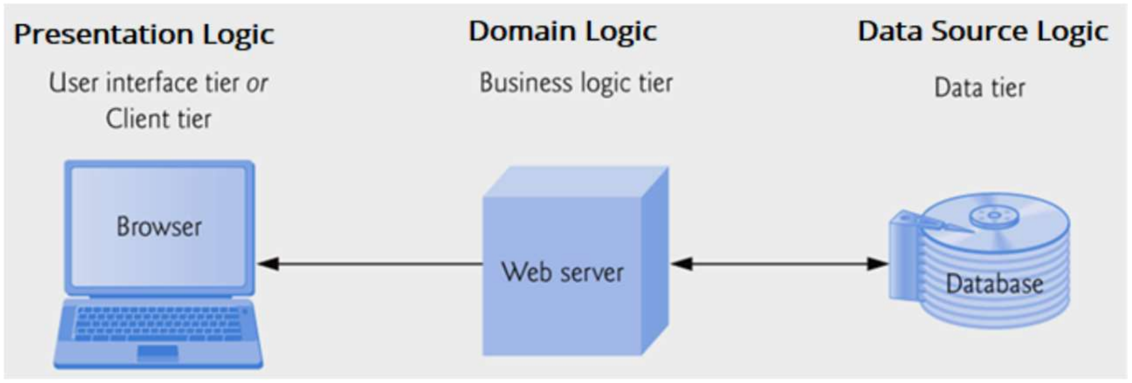
SOEN 387: Web-Based Enterprise Application Design

Chapter 2. Organizing Domain Logic

Domain can be separated it into three primary patterns:

- Transaction Script
- Domain Model
- Table Module



Transaction Script

- The simplest approach to storing domain logic is the *Transaction Script*.
- A *Transaction Script* is essentially a procedure that takes the input from the presentation, processes it with validations and calculations, stores data in the database, and invokes any operations from other systems.
- It then replies with more data to the presentation, perhaps doing more calculation to help organize and format the reply.
- The fundamental organization is of a single procedure for each action that a user might want to do. Hence, we can think of this pattern as being a script for an action, or business transaction. Pieces get separated into subroutines, and these subroutines can be shared between different *Transaction Scripts*.
- However, the driving force is still that of a procedure for each action, so a retailing system might have *Transaction Scripts* for checkout, for adding something to the shopping cart, for displaying delivery status, and so on.

A Transaction Script offers several advantages:

It's a simple procedural model that most developers understand.

It works well with a simple data source layer using *Row Data Gateway* or *Table Data Gateway*.

It is obvious how to set the transaction boundaries: Start with opening a transaction and end with closing it.

Transaction Script has also plenty of disadvantages, which tend to appear as the complexity of the domain logic increases.

- Often there will be duplicated code as several transactions need to do similar things. Some of this can be dealt with by factoring out common subroutines, but even so much of the duplication is tricky to remove and harder to spot.
- The resulting application can end up being quite a tangled web of routines without a clear structure.

Domain Model

- Complex logic is where objects come in, and the object-oriented way to handle this problem is with a *Domain Model* .
- With a *Domain Model* we build a model of our domain which is organized primarily around the nouns in the domain. Thus, a leasing system would have classes for lease, asset, and so forth.
- The logic for handling validations and calculations would be placed into this domain model, so shipment object might contain the logic to calculate the shipping charge for a delivery.

Domain Logic versus Transaction Logic

Using a *Domain Model* as opposed to a *Transaction Script* is the essence of the paradigm shift that object-oriented people talk about so much. Rather than one routine having all the logic for a user action, each object takes a part of the logic that's relevant to it.

Table Module

- A *Table Module* is in many ways a middle ground between a *Transaction Script* and a *Domain Model* .
- A *Table Module* organizes domain logic with one class per table in the database.
- The primary distinction with *Domain Model* is that a *Domain Model* will have one order object per order while a *Table Module* will have one object to handle all orders.
- One of the problems with *Domain Model* is the interface with relational databases, often need complex code to pull data in and out of the database.

