PART 2

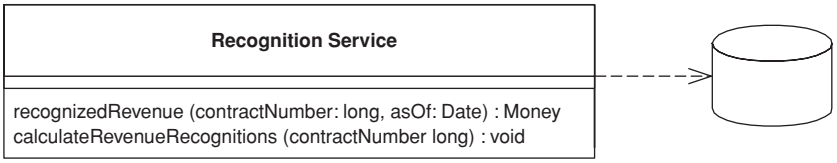
The Patterns

Chapter 9

Domain Logic Patterns

Transaction Script

Organizes business logic by procedures where each procedure handles a single request from the presentation.



Most business applications can be thought of as a series of transactions. A transaction may view some information as organized in a particular way, another will make changes to it. Each interaction between a client system and a server system contains a certain amount of logic. In some cases this can be as simple as displaying information in the database. In others it may involve many steps of validations and calculations.

A *Transaction Script* organizes all this logic primarily as a single procedure, making calls directly to the database or through a thin database wrapper. Each transaction will have its own *Transaction Script*, although common subtasks can be broken into subprocedures.

How It Works

With *Transaction Script* the domain logic is primarily organized by the transactions that you carry out with the system. If your need is to book a hotel room, the logic to check room availability, calculate rates, and update the database is found inside the Book Hotel Room procedure.

For simple cases there isn't much to say about how you organize this. Of course, as with any other program you should structure the code into modules in a way that makes sense. Unless the transaction is particularly complicated, that won't be much of a challenge. One of the benefits of this approach is that you don't need to worry about what other transactions are doing. Your task is to get the input, interrogate the database, munge, and save your results to the database.

Where you put the *Transaction Script* will depend on how you organize your layers. It may be in a server page, a CGI script, or a distributed session object. My preference is to separate *Transaction Scripts* as much as you can. At the very least put them in distinct subroutines; better still, put them in classes separate from those that handle presentation and data source. In addition, don't

have any calls from the *Transaction Scripts* to any presentation logic; that will make it easier to modify the code and test the *Transaction Scripts*.

You can organize your *Transaction Scripts* into classes in two ways. The most common is to have several *Transaction Scripts* in a single class, where each class defines a subject area of related *Transaction Scripts*. This is straightforward and the best bet for most cases. The other way is to have each *Transaction Script* in its own class (Figure 9.1), using the Command pattern [Gang of Four]. In this case you define a supertype for your commands that specifies some execute method in which *Transaction Script* logic fits. The advantage of this is that it allows you to manipulate instances of scripts as objects at runtime, although I've rarely seen a need to do this with the kinds of systems that use *Transaction Scripts* to organize domain logic. Of course, you can ignore classes completely in many languages and just use global functions. However, you'll often find that instantiating a new object helps with threading issues as it makes it easier to isolate data.

I use the term *Transaction Script* because most of the time you'll have one *Transaction Script* for each database transaction. This isn't a 100 percent rule, but it's true to the first approximation.

When to Use It

The glory of *Transaction Script* is its simplicity. Organizing logic this way is natural for applications with only a small amount of logic, and it involves very little overhead either in performance or in understanding.

As the business logic gets more complicated, however, it gets progressively harder to keep it in a well-designed state. One particular problem to watch for

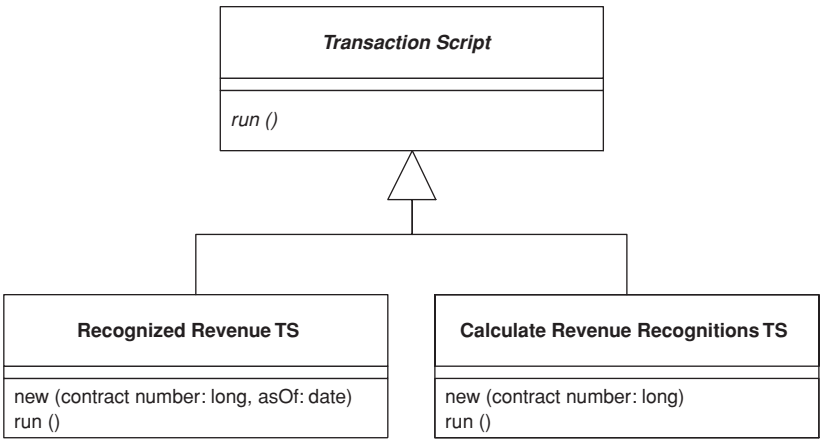


Figure 9.1 *Using commands for* Transaction Script.

is its duplication between transactions. Since the whole point is to handle one transaction, any common code tends to be duplicated.

Careful factoring can alleviate many of these problems, but more complex business domains need to build a *Domain Model (116)*. A *Domain Model (116)* will give you many more options in structuring the code, increasing readability and decreasing duplication.

It's hard to quantify the cutover level, especially when you're more familiar with one pattern than the other. You can refactor a *Transaction Script* design to a *Domain Model (116)* design, but it's a harder change than it otherwise needs to be. Therefore, an early shot is often the best way to move forward.

However much of an object bigot you become, don't rule out *Transaction Script*. There are a lot of simple problems out there, and a simple solution will get you up and running much faster.

The Revenue Recognition Problem

For this pattern, and others that talk about domain logic, I'm going to use the same problem as an illustration. To avoid typing the problem statement several times, I'm just putting it in here.

Revenue recognition is a common problem in business systems. It's all about when you can actually count the money you receive on your books. If I sell you a cup of coffee, it's a simple matter: I give you the coffee, I take your money, and I count the money to the books that nanosecond. For many things it gets complicated, however. Say you pay me a retainer to be available that year. Even if you pay me some ridiculous fee today, I may not be able to put it on my books right away because the service is to be performed over the course of a year. One approach might be to count only one-twelfth of that fee for each month in the year, since you might pull out of the contract after a month when you realize that writing has atrophied my programming skills.

The rules for revenue recognition are many, various, and volatile. Some are set by regulation, some by professional standards, and some by company policy. Revenue tracking ends up being quite a complex problem.

I don't fancy delving into the complexity right now, so instead we'll imagine a company that sells three kinds of products: word processors, databases, and spreadsheets. According to the rules, when you sign a contract for a word processor you can book all the revenue right away. If it's a spreadsheet, you can book one-third today, one-third in sixty days, and one-third in ninety days. If it's a database, you can book one-third today, one-third in thirty days, and one-third in sixty days. There's no basis for these rules other than my own fevered imagination. I'm told that the real rules are equally rational.

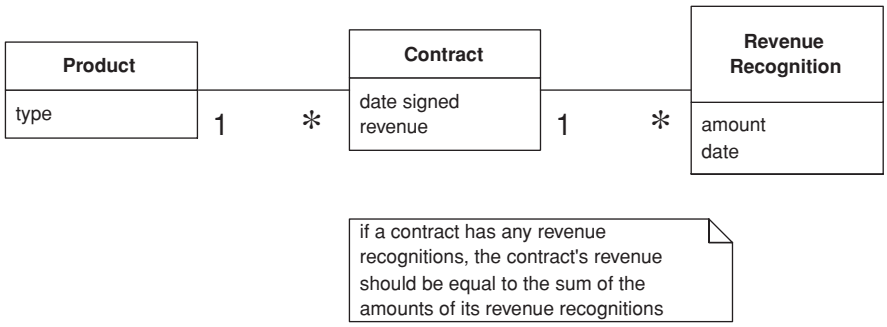


Figure 9.2 *A conceptual model for simplified revenue recognition. Each contract has multiple revenue recognitions that indicate when the various parts of the revenue should be recognized.*

Example: Revenue Recognition (Java)

This example uses two transaction scripts: one to calculate the revenue recognitions for a contract and one to tell how much revenue on a contract has been recognized by a certain date. The database structure has three tables: one for the products, one for the contracts, and one for the revenue recognitions.

```
CREATE TABLE products (ID int primary key, name varchar, type varchar)
CREATE TABLE contracts (ID int primary key, product int, revenue decimal, dateSigned date)
CREATE TABLE revenueRecognitions (contract int, amount decimal, recognizedOn date,
                                  PRIMARY KEY (contract, recognizedOn))
```

The first script calculates the amount of recognition due by a particular day. I can do this in two stages: In the first I select the appropriate rows in the revenue recognitions table; in the second I sum up the amounts.

