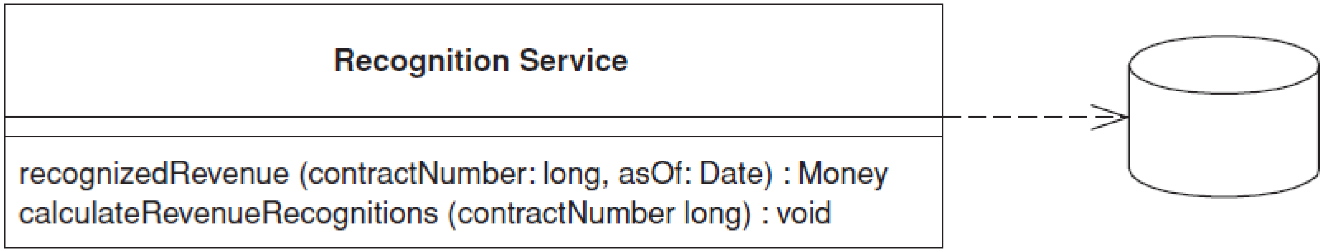
SOEN 387: Web-Based Enterprise Application Design

Chapter 10. Domain Logic Patterns

Most business applications can be thought of as a series of transactions.

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.



Recognition Service

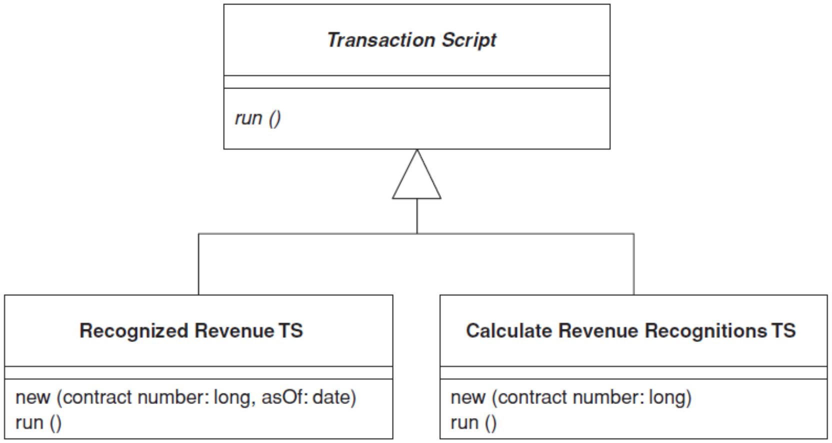
recognizedRevenue (contractNumber: long, asOf: Date) : Money
calculateRevenueRecognitions (contractNumber long) : void

Transaction Script

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*.

The other way is to have each *Transaction Script* in its own class (see Figure below).

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.


Transaction Script
run ()
Recognized Revenue TS
new (contract number: long, asOf: date)
run ()
Calculate Revenue Recognitions TS
new (contract number: long)
run ()

The main property 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 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* .

