaining a consistent view of the data.
  + God classes preventing decomposition.