Many *Transaction Script* designs have *scripts* that operate directly on the database, putting SQL code in the procedure. Here I'm using a simple *Table Data Gateway (144)* to wrap the SQL queries. Since this example is so simple, I'm using a single gateway rather than one for each table. I can define an appropriate find method on the gateway.

```
class Gateway...

   public ResultSet findRecognitionsFor(long contractID, MfDate asof) throws SQLException{
      PreparedStatement stmt = db.prepareStatement(findRecognitionsStatement);
      stmt = db.prepareStatement(findRecognitionsStatement);
      stmt.setLong(1, contractID);
      stmt.setDate(2, asof.toSqlDate());
      ResultSet result = stmt.executeQuery();
      return result;
   }
```

```
   private static final String findRecognitionsStatement =
      "SELECT amount " +
      "  FROM revenueRecognitions " +
      "  WHERE contract = ? AND recognizedOn <= ?";
   private Connection db;
```

I then use the script to sum up based on the result set passed back from the gateway.

```
class RecognitionService...

   public Money recognizedRevenue(long contractNumber, MfDate asOf) {
      Money result = Money.dollars(0);
      try {
         ResultSet rs = db.findRecognitionsFor(contractNumber, asOf);
         while (rs.next()) {
            result = result.add(Money.dollars(rs.getBigDecimal("amount")));
         }
         return result;
      } catch (SQLException e) {throw new ApplicationException (e);
      }
   }
```

When the calculation is as simple as this, you can replace the in-memory script with a call to a SQL statement that uses an aggregate function to sum the amounts.

For calculating the revenue recognitions on an existing contract, I use a similar split. The script on the service carries out the business logic.

```
class RecognitionService...

   public void calculateRevenueRecognitions(long contractNumber) {
      try {
         ResultSet contracts = db.findContract(contractNumber);
         contracts.next();
         Money totalRevenue = Money.dollars(contracts.getBigDecimal("revenue"));
         MfDate recognitionDate = new MfDate(contracts.getDate("dateSigned"));
         String type = contracts.getString("type");
         if (type.equals("S")){
            Money[] allocation = totalRevenue.allocate(3);
            db.insertRecognition
               (contractNumber, allocation[0], recognitionDate);
            db.insertRecognition
               (contractNumber, allocation[1], recognitionDate.addDays(60));
            db.insertRecognition
               (contractNumber, allocation[2], recognitionDate.addDays(90));
         } else if (type.equals("W")){
            db.insertRecognition(contractNumber, totalRevenue, recognitionDate);
         } else if (type.equals("D")) {
            Money[] allocation = totalRevenue.allocate(3);
            db.insertRecognition
               (contractNumber, allocation[0], recognitionDate);
```

```
            db.insertRecognition
               (contractNumber, allocation[1], recognitionDate.addDays(30));
            db.insertRecognition
               (contractNumber, allocation[2], recognitionDate.addDays(60));
         }
      } catch (SQLException e) {throw new ApplicationException (e);
      }
   }
```

Notice that I'm using *Money (488)* to carry out the allocation. When splitting an amount three ways it's very easy to lose a penny.

The *Table Data Gateway (144)* provides support on the SQL. First there's a finder for a contract.

```
class Gateway...

   public ResultSet findContract (long contractID) throws SQLException{
      PreparedStatement stmt = db.prepareStatement(findContractStatement);
      stmt.setLong(1, contractID);
      ResultSet result = stmt.executeQuery();
      return result;
   }
   private static final String findContractStatement =
      "SELECT * " +
      "  FROM contracts c, products p " +
      "  WHERE ID = ? AND c.product = p.ID";
```

And secondly there's a wrapper for the insert.

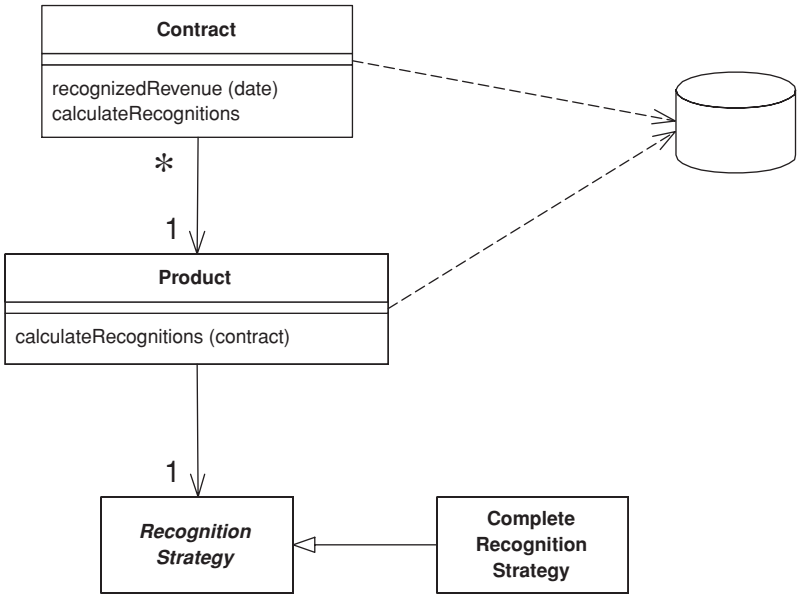
```
class Gateway...

   public void insertRecognition (long contractID, Money amount, MfDate asof) throws SQLException {
      PreparedStatement stmt = db.prepareStatement(insertRecognitionStatement);
      stmt.setLong(1, contractID);
      stmt.setBigDecimal(2, amount.amount());
      stmt.setDate(3, asof.toSqlDate());
      stmt.executeUpdate();
   }
   private static final String insertRecognitionStatement =
      "INSERT INTO revenueRecognitions VALUES (?, ?, ?)";
```

In a Java system the recognition service might be a regular class or a session bean.

As you compare this to the example in *Domain Model (116)*, unless your mind is as twisted as mine, you'll probably be thinking that this is much simpler. The harder thing to imagine is what happens as the rules get more complicated. Typical revenue recognition rules get very involved, varying not just by product but also by date (if the contract was signed before April 15 this rule applies . . .). It's difficult to keep a coherent design with *Transaction Script* once things get that complicated, which is why object bigots like me prefer using a *Domain Model (116)* in these circumstances.

Domain Model

An object model of the domain that incorporates both behavior and data.



At its worst business logic can be very complex. Rules and logic describe many different cases and slants of behavior, and it's this complexity that objects were designed to work with. A *Domain Model* creates a web of interconnected objects, where each object represents some meaningful individual, whether as large as a corporation or as small as a single line on an order form.

How It Works

Putting a *Domain Model* in an application involves inserting a whole layer of objects that model the business area you're working in. You'll find objects that mimic the data in the business and objects that capture the rules the business uses. Mostly the data and process are combined to cluster the processes close to the data they work with.

An OO domain model will often look similar to a database model, yet it will still have a lot of differences. A *Domain Model* mingles data and process, has multivalued attributes and a complex web of associations, and uses inheritance.

As a result I see two styles of *Domain Model* in the field. A simple *Domain Model* looks very much like the database design with mostly one domain object for each database table. A rich *Domain Model* can look different from the database design, with inheritance, strategies, and other [Gang of Four] patterns, and complex webs of small interconnected objects. A rich *Domain Model* is better for more complex logic, but is harder to map to the database. A simple *Domain Model* can use *Active Record (160)*, whereas a rich *Domain Model* requires *Data Mapper (165)*.

Since the behavior of the business is subject to a lot of change, it's important to be able to modify, build, and test this layer easily. As a result you'll want the minimum of coupling from the *Domain Model* to other layers in the system. You'll notice that a guiding force of many layering patterns is to keep as few dependencies as possible between the domain model and other parts of the system.

With a *Domain Model* there are a number of different scopes you might use. The simplest case is a single-user application where the whole object graph is read from a file and put into memory. A desktop application may work this way, but it's less common for a multitiered IS application simply because there are too many objects. Putting every object into memory consumes too much memory and takes too long. The beauty of object-oriented databases is that they give the impression of doing this while moving objects between memory and disk.

Without an OO database you have to do this yourself. Usually a session will involve pulling in an object graph of all the objects involved in it. This will certainly not be all objects and usually not all the classes. Thus, if you're looking at a set of contracts you might pull in only the products referenced by contracts within your working set. If you're just performing calculations on contracts and revenue recognition objects, you may not pull in any product objects at all. Exactly what you pull into memory is governed by your database mapping objects.

If you need the same object graph between calls to the server, you have to save the server state somewhere, which is the subject of the section on saving server state (page 81).

A common concern with domain logic is bloated domain objects. As you build a screen to manipulate orders you'll notice that some of the order behavior is only needed only for it. If you put these responsibilities on the order, the risk is that the Order class will become too big because it's full of responsibilities that are only used in a single use case. This concern leads people to consider whether some responsibility is general, in which case it should sit in the order class, or specific, in which case it should sit in some usage-specific class, which might be a *Transaction Script (110)* or perhaps the presentation itself.