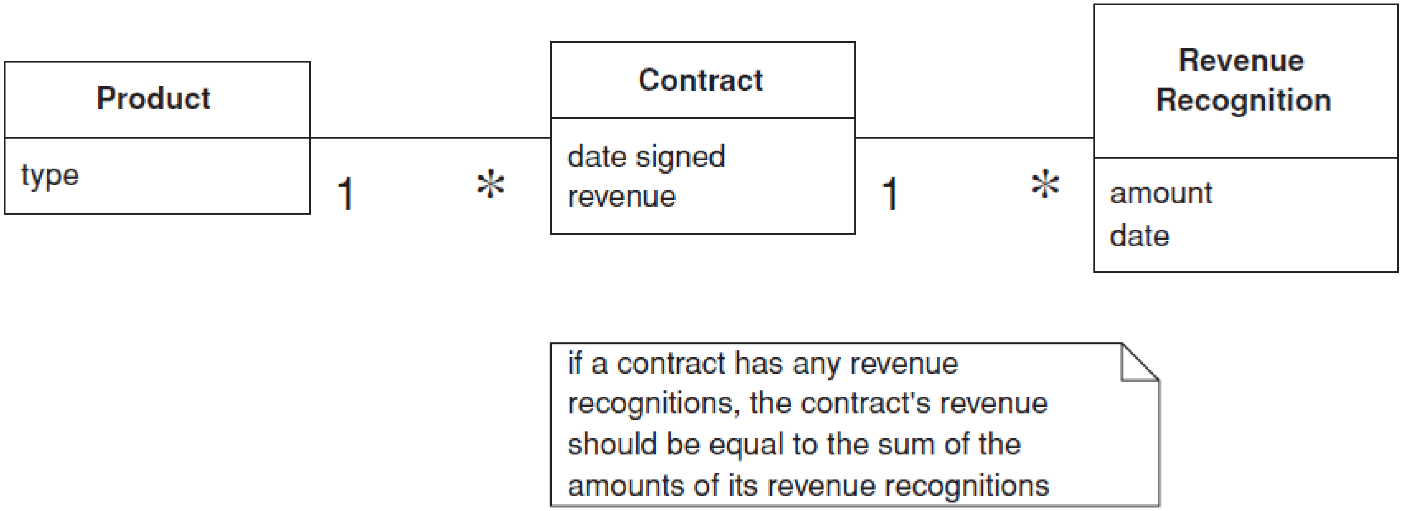
A *Domain Model* will give you many more options in structuring the code, increasing readability and decreasing duplication.

The Revenue Recognition Problem

Revenue recognition is a common problem in business systems. It is all about when you can actually count the money you receive on your books.

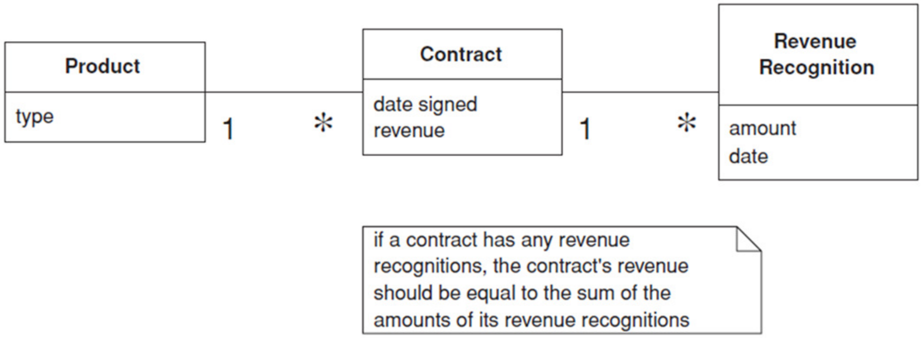
The rules for revenue recognition are many, various, and volatile. Some are set by regulation, some by professional standards, and some by company policy. Revenue tracking ends up being quite a complex problem.

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.



if a contract has any revenue recognitions, the contract's revenue should be equal to the sum of the amounts of its revenue recognitions

Example: Revenue Recognition (Java)


Product
type
1
*
Contract
date signed
revenue
1
*
Revenue Recognition
amount
date
if a contract has any revenue recognitions, the contract's revenue should be equal to the sum of the amounts of its revenue recognitions

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.

Here is done in two stages: In the first, select the appropriate rows in the revenue recognitions table; in the second sum up the amounts.

Many Transaction Script designs have scripts that operate directly on the Database by putting SQL code in the procedure. Here a simple Table Data Gateway to wrap the SQL queries is sued .

```
class Gateway...
public ResultSet findRecognitionsFor(long contractID, MfDate asof) throws SQLException{
PreparedStatement 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;
```

