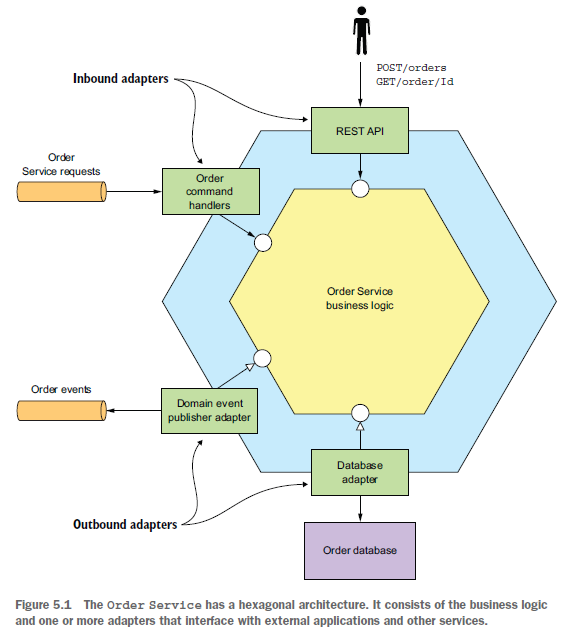
Business Logic Organization Patterns

Figure 5.1 shows the architecture of a typical service. As described in chapter 2, the business logic is the core of a hexagonal architecture. Surrounding the business logic are the inbound and outbound adapters. An *inbound adapter* handles requests from clients and invokes the business logic. An *outbound adapter*, which is invoked by the business logic, invokes other services and applications.



This service consists of the business logic and the following adapters:

 REST API adapter—An inbound adapter that implements a REST API which invokes the business logic

 OrderCommandHandlers—An inbound adapter that consumes command messages from a message channel and invokes the business logic

 Database Adapter—An outbound adapter that’s invoked by the business logic to access the database

 Domain Event Publishing Adapter—An outbound adapter that publishes events to a message broker

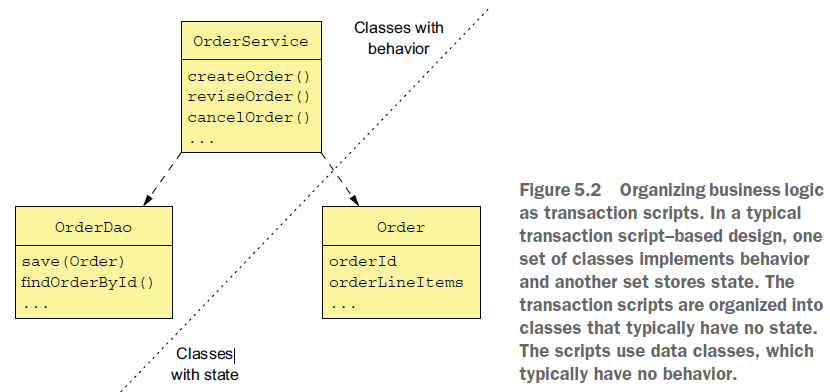
The business logic is typically the most complex part of the service. When developing business logic, you should consciously organize your business logic in the way that’s most appropriate for your application. Most enterprise applications are written in an object-oriented language such as Java, so they consist of classes and methods. But **using an object-oriented language doesn’t guarantee that the business logic has an object-oriented design.** The **key decision you must make when developing business logic is whether to use an object-oriented approach or a procedural approach**. There are two main patterns for organizing business logic: **the procedural Transaction script pattern**, and the **object-oriented Domain model pattern.**

# Designing Business Logic Using the Transaction Script Pattern

*“Organize the business logic as a* ***collection of******procedural*** *transaction scripts,* ***one for each type of request.”***

Although I’m a strong advocate of the **object-oriented** approach, there are **some situations** where it **is overkill**, such as when you are developing simple business logic. In such a situation, **a better** approach is **to write procedural code** and use what the book *Patterns* *of Enterprise Application Architecture* by Martin Fowler (Addison-Wesley Professional, 2002) calls the Transaction script pattern. Rather than doing any object-oriented design, **you write a method called a *transaction script* to handle each request from the presentation tier.**

**an important characteristic of this approach is that the classes that implement behavior are separate from those that store state:**



When using the Transaction script pattern, the scripts are usually located in service classes, which in this example is the OrderService class

* A service class has **one method for each request/system operation.** The method implements the business logic for that request.
* It accesses the database using data access objects (DAOs), such as the OrderDao. **The data objects,** which in this example is the Order class, **are pure data with little or no behavior.**

This style of design is highly procedural and relies on **few of the capabilities of object-oriented** programming (OOP) languages. This what you would create if you were writing the application in C or another non-OOP language.

Nevertheless, you shouldn’t be ashamed to use a procedural design when it’s appropriate. This approach works well for simple business logic. The drawback is that this tends not to be a good way to implement complex business logic.