The problem with separating usage-specific behavior is that it can lead to duplication. Behavior that's separated from the order is harder to find, so people tend to not see it and duplicate it instead. Duplication can quickly lead to more complexity and inconsistency, but I've found that bloating occurs much less frequently than predicted. If it does occur, it's relatively easy to see and not difficult to fix. My advice is not to separate usage-specific behavior. Put it all in the object that's the natural fit. Fix the bloating when, and if, it becomes a problem.

Java Implementation

There's always a lot of heat generated when people talk about developing a *Domain Model* in J2EE. Many of the teaching materials and introductory J2EE books suggest that you use entity beans to develop a domain model, but there are some serious problems with this approach, at least with the current (2.0) specification.

Entity beans are most useful when you use Container Managed Persistence (CMP). Indeed, I would say there's little point in using entity beans without CMP. However, CMP is a limited form of object-relational mapping, and it can't support many of the patterns that you need in a rich *Domain Model*.

Entity beans can't be re-entrant. That is, if you call out from one entity bean into another object, that other object (or any object it calls) can't call back into the first entity bean. A rich *Domain Model* often uses re-entrancy, so this is a handicap. It's made worse by the fact that it's hard to spot re-entrant behavior. As a result, some people say that one entity bean should never call another. While this avoids re-entrancy, it very much cripples the advantages using a *Domain Model*.

A *Domain Model* should use fine-grained objects with fine-grained interfaces. Entity beans may be remotable (prior to version 2.0 they had to be). If you have remote objects with fine-grained interfaces you get terrible performance. You can avoid this problem quite easily by only using local interfaces for your entity beans in a *Domain Model*.

To run with entity beans you need a container and a database connected. This will increase build times and also increase the time to do test runs since the tests have to execute against a database. Entity beans are also tricky to debug.

The alternative is to use normal Java objects, although this often causes a surprised reaction—it's amazing how many people think that you can't run regular Java objects in an EJB container. I've come to the conclusion that people forget about regular Java objects because they haven't got a fancy name. That's why, while preparing for a talk in 2000, Rebecca Parsons, Josh Mackenzie, and I gave them one: POJOs (plain old Java objects). A POJO domain model is easy to put together, is quick to build, can run and test outside an EJB container, and is independent of EJB (maybe that's why EJB vendors don't encourage you to use them).

My view on the whole is that using entity beans as a *Domain Model* works if you have pretty modest domain logic. If so, you can build a *Domain Model* that has a simple relationship with the database: where there's mostly one entity bean class per database table. If you have a richer domain logic with inheritance, strategies, and other more sophisticated patterns, you're better off with a POJO domain model and *Data Mapper (165)*, using a commercial tool or with a homegrown layer.

The biggest frustration for me with the use of EJB is that I find a rich *Domain Model* complicated enough to deal with, and I want to keep as independent as possible from the details of the implementation environment. EJB forces itself into your thinking about the *Domain Model*, which means that I have to worry about both the domain and the EJB environment.

When to Use It

If the *how* for a *Domain Model* is difficult because it's such a big subject, the *when* is hard because of both the vagueness and the simplicity of the advice. It all comes down to the complexity of the behavior in your system. If you have complicated and everchanging business rules involving validation, calculations, and derivations, chances are that you'll want an object model to handle them. On the other hand, if you have simple not-null checks and a couple of sums to calculate, a *Transaction Script (110)* is a better bet.

One factor that comes into this is comfortable used the development team is with domain objects. Learning how to design and use a *Domain Model* is a significant exercise—one that has led to many articles on the "paradigm shift" of objects use. It certainly takes practice and coaching to get used to a *Domain Model*, but once used to it I've found that few people want to go back to a *Transaction Script (110)* for any but the simplest problems.

If you're using *Domain Model*, my first choice for database interaction is *Data Mapper (165)*. This will help keep your *Domain Model* independent from the database and is the best approach to handle cases where the *Domain Model* and database schema diverge.

When you use *Domain Model* you may want to consider *Service Layer (133)* to give your *Domain Model* a more distinct API.

Further Reading

Almost any book on OO design will talk about *Domain Models*, since most of what people refer to as OO development is centered around their use.

If you're looking for an introductory book on OO design, my current favorite is [Larman]. For examples of *Domain Model* take a look at [Fowler AP].

[Hay] also gives good examples in a relational context. To build a good *Domain Model* you should have an understanding of conceptual thinking about objects. For this I've always liked [Martin and Odell]. For an understanding of the patterns you'll see in a rich *Domain Model*, or any other OO system, you must read [Gang of Four].

Eric Evans is currently writing a book [Evans] on building *Domain Models*. As I write this I've seen only an early manuscript, but it looks very promising.

Example: Revenue Recognition (Java)

One of the biggest frustrations of describing a *Domain Model* is the fact that any example I show is necessarily simple so you can understand it; yet that simplicity hides the *Domain Model's* strength. You only appreciate these strengths when you have a really complicated domain.

But even if the example can't do justice to why you would want a *Domain Model*, at least it will give you a sense of what one can look like. Therefore, I'm using the same example (page 112) that I used for *Transaction Script (110)*, a little matter of revenue recognition.

An immediate thing to notice is that every class, in this small example (Figure 9.3) contains both behavior and data. Even the humble Revenue Recognition class contains a simple method to find out if that object's value is recognizable on a certain date.

```
class RevenueRecognition...

   private Money amount;
   private MfDate date;
   public RevenueRecognition(Money amount, MfDate date) {
      this.amount = amount;
      this.date = date;
   }
   public Money getAmount() {
      return amount;
   }
   boolean isRecognizableBy(MfDate asOf) {
      return asOf.after(date) || asOf.equals(date);
   }
```

Calculating how much revenue is recognized on a particular date involves both the contract and revenue recognition classes.

```
class Contract...

   private List revenueRecognitions = new ArrayList();
   public Money recognizedRevenue(MfDate asOf) {
      Money result = Money.dollars(0);
      Iterator it = revenueRecognitions.iterator();
```

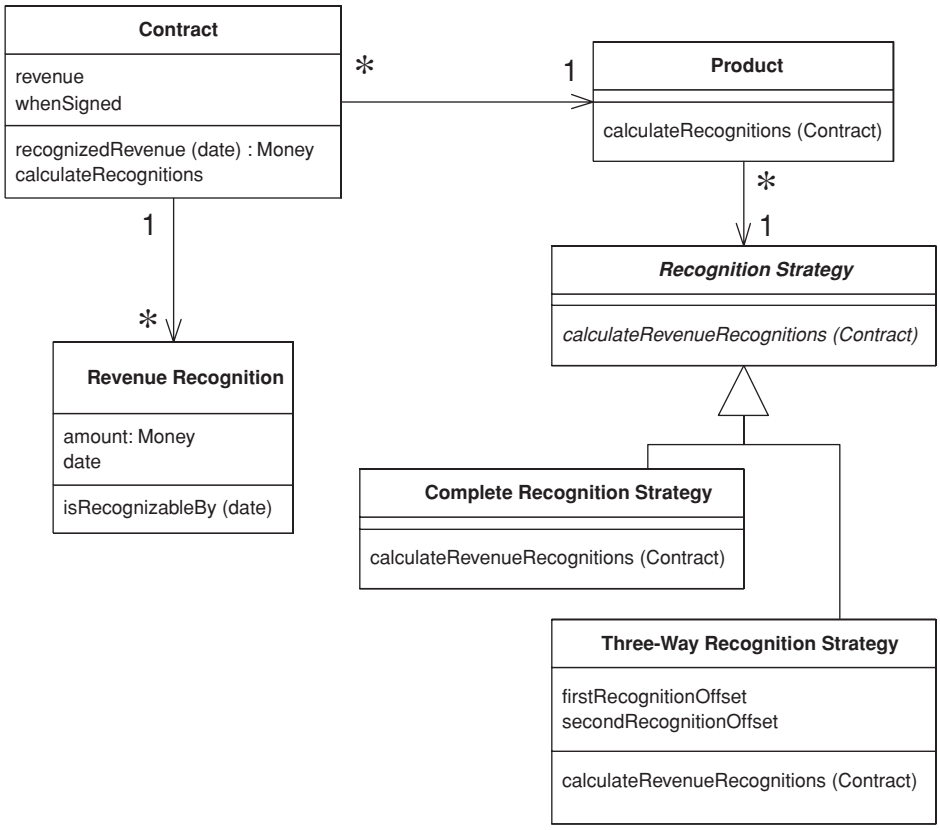


Figure 9.3 *Class diagram of the example classes for a Domain Model.*

```
    while (it.hasNext()) {
       RevenueRecognition r = (RevenueRecognition) it.next();
       if (r.isRecognizableBy(asOf))
          result = result.add(r.getAmount());
    }
    return result;
}
```

A common thing you find in *domain model*s is how multiple classes interact to do even the simplest tasks. This is what often leads to the complaint that with OO programs you spend a lot of time hunting around from class to class trying to find them. There's a lot of merit to this complaint. The value comes as the decision on whether something is recognizable by a certain date gets more complex and as other objects need to know. Containing the behavior on the object that needs to know avoids duplication and reduces coupling between the different objects.