The *Table Data Gateway* 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";
```

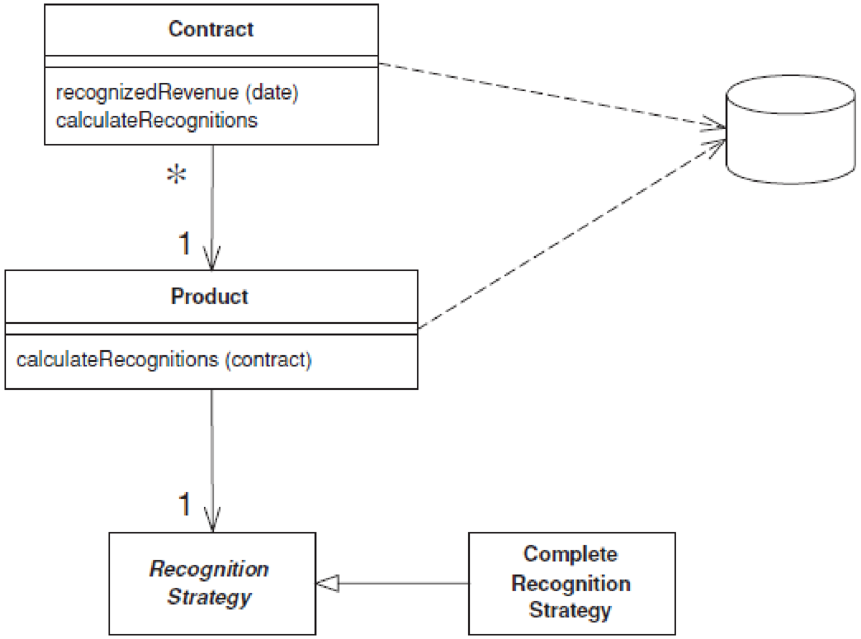
And secondly there’s a wrapper for the insert.

```
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 (?, ?, ?)";
```

Domain Model

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.

If you are using Domain Model, good practice to use Data Mapper for database interaction. This will help keep your Domain Model independent from the database.

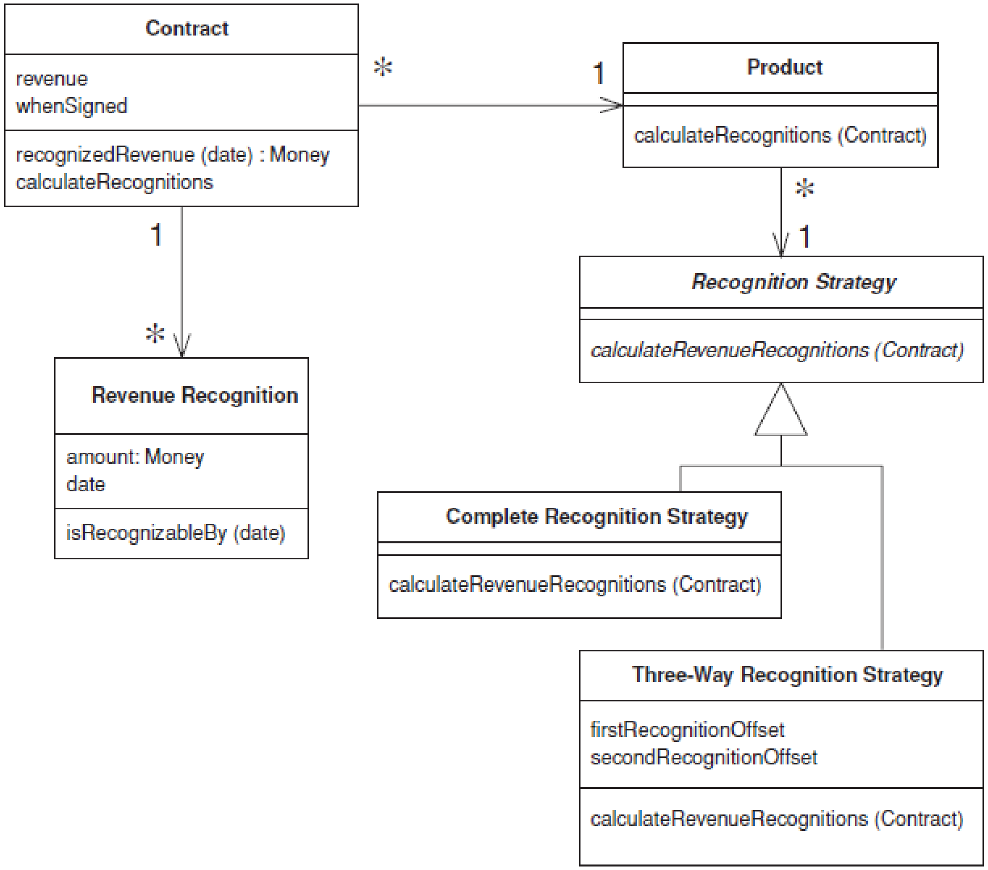


Every class contains both behavior and data.