# Designing Business Logic Using the Domain Model Pattern

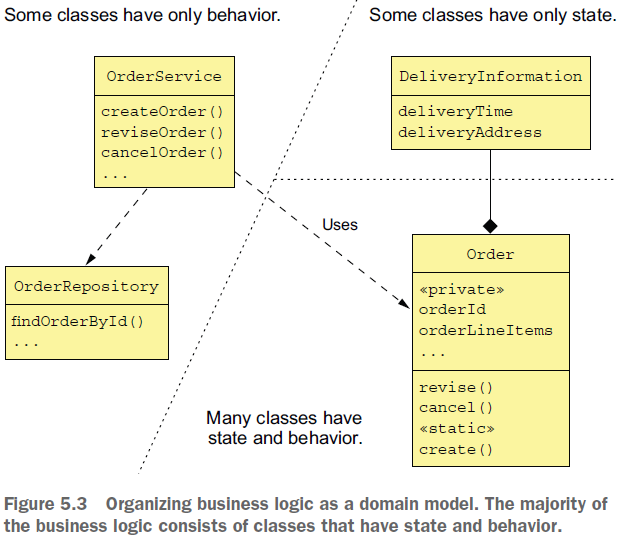
*“Organize the business logic as* ***an object model*** *consisting* ***of classes that have state***

***and behavior.”***

The problem with the procedural approach is that if your business logic becomes complex, you can end up with code that’s a nightmare to maintain.

In fact, in the same way that a monolithic application has a habit of continually growing, transaction scripts have the same problem. Consequently**, unless you’re writing an extremely simple application, you should resist the temptation to write procedural code and instead apply the Domain model pattern and develop an object-oriented design.**

In an object-oriented design, the business logic consists of **an object model**, **a network of relatively small classes**. **These classes typically correspond directly to concepts from the problem domain**. **In such a design some classes have only either state or behavior, but many contain both, which is the hallmark of a well-designed class.** Figure 5.3 shows an example of the Domain model pattern.



* As with the Transaction script pattern, an OrderService class has a method for each request/system operation.
* But **when using the Domain model pattern**, **the service methods are usually simple**. That’s because **a service method almost always** **delegates to persistent domain objects**, **which contain the bulk of the business logic:**

A service method might, for example, **load a domain object from the database** **and invoke one of its methods.** In this example, the Order class has both state and behavior. Moreover, its state is private and can only be accessed indirectly via its methods.

Using an object-oriented design has a number of benefits:

* **First**, the design is **easy to understand and maintain**. Instead of consisting of one big class that does everything, it consists of a number of small classes that each have a small number of responsibilities.

In addition, classes such as Account, BankingTransaction, and OverdraftPolicy **closely mirror the real world**, which makes their role in the design easier to understand.

* **Second**, our object-oriented design is **easier to test**: **each class can and should be tested independently**
* **Finally**, an object-oriented design is **easier to extend** because it can use well-known design patterns, such as **the Strategy pattern** and the **Template method pattern,** that **define ways of extending a component without modifying the code.**

The Domain model pattern works well, but there are a number of problems with this approach, especially in a microservice architecture. To address those problems, you need to **use a refinement of OOD known as DDD.**

# About Domain-Driven Design

DDD, which is described in the book *Domain-Driven Design* by Eric Evans (Addison-Wesley Professional, 2003), **is a refinement of OOD** and is **an approach for developing complex business logic.**

We talked DDD in the document about defining a microservice architecture when discussing the usefulness of DDD subdomains when decomposing an application into services. **When using DDD, each service has its own domain model, which avoids the problems of a single, application-wide domain model.** Subdomains and the associated concept of Bounded Context are two of the **strategic DDD patterns**.

**DDD also has some tactical patterns that are building blocks for domain models.**

**Each pattern is a role that a class plays in a domain model** and **defines the characteristics of the class.** The building blocks that have been widely adopted by developers include the following 5:

** *Entity***—**An object that has a persistent identity.** Two entities whose attributes have the same values are still different objects. In a Java EE application, classes that are persisted using JPA @Entity are usually DDD entities.

** *Value object***—An object that is a collection of values. Two value objects whose attributes have the same values can be used interchangeably. An example of a value object is a Money class, which consists of a currency and an amount.

** *Factory***—An object or method that implements object creation logic that’s too complex to be done directly by a constructor. It can also hide the concrete classes that are instantiated. **A factory might be implemented as a static method of a class.**

** *Repository***—An object that provides access to persistent entities and **encapsulates the mechanism for accessing the database.**

** *Service***—An object that implements **business logic that doesn’t belong in an entity or a value object.**

These building blocks are used by many developers. Some are supported by frameworks such as JPA and the Spring framework.

**There is one more building block that has been generally ignored** (myself included!) except by DDD purists: **aggregates.** **As it turns out, aggregates are an extremely useful concept when developing microservices.**

We have looked at some subtle problems with classic OOD that are solved by using aggregates in a separate document on DDD.