Looking at calculating and creating these revenue recognition objects further demonstrates the notion of lots of little objects. In this case the calculation and creation begin with the customer and are handed off via the product to a strategy hierarchy. The strategy pattern [Gang of Four] is a well-known OO pattern that allows you combine a group of operations in a small class hierarchy. Each instance of product is connected to a single instance of recognition strategy, which determines which algorithm is used to calculate revenue recognition. In this case we have two subclasses of recognition strategy for the two different cases. The structure of the code looks like this:

```
class Contract...

   private Product product;
   private Money revenue;
   private MfDate whenSigned;
   private Long id;
   public Contract(Product product, Money revenue, MfDate whenSigned) {
      this.product = product;
      this.revenue = revenue;
      this.whenSigned = whenSigned;
   }

class Product...

   private String name;
   private RecognitionStrategy recognitionStrategy;
   public Product(String name, RecognitionStrategy recognitionStrategy) {
      this.name = name;
      this.recognitionStrategy = recognitionStrategy;
   }
   public static Product newWordProcessor(String name) {
      return new Product(name, new CompleteRecognitionStrategy());
   }
   public static Product newSpreadsheet(String name) {
      return new Product(name, new ThreeWayRecognitionStrategy(60, 90));
   }
   public static Product newDatabase(String name) {
      return new Product(name, new ThreeWayRecognitionStrategy(30, 60));
   }

class RecognitionStrategy...

   abstract void calculateRevenueRecognitions(Contract contract);

class CompleteRecognitionStrategy...
   void calculateRevenueRecognitions(Contract contract) {
      contract.addRevenueRecognition(new RevenueRecognition(contract.getRevenue(),
                                                            contract.getWhenSigned()));
   }
```

```
class ThreeWayRecognitionStrategy...

   private int firstRecognitionOffset;
   private int secondRecognitionOffset;
   public ThreeWayRecognitionStrategy(int firstRecognitionOffset,
                     int secondRecognitionOffset)
   {
      this.firstRecognitionOffset = firstRecognitionOffset;
      this.secondRecognitionOffset = secondRecognitionOffset;
   }
   void calculateRevenueRecognitions(Contract contract) {
      Money[] allocation = contract.getRevenue().allocate(3);
      contract.addRevenueRecognition(new RevenueRecognition
            (allocation[0], contract.getWhenSigned()));
      contract.addRevenueRecognition(new RevenueRecognition
            (allocation[1], contract.getWhenSigned().addDays(firstRecognitionOffset)));
      contract.addRevenueRecognition(new RevenueRecognition
            (allocation[2], contract.getWhenSigned().addDays(secondRecognitionOffset)));
   }
```

The great value of the strategies is that they provide well-contained plug points to extend the application. Adding a new revenue recognition algorithm involves creating a new subclass and overriding the `calculateRevenueRecognitions` method. This makes it easy to extend the algorithmic behavior of the application.

When you create products, you hook them up with the appropriate strategy objects. I'm doing this in my test code.

```
class Tester...

   private Product word = Product.newWordProcessor("Thinking Word");
   private Product calc = Product.newSpreadsheet("Thinking Calc");
   private Product db = Product.newDatabase("Thinking DB");
```

Once everything is set up, calculating the recognitions requires no knowledge of the strategy subclasses.

```
class Contract...

   public void calculateRecognitions() {
      product.calculateRevenueRecognitions(this);
   }

class Product...

   void calculateRevenueRecognitions(Contract contract) {
      recognitionStrategy.calculateRevenueRecognitions(contract);
   }
```

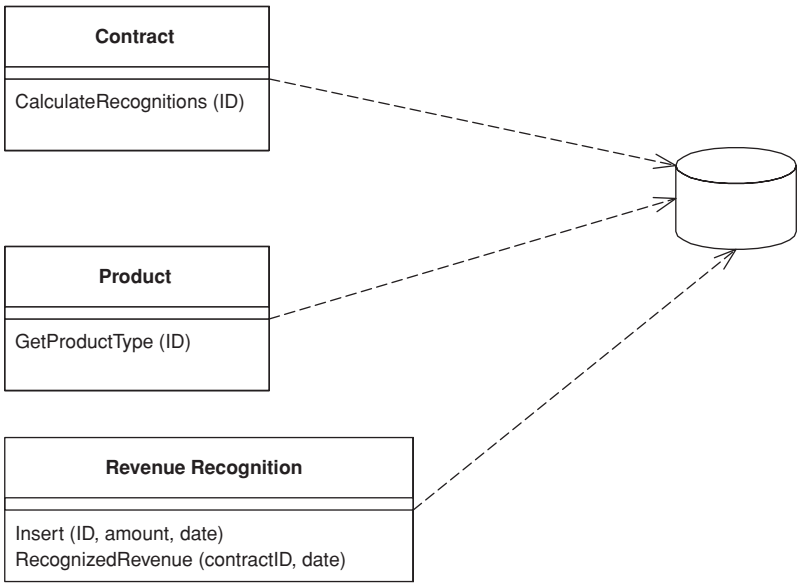
The OO habit of successive forwarding from object to object moves the behavior to the object most qualified to handle it, but it also resolves much of

the conditional behavior. You'll notice that there are no conditionals in this calculation. You set up the decision path when you create the products with the appropriate strategy. Once everything is wired together like this, the algorithms just follow the path. Domain models work very well when you have similar conditionals because the similar conditionals can be factored out into the object structure itself. This moves complexity out of the algorithms and into the relationships between objects. The more similar the logic, the more you find the same network of relationships used by different parts of the system. Any algorithm that's dependent on the type of recognition calculation can follow this particular network of objects.

Notice in this example that I've shown nothing about how the objects are retrieved from, and written to, the database. This is for a couple of reasons. First, mapping a *Domain Model* to a database is always somewhat hard, so I'm chickening out and not providing an example. Second, in many ways the whole point of a *Domain Model* is to hide the database, both from upper layers and from people working the *Domain Model* itself. Thus, hiding it here reflects what it's like to actually program in this environment.

Table Module

A single instance that handles the business logic for all rows in a database table or view.



One of the key messages of object orientation is bundling the data with the behavior that uses it. The traditional object-oriented approach is based on objects with identity, along the lines of *Domain Model (116)*. Thus, if we have an Employee class, any instance of it corresponds to a particular employee. This scheme works well because once we have a reference to an employee, we can execute operations, follow relationships, and gather data on him.

One of the problems with *Domain Model (116)* is the interface with relational databases. In many ways this approach treats the relational database like a crazy aunt who's shut up in an attic and whom nobody wants to talk about. As a result you often need considerable programmatic gymnastics to pull data in and out of the database, transforming between two different representations of the data.

A *Table Module* organizes domain logic with one class per table in the database, and a single instance of a class contains the various procedures that will act on the data. The primary distinction with *Domain Model (116)* is that, if you have many orders, a *Domain Model (116)* will have one order object per order while a *Table Module* will have one object to handle all orders.

How It Works

The strength of *Table Module* is that it allows you to package the data and behavior together and at the same time play to the strengths of a relational database. On the surface *Table Module* looks much like a regular object. The key difference is that it has no notion of an identity for the objects it's working with. Thus, if you want to obtain the address of an employee, you use a method like `anEmployeeModule.getAddress(long employeeID)`. Every time you want to do something to a particular employee you have to pass in some kind of identity reference. Often this will be the primary key used in the database.

Usually you use *Table Module* with a backing data structure that's table oriented. The tabular data is normally the result of a SQL call and is held in a *Record Set (508)* that mimics a SQL table. The *Table Module* gives you an explicit method-based interface that acts on that data. Grouping the behavior with the table gives you many of the benefits of encapsulation in that the behavior is close to the data it will work on.

Often you'll need behavior from multiple *Table Modules* in order to do some useful work. Many times you see multiple *Table Modules* operating on the same *Record Set (508)* (Figure 9.4).

The most obvious example of *Table Module* is the use of one for each table in the database. However, if you have interesting queries and views in the database you can have *Table Modules* for them as well.