Service Layer

A common approach in handling domain logic is to split the domain layer in two. A Service Layer is placed over an underlying *Domain Model* or *Table Module* .Usually you only get this with a *Domain Model* or *Table Module.*

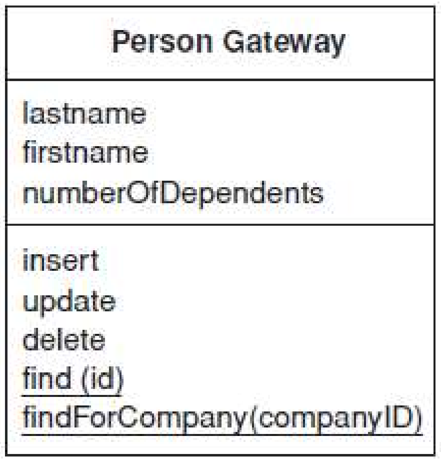
Domain layer that uses only *Transaction Script* is not complex enough to warrant a separate layer.

The presentation logic interacts with the domain purely through the Service Layer , which acts as an API for the application.

Chapter 3. Mapping to Relational Databases

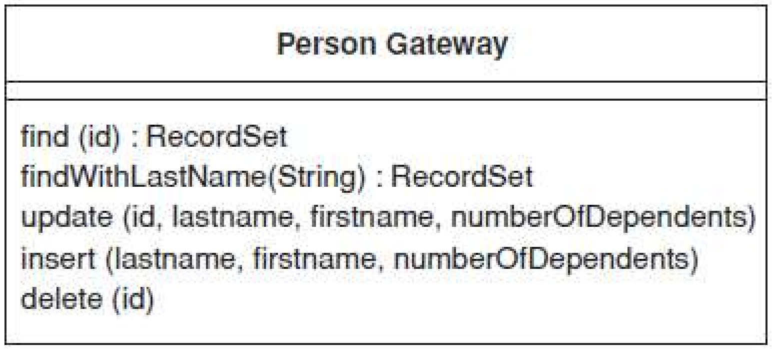
- Many application developers don't understand SQL well and, as a result, have problems defining effective queries and commands.
- It is wise to separate SQL access from the domain logic and place it in separate classes. A good way of organizing these classes is to base them on the table structure of the database so that you have one class per database table.
- These classes then form a *Gateway* to the table.
- The rest of the application needs to know nothing about SQL
- The most obvious way to use the Gateway is to have an instance of it for each row that's returned by a query
- The Row Data Gateway will do any type conversion from the data

source types to the in-memory types.



A Row Data Gateway
has one instance per row returned by a query

- Many environments provide a Record Set that is, a generic data structure of tables and rows that mimics the tabular nature of a database.
- Because a *Record Set* is a generic data structure, environments can use it in many parts of an application.
- If you use a *Record Set* , you only need a single class for each table in the database. This Table Data Gateway provides methods to query the database that return a Record Set.


Person Gateway
find (id) : RecordSet
findWithLastName(String) : RecordSet
update (id, lastname, firstname, numberOfDependents)
insert (lastname, firstname, numberOfDependents)
delete (id)

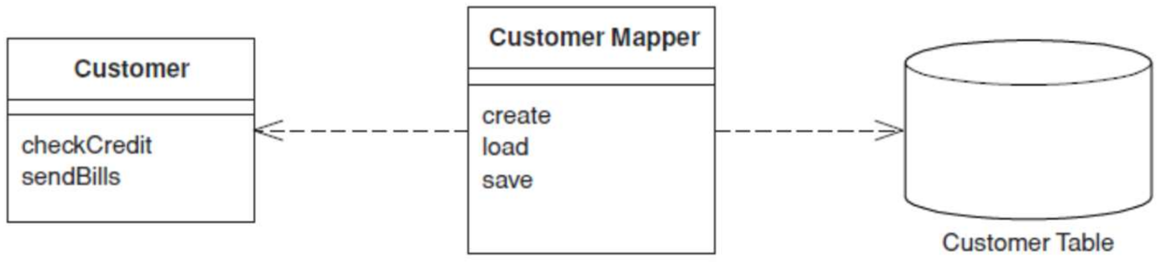
A Table Data Gateway has one instance per table.