```
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);
}
```

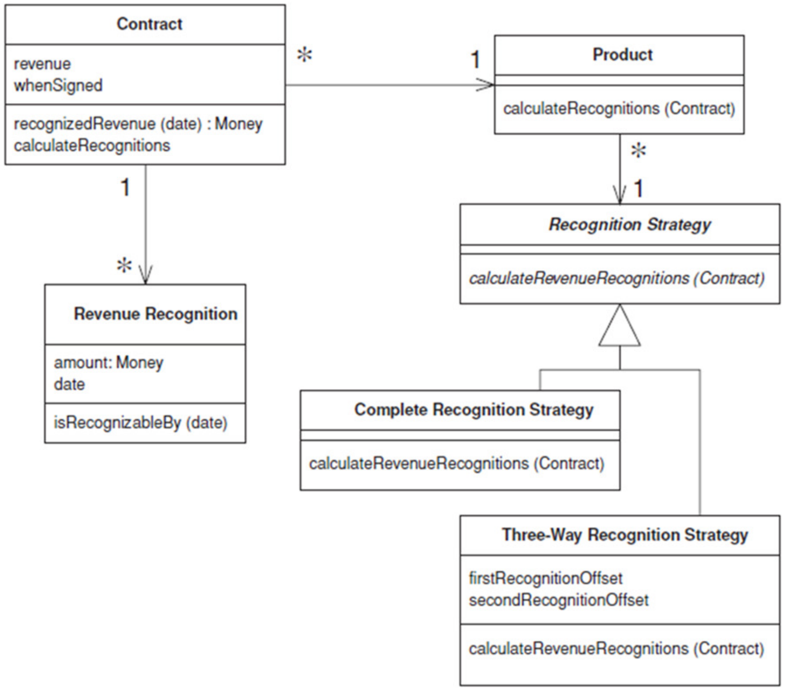
Calculating how much revenue is recognized on a particular date involves both the contract and revenue recognition classes.

```
class Contract...
private List revenueRecognitions = new ArrayList();
public Money recognizedRevenue(MfDate asOf)
{
          Money result = Money.dollars(0);
          Iterator it = revenueRecognitions.iterator();
          while (it.hasNext())
          { RevenueRecognition r = (RevenueRecognition) it.next();
          if (r.isRecognizableBy(asOf))
          result = result.add(r.getAmount());
          }
return result;
}
```

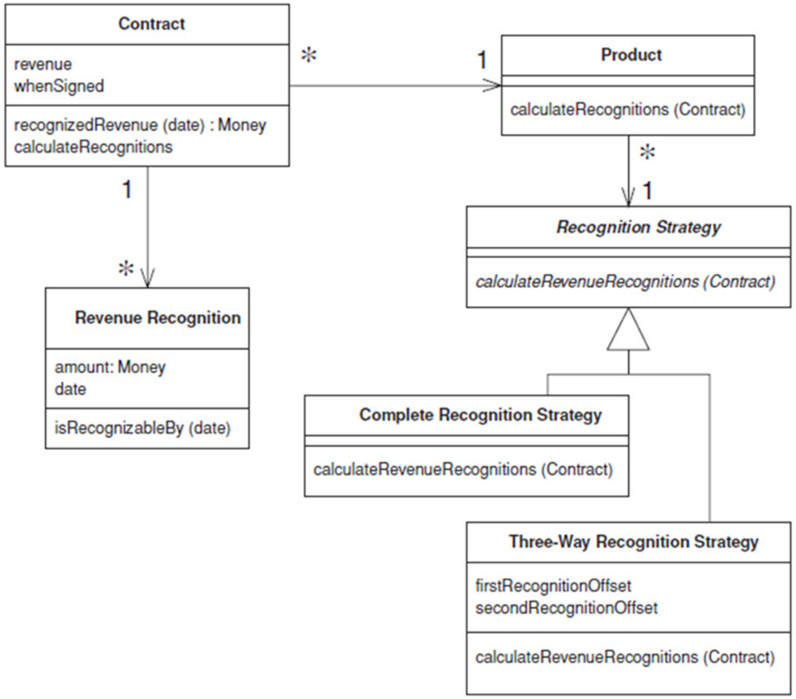

Contract
revenue
whenSigned
recognizedRevenue (date) : Money
calculateRecognitions
*
1
Product
calculateRecognitions (Contract)
1
*
*
1
Revenue Recognition
amount: Money
date
isRecognizableBy (date)
Recognition Strategy
calculateRevenueRecognitions (Contract)
Complete Recognition Strategy
calculateRevenueRecognitions (Contract)
Three-Way Recognition Strategy
firstRecognitionOffset
secondRecognitionOffset
calculateRevenueRecognitions (Contract)

Class diagram of the example classes for a Domain Model.