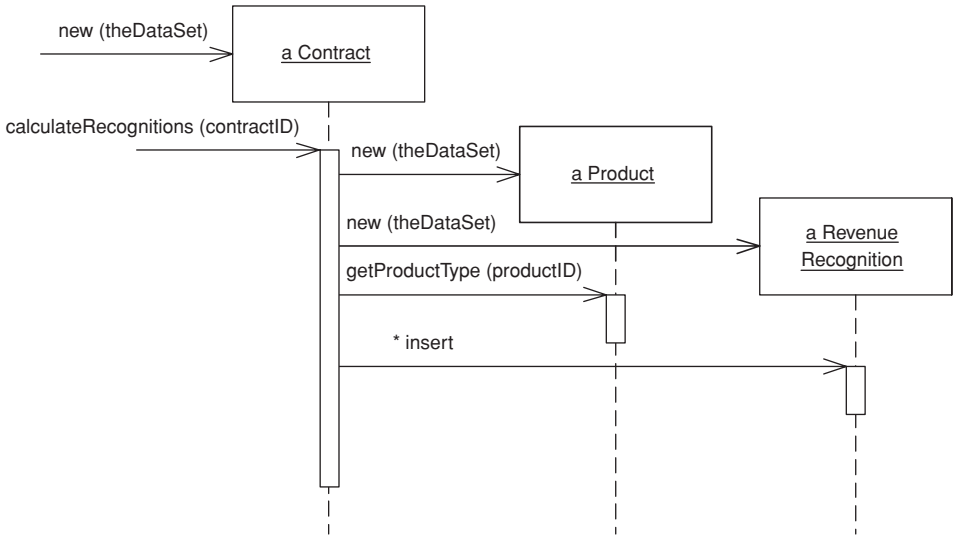


Figure 9.4 *Several* Table Modules *can collaborate with a single* Record Set (508).

The *Table Module* may be an instance or it may be a collection of static methods. The advantage of an instance is that it allows you to initialize the *Table Module* with an existing record set, perhaps the result of a query. You can then use the instance to manipulate the rows in the record set. Instances also make it possible to use inheritance, so we can write a rush contract module that contains additional behavior to the regular contract.

The *Table Module* may include queries as factory methods. The alternative is a *Table Data Gateway (144)*, but the disadvantage of this is having an extra *Table Data Gateway (144)* class and mechanism in the design. The advantage is that you can use a single *Table Module* on data from different data sources, since you use a different *Table Data Gateway (144)* for each data source.

When you use a *Table Data Gateway (144)* the application first uses the *Table Data Gateway (144)* to assemble data in a *Record Set (508)*. You then create a *Table Module* with the *Record Set (508)* as an argument. If you need behavior from multiple *Table Modules*, you can create them with the same *Record Set (508)*. The *Table Module* can then do business logic on the *Record Set (508)* and pass the modified *Record Set (508)* to the presentation for display and editing using the table-aware widgets. The widgets can't tell if the record

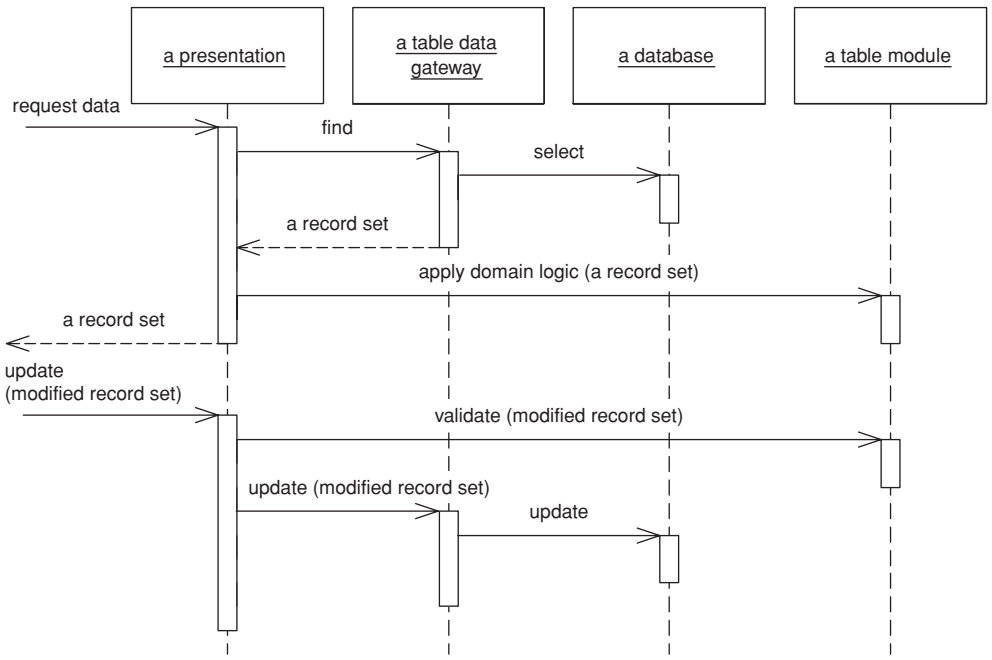


Figure 9.5 *Typical interactions for the layers around a* Table Module.

sets came directly from the relational database or if a *Table Module* manipulated the data on the way out. After modification in the GUI, the data set goes back to the *Table Module* for validation before it's saved to the database. One of the benefits of this style is that you can test the *Table Module* by creating a *Record Set (508)* in memory without going to the database.

The word "table" in the pattern name suggests that you have one *Table Module* per table in the database. While this is true to the first approximation, it isn't completely true. It's also useful to have a *Table Module* for commonly used views or other queries. Indeed, the structure of the *Table Module* doesn't really depend on the structure of tables in the database but more on the virtual tables perceived by the application, including views and queries.

When to Use It

Table Module is very much based on table-oriented data, so obviously using it makes sense when you're accessing tabular data using *Record Set (508)*. It also puts that data structure very much in the center of the code, so you also want the way you access the data structure to be fairly straightforward.

However, *Table Module* doesn't give you the full power of objects in organizing complex logic. You can't have direct instance-to-instance relationships, and polymorphism doesn't work well. So, for handling complicated domain logic, a *Domain Model (116)* is a better choice. Essentially you have to trade off *Domain Model (116)*'s ability to handle complex logic against *Table Module*'s easier integration with the underlying table-oriented data structures.

If the objects in a *Domain Model (116)* and the database tables are relatively similar, it may be better to use a *Domain Model (116)* that uses *Active Record (160)*. *Table Module* works better than a combination of *Domain Model (116)* and *Active Record (160)* when other parts of the application are based on a common table-oriented data structure. That's why you don't see *Table Module* very much in the Java environment, although that may change as row sets become more widely used.

The most well-known situation in which I've come across this pattern is in Microsoft COM designs. In COM (and .NET) the *Record Set (508)* is the primary repository of data in an application. Record sets can be passed to the UI, where data-aware widgets display information. Microsoft's ADO libraries give you a good mechanism to access the relational data as record sets. In this situation *Table Module* allows you to fit business logic into the application in a well-organized manner, without losing the way the various elements work on the tabular data.

Example: Revenue Recognition with a Table Module (C#)

Time to revisit the revenue recognition example (page 112) I used in the other domain modeling patterns, this time with a *Table Module*. To recap, our mission is to recognize revenue on orders when the rules vary depending on the product type. In this example we have different rules for word processors, spreadsheets, and databases.

Table Module is based on a data schema of some kind, usually a relational data model (although in the future we may well see an XML model used in a similar way). In this case I'll use the relational schema from Figure 9.6.

The classes that manipulate this data are in pretty much the same form; there's one *Table Module* class for each table. In the .NET architecture a data set object provides an in-memory representation of a database structure. It thus makes sense to create classes that operate on this data set. Each *Table Module* class has a data member of a data table, which is the .NET system class corresponding to a table within the data set. This ability to read a table is common to all *Table Modules* and so can appear in a *Layer Supertype (475)*.

```
class TableModule...

      protected DataTable table;
      protected TableModule(DataSet ds, String tableName) {
         table = ds.Tables[tableName];
      }
```

The subclass constructor calls the superclass constructor with the correct table name.

```
class Contract...

      public Contract (DataSet ds) : base (ds, "Contracts") {}
```

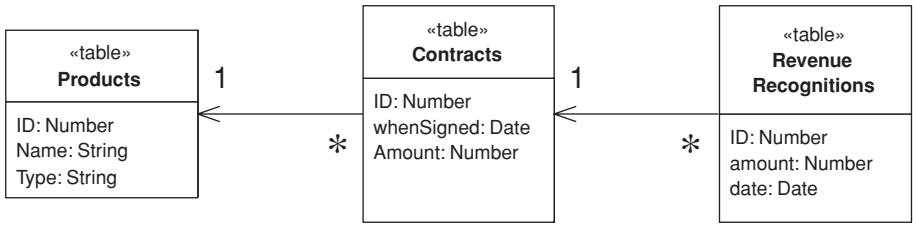


Figure 9.6 *Database schema for revenue recognition.*

This allows you to create a new *Table Module* just by passing in a data set to *Table Module*'s constructor

```
contract = new Contract(dataset);
```

which keeps the code that creates the data set away from the *Table Modules*, following the guidelines of ADO.NET.