- The fact that *Table Data Gateway* fits very nicely with *Record Set* makes it the obvious choice if you are using Table Module .

- It’s also a pattern you can use to think about organizing stored procedures. Many designers like to do all of their database access through stored procedures rather than through explicit SQL. In this case you can think of the collection of stored procedures as defining a *Table Data Gateway* for a table.

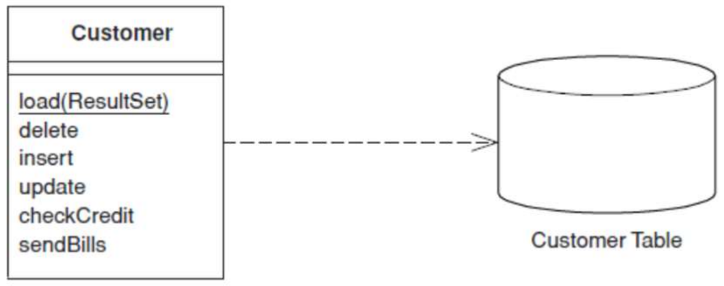
Good Practice

- A good practice is to isolate the *Domain Model* from the database completely, by making your indirection layer entirely responsible for the mapping between domain objects and database tables.
- This *Data Mapper* (165) (see Figure) handles all of the loading and storing between the database and the *Domain Model* (116) and allows both to vary independently.
- It's the most complicated of the database mapping architectures, but its benefit is complete isolation of the two layers.



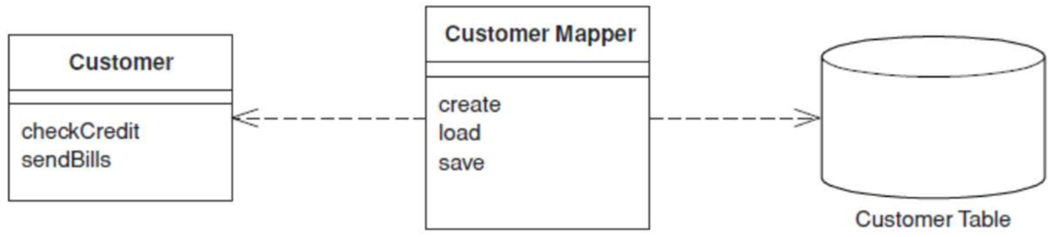
A Data Mapper *insulates the domain objects and the database from each other.*

If the domain logic is simple and you have a close correspondence between classes and tables, *Active Record* is the simple way to go.



In the Active Record *a customer domain object knows how to interact with database tables.*

If you have something more complicated, *Data Mapper* is what you need.



A Data Mapper *insulates the domain objects and the database from each other.*

The Behavioral Problem

That behavioral problem is how to get the various objects to load and save themselves to the database.

- If you load a bunch of objects into memory and modify them, you have to keep track of which ones you have modified and make sure to write all of them back out to the database.
- As you read objects and modify them, you have to ensure that the database state you are working with stays consistent. If you read some objects, it's important to ensure that the reading is isolated so that no other process changes any of the objects you have read while you are working on them. Otherwise, you could have inconsistent and invalid data in your objects.

A pattern to solving both of these problems is *Unit of Work*

- A *Unit of Work* keeps track of all objects read from the database, together with all objects modified in any way.
- It also handles how updates are made to the database. Instead of the application programmer invoking explicit save methods, the programmer tells the unit of work to commit. That unit of work then sequences all of the appropriate behavior to the database, putting all of the complex commit processing in one place.
- A good way of thinking about *Unit of Work* is as an object that acts as the controller of the database mapping.

- As you load objects, you have to be wary about loading the same one twice. If you do that, you'll have two in-memory objects that correspond to a single database row.
- To deal with this you need to keep a record of every row you read in an *Identity Map* .
- Each time you read in some data, you check the *Identity Map* first to make sure that you don not already have it. If the data is already loaded, you can return a second reference to it. That way any updates will be properly coordinated.
- As a benefit you may also be able to avoid a database call since the *Identity Map* also doubles as a cache for the database.