```
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;
}
```


Contract
revenue
whenSigned
recognizedRevenue (date) : Money
calculateRecognitions
*
1
Product
calculateRecognitions (Contract)
*
1
Recognition Strategy
calculateRevenueRecognitions (Contract)
1
*
Revenue Recognition
amount: Money
date
isRecognizableBy (date)
Complete Recognition Strategy
calculateRevenueRecognitions (Contract)
Three-Way Recognition Strategy
firstRecognitionOffset
secondRecognitionOffset
calculateRevenueRecognitions (Contract)

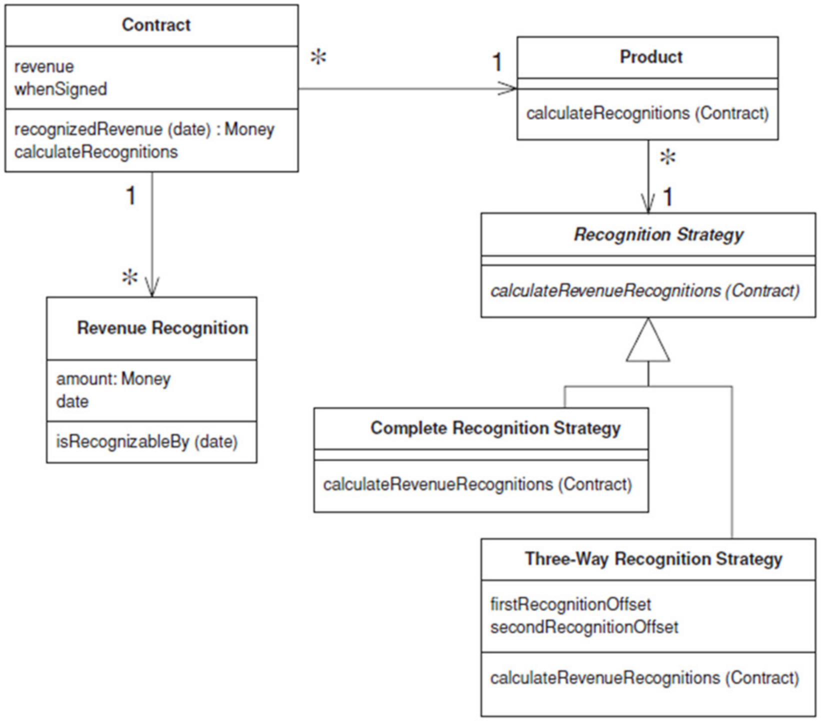
Imagine a company that sells three kinds of products: word processors, databases, and spreadsheets. According to the rules, when you sign a contract for a word processor you can book all the revenue right away. If it is a spreadsheet, you can book one-third today, one-third in sixty days, and one-third 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.


Contract
revenue
whenSigned
recognizedRevenue (date) : Money
calculateRecognitions
*
1
Product
calculateRecognitions (Contract)
*
1
Recognition Strategy
calculateRevenueRecognitions (Contract)
1
*
Revenue Recognition
amount: Money
date
isRecognizableBy (date)
Complete Recognition Strategy
calculateRevenueRecognitions (Contract)
Three-Way Recognition Strategy
firstRecognitionOffset
secondRecognitionOffset
calculateRevenueRecognitions (Contract)