A useful feature is the C# indexer, which gets to a particular row in the data table given the primary key.

```
class Contract...

public DataRow this [long key] {
get {
   String filter = String.Format("ID = {0}", key);
   return table.Select(filter)[0];
   }
}
```

The first piece of functionality calculates the revenue recognition for a contract, updating the revenue recognition tables accordingly. The amount recognized depends on the kind of product we have. Since this behavior mainly uses data from the contract table, I decided to add the method to the contract class.

```
class Contract...

public void CalculateRecognitions (long contractID) {
   DataRow contractRow = this[contractID];
   Decimal amount = (Decimal)contractRow["amount"];
   RevenueRecognition rr = new RevenueRecognition (table.DataSet);
   Product prod = new Product(table.DataSet);
   long prodID = GetProductId(contractID);
   if (prod.GetProductType(prodID) == ProductType.WP) {
      rr.Insert(contractID, amount, (DateTime) GetWhenSigned(contractID));
   } else if (prod.GetProductType(prodID) == ProductType.SS) {
      Decimal[] allocation = allocate(amount,3);
      rr.Insert(contractID, allocation[0], (DateTime) GetWhenSigned(contractID));
      rr.Insert(contractID, allocation[1], (DateTime)
         GetWhenSigned(contractID).AddDays(60));
      rr.Insert(contractID, allocation[2], (DateTime)
         GetWhenSigned(contractID).AddDays(90));
   } else if (prod.GetProductType(prodID) == ProductType.DB) {
      Decimal[] allocation = allocate(amount,3);
      rr.Insert(contractID, allocation[0], (DateTime) GetWhenSigned(contractID));
      rr.Insert(contractID, allocation[1], (DateTime)
         GetWhenSigned(contractID).AddDays(30));
      rr.Insert(contractID, allocation[2], (DateTime)
         GetWhenSigned(contractID).AddDays(60));
   } else throw new Exception("invalid product id");
}
private Decimal[] allocate(Decimal amount, int by) {
   Decimal lowResult = amount / by;
```

```
        lowResult = Decimal.Round(lowResult,2);
        Decimal highResult = lowResult + 0.01m;
        Decimal[] results = new Decimal[by];
        int remainder = (int) amount % by;
        for (int i = 0; i < remainder; i++) results[i] = highResult;
        for (int i = remainder; i < by; i++) results[i] = lowResult;
        return results;
    }
```

Usually I would use *Money (488)* here, but for variety's sake I'll show this using a decimal. I use an allocation method similar to the one I use for *Money (488)*.

To carry this out, we need some behavior that's defined on the other classes. The product needs to be able to tell us which type it is. We can do this with an enum for the product type and a lookup method.

```
public enum ProductType {WP, SS, DB};
```

```
class Product...

    public ProductType GetProductType (long id) {
        String typeCode = (String) this[id]["type"];
        return (ProductType) Enum.Parse(typeof(ProductType), typeCode);
    }
```

`GetProductType` encapsulates the data in the data table. There's an argument for doing this for all columns of data, as opposed to accessing them directly as I did with the amount on the contract. While encapsulation is generally a Good Thing, I don't use it here because it doesn't fit with the assumption of the environment that different parts of the system access the data set directly. There's no encapsulation when the data set moves over to the UI, so column access functions only make sense when there's some additional functionality to be done, such as converting a string to a product type.

This is also a good time to mention that, although I'm using an untyped data set here because these are more common on different platforms, there's a strong argument (page 509) for using .NET's strongly typed data set.

The other additional behavior is inserting a new revenue recognition record.

```
class RevenueRecognition...

    public long Insert (long contractID, Decimal amount, DateTime date) {
        DataRow newRow = table.NewRow();
        long id = GetNextID();
        newRow["ID"] = id;
        newRow["contractID"] = contractID;
        newRow["amount"] = amount;
        newRow["date"]= String.Format("{0:s}", date);
        table.Rows.Add(newRow);
        return id;
    }
```

Again, the point of this method is less to encapsulate the data row and more to have a method instead of several lines of code that are repeated.

The second piece of functionality is to sum up all the revenue recognized on a contract by a given date. Since this uses the revenue recognition table it makes sense to define the method there.

```
class RevenueRecognition...

      public Decimal RecognizedRevenue (long contractID, DateTime asOf) {
         String filter = String.Format("ContractID = {0} AND date <= #{1:d}#", contractID,asOf);
         DataRow[] rows = table.Select(filter);
         Decimal result = 0m;
         foreach (DataRow row in rows) {
            result += (Decimal)row["amount"];
         }
         return result;
      }
```

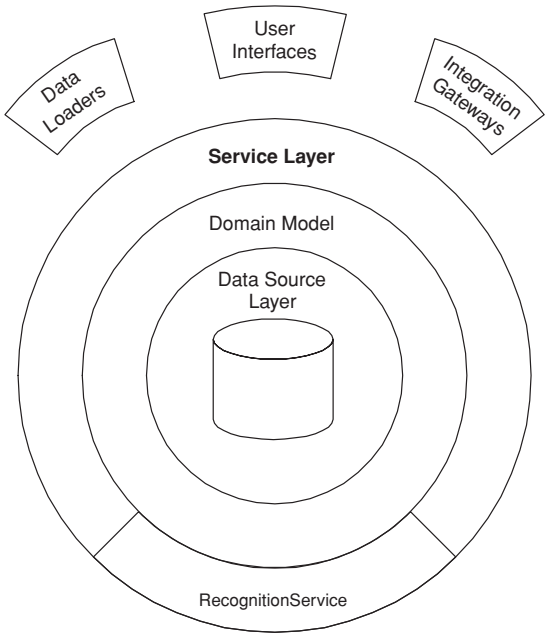
This fragment takes advantage of the really nice feature of ADO.NET that allows you to define a where clause and then select a subset of the data table to manipulate. Indeed, you can go further and use an aggregate function.

```
class RevenueRecognition...

      public Decimal RecognizedRevenue2 (long contractID, DateTime asOf) {
         String filter = String.Format("ContractID = {0} AND date <= #{1:d}#", contractID,asOf);
         String computeExpression = "sum(amount)";
         Object sum = table.Compute(computeExpression, filter);
         return (sum is System.DBNull) ? 0 : (Decimal) sum;
      }
```

Service Layer

by Randy Stafford

Defines an application's boundary with a layer of services that establishes a set of available operations and coordinates the application's response in each operation.



Enterprise applications typically require different kinds of interfaces to the data they store and the logic they implement: data loaders, user interfaces, integration gateways, and others. Despite their different purposes, these interfaces often need common interactions with the application to access and manipulate its data and invoke its business logic. The interactions may be complex, involving transactions across multiple resources and the coordination of several responses to an action. Encoding the logic of the interactions separately in each interface causes a lot of duplication.

A *Service Layer* defines an application's boundary [Cockburn PloP] and its set of available operations from the perspective of interfacing client layers. It

encapsulates the application's business logic, controlling transactions and coordinating responses in the implementation of its operations.

How It Works

A *Service Layer* can be implemented in a couple of different ways, without violating the defining characteristics stated above. The differences appear in the allocation of responsibility behind the *Service Layer* interface. Before I delve into the various implementation possibilities, let me lay a bit of groundwork.

Kinds of "Business Logic" Like *Transaction Script (110)* and *Domain Model (116)*, *Service Layer* is a pattern for organizing business logic. Many designers, including me, like to divide "business logic" into two kinds: "domain logic," having to do purely with the problem domain (such as strategies for calculating revenue recognition on a contract), and "application logic," having to do with application responsibilities [Cockburn UC] (such as notifying contract administrators, and integrated applications, of revenue recognition calculations). Application logic is sometimes referred to as "workflow logic," although different people have different interpretations of "workflow."

Domain Models (116) are preferable to *Transaction Scripts (110)* for avoiding domain logic duplication and for managing complexity using classical design patterns. But putting application logic into pure domain object classes has a couple of undesirable consequences. First, domain object classes are less reusable across applications if they implement application-specific logic and depend on application-specific packages. Second, commingling both kinds of logic in the same classes makes it harder to reimplement the application logic in, say, a workflow tool if that should ever become desirable. For these reasons *Service Layer* factors each kind of business logic into a separate layer, yielding the usual benefits of layering and rendering the pure domain object classes more reusable from application to application.

Implementation Variations The two basic implementation variations are the domain facade approach and the operation script approach. In the **domain facade** approach a *Service Layer* is implemented as a set of thin facades over a *Domain Model (116)*. The classes implementing the facades don't implement any business logic. Rather, the *Domain Model (116)* implements all of the business logic. The thin facades establish a boundary and set of operations through which client layers interact with the application, exhibiting the defining characteristics of *Service Layer.*

In the **operation script** approach a *Service Layer* is implemented as a set of thicker classes that directly implement application logic but delegate to encapsulated domain object classes for domain logic. The operations available to clients of a *Service Layer* are implemented as scripts, organized several to a class defining a subject area of related logic. Each such class forms an application "service," and it's common for service type names to end with "Service." A *Service Layer* is comprised of these application service classes, which should extend a *Layer Supertype (475)*, abstracting their responsibilities and common behaviors.

To Remote or Not to Remote The interface of a *Service Layer* class is coarse grained almost by definition, since it declares a set of application operations available to interfacing client layers. Therefore, *Service Layer* classes are well suited to remote invocation from an interface granularity perspective.

However, remote invocation comes at the cost of dealing with object distribution. It likely entails a lot of extra work to make your *Service Layer* method signatures deal in *Data Transfer Objects (401)*. Don't underestimate the cost of this work, especially if you have a complex *Domain Model (116)* and rich editing UIs for complex update use cases! It's significant, and it's painful—perhaps second only to the cost and pain of object-relational mapping. Remember the First Law of Distributed Object Design (page 89).

My advice is to start with a locally invocable *Service Layer* whose method signatures deal in domain objects. Add remotability when you need it (if ever) by putting *Remote Facades (388)* on your *Service Layer* or having your *Service Layer* objects implement remote interfaces. If your application has a Web-based UI or a Web-services-based integration gateway, there's no law that says your business logic has to run in a separate process from your server pages and Web services. In fact, you can save yourself some development effort and runtime response time, without sacrificing scalability, by starting out with a colocated approach.

Identifying Services and Operations Identifying the operations needed on a *Service Layer* boundary is pretty straightforward. They're determined by the needs of *Service Layer* clients, the most significant (and first) of which is typically a user interface. Since a user interface is designed to support the use cases that actors want to perform with an application, the starting point for identifying *Service Layer* operations is the use case model and the user interface design for the application.

Disappointing as it is, many of the use cases in an enterprise application are fairly boring "CRUD" (create, read, update, delete) use cases on domain objects—create one of these, read a collection of those, update this other

thing. My experience is that there's almost always a one-to-one correspondence between CRUD use cases and *Service Layer* operations.

The application's responsibilities in carrying out these use cases, however, may be anything but boring. Validation aside, the creation, update, or deletion of a domain object in an application increasingly requires notification of other people and other integrated applications. These responses must be coordinated, and transacted atomically, by *Service Layer* operations.

If only it were as straightforward to identify *Service Layer* abstractions to group related operations. There are no hard-and-fast prescriptions in this area; only vague heuristics. For a sufficiently small application, it may suffice to have but one abstraction, named after the application itself. In my experience larger applications are partitioned into several "subsystems," each of which includes a complete vertical slice through the stack of architecture layers. In this case I prefer one abstraction per subsystem, named after the subsystem. Other possibilities include abstractions reflecting major partitions in a domain model, if these are different from the subsystem partitions (e.g., `ContractsService`, `ProductsService`), and abstractions named after thematic application behaviors (e.g., `RecognitionService`).

Java Implementation

In both the domain facade approach and the operation script approach, a *Service Layer* class can be implemented as either a POJO (plain old Java object) or a stateless session bean. The trade-off pits ease of testing against ease of transaction control. POJOs might be easier to test, since they don't have to be deployed in an EJB container to run, but it's harder for a POJO *Service Layer* to hook into distributed container-managed transaction services, especially in interservice invocations. EJBs, on the other hand, come with the potential for container-managed distributed transactions but have to be deployed in a container before they can be tested and run. Choose your poison.

My preferred way of applying a *Service Layer* in J2EE is with EJB 2.0 stateless session beans, using local interfaces, and the operation script approach, delegating to POJO domain object classes. It's just so darned convenient to implement a *Service Layer* using stateless session bean, because of the distributed container-managed transactions provided by EJB. Also, with the local interfaces introduced in EJB 2.0, a *Service Layer* can exploit the valuable transaction services while avoiding the thorny object distribution issues.

On a related Java-specific note, let me differentiate *Service Layer* from the *Session Facade* pattern documented in the J2EE patterns literature [Alur et al.] and [Marinescu]. *Session Facade* was motivated by the desire

to avoid the performance penalty of too many remote invocations on entity beans; it therefore prescribes facading entity beans with session beans. *Service Layer* is motivated instead by factoring responsibility to avoid duplication and promote reusability; it's an architecture pattern that transcends technology. In fact, the application boundary pattern [Cockburn PloP] that inspired *Service Layer* predates EJB by three years. *Session Facade* may be in the spirit of *Service Layer* but, as currently named, scoped, and presented, is not the same.

When to Use It

The benefit of *Service Layer* is that it defines a common set of application operations available to many kinds of clients and it coordinates an application's response in each operation. The response may involve application logic that needs to be transacted atomically across multiple transactional resources. Thus, in an application with more than one kind of client of its business logic, and complex responses in its use cases involving multiple transactional resources, it makes a lot of sense to include a *Service Layer* with container-managed transactions, even in an undistributed architecture.

The easier question to answer is probably when not to use it. You probably don't need a *Service Layer* if your application's business logic will only have one kind of client—say, a user interface—and its use case responses don't involve multiple transactional resources. In this case your Page Controllers can manually control transactions and coordinate whatever response is required, perhaps delegating directly to the Data Source layer.

But as soon as you envision a second kind of client, or a second transactional resource in use case responses, it pays to design in a *Service Layer* from the beginning.

Further Reading

There's not a great deal of prior art on *Service Layer*, whose inspiration is Alistair Cockburn's application boundary pattern [Cockburn PloP]. In the remotable services vein [Alpert, et al.] discuss the role of facades in distributed systems. Compare and contrast this with the various presentations of Session Facade [Alur et al.] and [Marinescu]. On the topic of application responsibilities that must be coordinated within *Service Layer* operations, Cockburn's description of use cases as a contract for behavior [Cockburn UC] is very helpful. An earlier background reference is the Fusion methodology's recognition of "system operations" [Coleman et al.].

Example: Revenue Recognition (Java)

This example continues the revenue recognition example of the *Transaction Script (110)* and *Domain Model (116)* patterns, demonstrating how *Service Layer* is used to script application logic and delegate for domain logic in a *Service Layer* operation. It uses the operation script approach to implement a *Service Layer*, first with POJOs and then with EJBs.

To make the demonstration we expand the scenario to include some application logic. Suppose the use cases for the application require that, when the revenue recognitions for a contract are calculated, the application must respond by sending an e-mail notification of that event to a designated contract administrator and by publishing a message using message-oriented middleware to notify other integrated applications.

We start by changing the RecognitionService class from the *Transaction Script (110)* example to extend a *Layer Supertype (475)* and to use a couple of *Gateways (466)* in carrying out application logic. This yields the class diagram of Figure 9.7. RecognitionService becomes a POJO implementation of a *Service Layer* application service, and its methods represent two of the operations available at the application's boundary.