```
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)));
}
```



Contract

revenue
whenSigned

recognizedRevenue (date) : Money
calculateRecognitions

Product

calculateRecognitions (Contract)

Revenue Recognition

amount: Money
date

isRecognizableBy (date)

Recognition Strategy

calculateRevenueRecognitions (Contract)

Complete Recognition Strategy

calculateRevenueRecognitions (Contract)

Three-Way Recognition Strategy

firstRecognitionOffset
secondRecognitionOffset

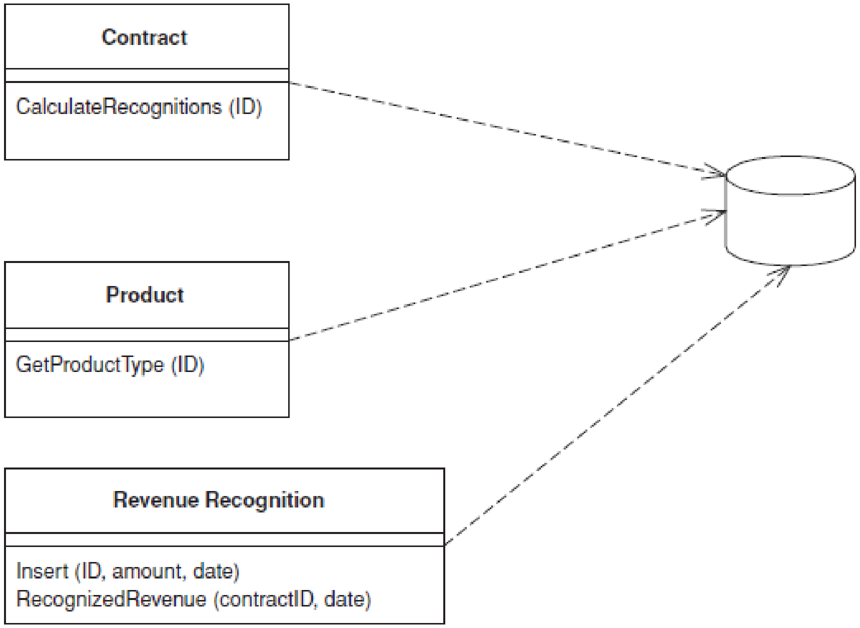
calculateRevenueRecognitions (Contract)

Table Module

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

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 is that, if you have many orders, a Domain Model will have one order object per order while a Table Module will have one object to handle all orders.



Contract

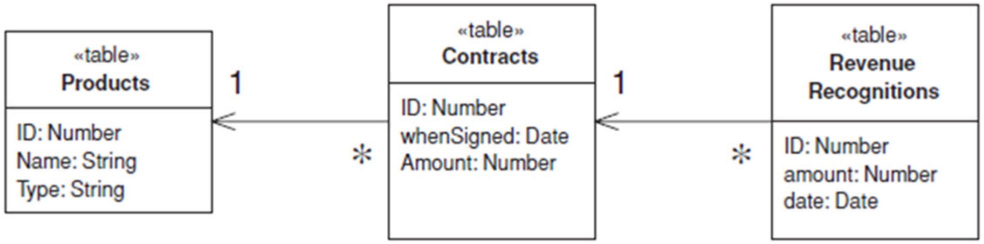
CalculateRecognitions (ID)

Product

GetProductType (ID)

Revenue Recognition

Insert (ID, amount, date)
RecognizedRevenue (contractID, date)



«table»
Products

ID: Number
Name: String
Type: String

«table»
Contracts

ID: Number
whenSigned: Date
Amount: Number

«table»
Revenue Recognitions

ID: Number
amount: Number
date: Date

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** .

However, *Table Module* does not give you the full power of objects in organizing complex logic. You can not have direct instance-to-instance relationships, and polymorphism does not work well. So, for handling complicated domain logic, a *Domain Model* is a better choice.

Essentially you have to trade off *Domain Model* 's ability to handle complex logic against *Table Module*'s easier integration with the underlying table-oriented data structures.

**Record Set* *is a*n in-memory representation of tabular data.