The methods of the RecognitionService class script the application logic of the operations, delegating to domain object classes (of the example from *Domain Model (116)*) for domain logic.

```
public class ApplicationService {
   protected EmailGateway getEmailGateway() {
      //return an instance of EmailGateway
   }
   protected IntegrationGateway getIntegrationGateway() {
      //return an instance of IntegrationGateway
   }
}
public interface EmailGateway {
   void sendEmailMessage(String toAddress, String subject, String body);
}
public interface IntegrationGateway {
   void publishRevenueRecognitionCalculation(Contract contract);
}
public class RecognitionService
extends ApplicationService {
   public void calculateRevenueRecognitions(long contractNumber) {
      Contract contract = Contract.readForUpdate(contractNumber);
      contract.calculateRecognitions();
      getEmailGateway().sendEmailMessage(
         contract.getAdministratorEmailAddress(),
         "RE: Contract #" + contractNumber,
         contract + " has had revenue recognitions calculated.");
      getIntegrationGateway().publishRevenueRecognitionCalculation(contract);
   }
```

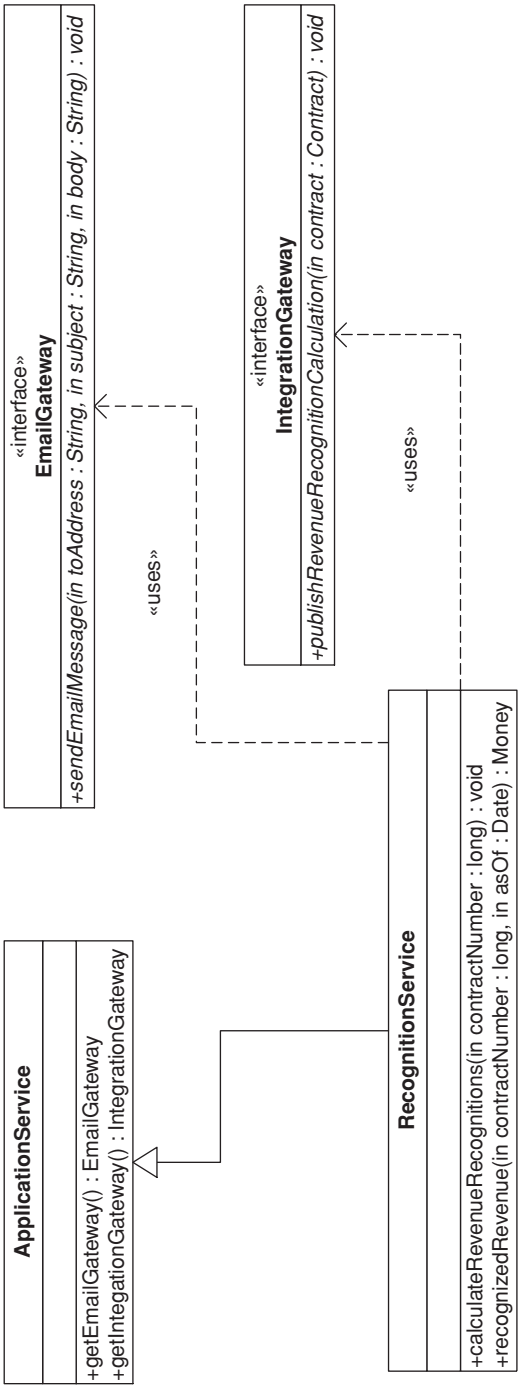


Figure 9.7 *RecognitionService POJO class diagram.*

```
    public Money recognizedRevenue(long contractNumber, Date asOf) {
        return Contract.read(contractNumber).recognizedRevenue(asOf);
    }
}
```

Persistence details are again left out of the example. Suffice it to say that the Contract class implements static methods to read contracts from the Data Source layer by their numbers. One of these methods has a name revealing an intention to update the contract that's read, which allows an underlying *Data Mapper (165)* to register the read object(s) with for example, a *Unit of Work (184)*.

Transaction control details are also left out of the example. The `calculateRevenueRecognitions()` method is inherently transactional because, during its execution, persistent contract objects are modified via addition of revenue recognitions; messages are enqueued in message-oriented middleware; and e-mail messages are sent. All of these responses must be transacted atomically because we don't want to send e-mail and publish messages to other applications if the contract changes fail to persist.

In the J2EE platform we can let the EJB container manage distributed transactions by implementing application services (and *Gateways (466)*) as stateless session beans that use transactional resources. Figure 9.8 shows the class diagram of a `RecognitionService` implementation that uses EJB 2.0 local interfaces and the "business interface" idiom. In this implementation a *Layer Supertype (475)* is still used, providing default implementations of the bean implementation class methods required by EJB, in addition to the application-specific methods. If we assume that the `EmailGateway` and `IntegrationGateway` interfaces are also "business interfaces" for their respective stateless session beans, then control of the distributed transaction is achieved by declaring the `calculateRevenueRecognitions`, `sendEmailMessage`, and `publishRevenueRecognitionCalculation` methods to be transactional. The `RecognitionService` methods from the POJO example move unchanged to `RecognitionServiceBeanImpl`.

The important point about the example is that the *Service Layer* uses both operation scripting and domain object classes in coordinating the transactional response of the operation. The `calculateRevenueRecognitions` method scripts the application logic of the response required by the application's use cases, but it delegates to the domain object classes for domain logic. It also presents a couple of techniques for combating duplicated logic within operation scripts of a *Service Layer*. Responsibilities are factored into different objects (e.g., *Gateways (466)*) that can be reused via delegation. A *Layer Supertype (475)* provides convenient access to these other objects.

Some might argue that a more elegant implementation of the operation script would use the Observer pattern [Gang of Four], but Observer is difficult

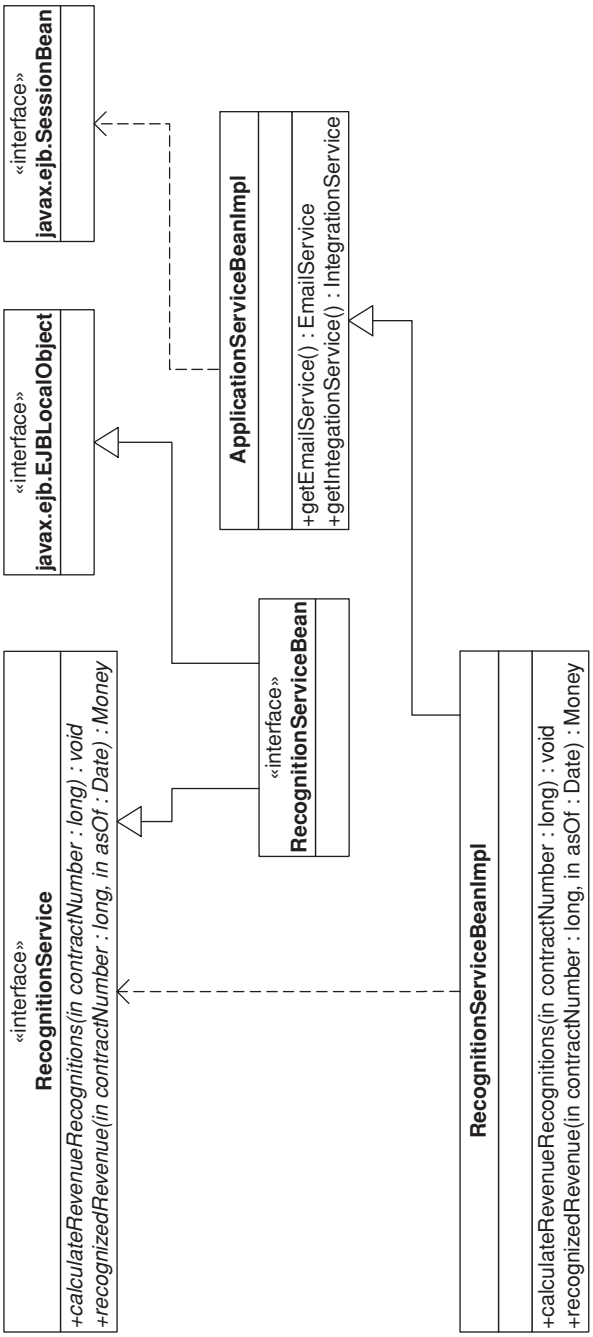


Figure 9.8 *RecognitionService EJB class diagram.*

to implement in a stateless, multithreaded *Service Layer*. In my opinion the open code of the operation script is clearer and simpler.

Some might also argue that the application logic responsibilities could be implemented in domain object methods, such as `Contract.calculateRevenueRecognitions()`, or even in the data source layer, thereby eliminating the need for a separate *Service Layer*. However, I find those allocations of responsibility undesirable for a number of reasons. First, domain object classes are less reusable across applications if they implement application-specific logic (and depend on application-specific *Gateways (466)*, and the like). They should model the parts of the problem domain that are of interest to the application, which doesn't mean all of the application's use case responsibilities. Second, encapsulating application logic in a "higher" layer dedicated to that purpose (which the data source layer isn't) facilitates changing the implementation of that layer—perhaps to use a workflow engine.

As an organization pattern for the logic layer of an enterprise application, *Service Layer* combines scripting and domain object classes, leveraging the best aspects of both. Several variations are possible in a *Service Layer* implementation—for example, domain facades or operation scripts, POJOs or session beans, or a combination of both. *Service Layer* can be designed for local invocation, remote invocation, or both. Most important, regardless of these variations, this pattern lays the foundation for encapsulated implementation of an application's business logic and consistent invocation of that logic by its various clients.