Software Architecture Design II
Enterprise Software Architecture

Matthew Dailey

Computer Science and Information Management
Asian Institute of Technology

Readings

Readings for these lecture notes:

- Fowler (2002), *Patterns of Enterprise Application Architecture*, Addison-Wesley.

Some material © Fowler (2002).

Outline

This semester, we study large-scale software architecture.

We begin with architectural patterns for a single enterprise application.

Then we'll move to larger distributed systems.

The focus will be on enterprise applications but most of the methods will apply to other application areas.

First we will:

- Discuss enterprise applications.
- Review their primary layers.
- Understand what a pattern is.
- Begin with patterns for the domain logic layer.

Enterprise applications (Fowler, 2002, p. xviii)

- Entail display, manipulation, and storage of large amounts of often complex data.
- Support or automate business processes using that data.

Example enterprise apps:

- Payroll
- Patient records
- Shipping tracking
- Cost analysis
- Credit scoring
- Insurance
- Supply chain management
- Accounting
- Customer service

Counterexamples:

- Fuel injection
- Word processors
- Elevator controllers
- Chemical plant controllers
- Telephone switches
- Operating systems
- Compilers
- Games

All enterprise applications share a few traits:

- Persistent data, lots of it, with complex, evolving data models
- Concurrent access to the persistent data
- Complex user interfaces
- A need to integrate with other enterprise applications
- Complex business logic, or "illogic" — business rules that come from special cases for specific customers.

The nature of enterprise applications constrains our architectural choices:

- Most enterprises need many small apps and a few large apps. Architectural integration is important.
- Due to similarities between applications, many powerful architectural patterns have emerged.
- Due to diversity between applications, in practice, we have to heavily customize the patterns we apply.

Layering is one of the most common architectural structures:

- Higher layers have dependencies on the lower layers
- Lower layers do not depend on layers above

By the way, don't get confused about layers vs. tiers. "Tier" usually implies a physical separation, whereas a layer is merely logical.

[Q: Why is layering such a great idea?]

[Q: Are there any disadvantages of layering (hint: think TCP/IP vs. OSI)]

The 3-layer architecture has become the de-facto organization for enterprise systems.

- In the early days, applications were just programs that manipulated files.
- The next step was to separate the database from the interface and application logic. This is the 2-layer client-server architecture. It gets unmanageable when the business logic gets complex.
- OO folks eventually figured out that a 3-layer approach, separating the interface (presentation layer), the application logic (domain layer), and the database management system (data source layer) was better.
- The early client-server tools did not support 3-layered architectures very well, but the Web forced the business logic out of the client and onto the server, making it easy.

The presentation layer:

- Handles interaction between the user and the software.
- Is usually an HTML interface or rich-client GUI.

The data source layer:

- Communicates with other systems to get and send data.
- Normally includes a database management system to store persistent data.

The domain logic or business logic layer:

- Does calculations, validation, and dispatch.
- Might encapsulate the data source but usually doesn't.

Generally, in a large application, the layers will be separate packages (draw the package diagram).

The dependencies are important: the domain and data source layers, in particular, should not know about the presentation layer.

The physical distribution of the layers depends on the client style:

- For Web interfaces, all layers normally run server-side.
  - For responsiveness, Ajax-style interfaces may be useful to move some presentation logic to the client.
  - Moving business logic into an Ajax client should be avoided!
- For rich client interfaces, the presentation layer normally runs client-side.
  - Business logic is best left on the server if possible.
  - If a lot of the business logic needs to be client side, it is best to push it all to the client.
  - In some cases, a small amount of business logic may run client side with the majority running server side.

Generally, Web interfaces are preferred, to eliminate client maintenance issues, unless there are compelling reasons for a rich client.

Fowler (2002) opens with a quote from Christopher Alexander:

> *Each pattern describes a problem which occurs over and over again in our environment, then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*

Alexander was actually a (building) architect, but the same idea is relevant in software system architecture. Patterns represent best practices we can apply to ensure we don't make the same mistakes the pioneers did.

In software architecture as in building architecture, patterns are references forming a common vocabulary for design.

Patterns are not recipes! You can never blindly apply them, since they are always incomplete to some degree.

Fowler's (2002) patterns share a common structure based on GoF:

- Each pattern has a carefully chosen name.
- Each pattern has an intent briefly summarizing it.
- Each pattern has a sketch representing it visually, in UML or otherwise.
- Each pattern has a motivating problem as an example of what kind of problems this pattern solves.
- Each pattern has a detailed how it works section including a discussion of implementation issues and variations on the theme.
- Each pattern has a when to use it section describing when it should be used. There will often be multiple patterns applicable in a particular situation.
- There may be further reading for more information.
- Example code gives the idea of how to apply the pattern in Java or C#. The code is not to be plugged in but adapted to the situation.

Outline

Domain logic patterns

Within the domain logic or business logic layer, how can we organize the logic?

There are three main patterns:

- Transaction Script
- Domain Model
- Table Module

We might also add a Service Layer on top of a Domain Model.

We consider the basic idea of each pattern, then the details.

Transaction Script *organizes business logic by procedures where each procedure handles a single request from the presentation.*

Transaction Scripts are the simplest way to organize domain logic. A Transaction Script:

- Takes input data from the presentation layer;
- Processes the data with validations and calculations;
- Updates the database;
- Invokes operations from other systems;
- Replies to the presentation layer with data to display.

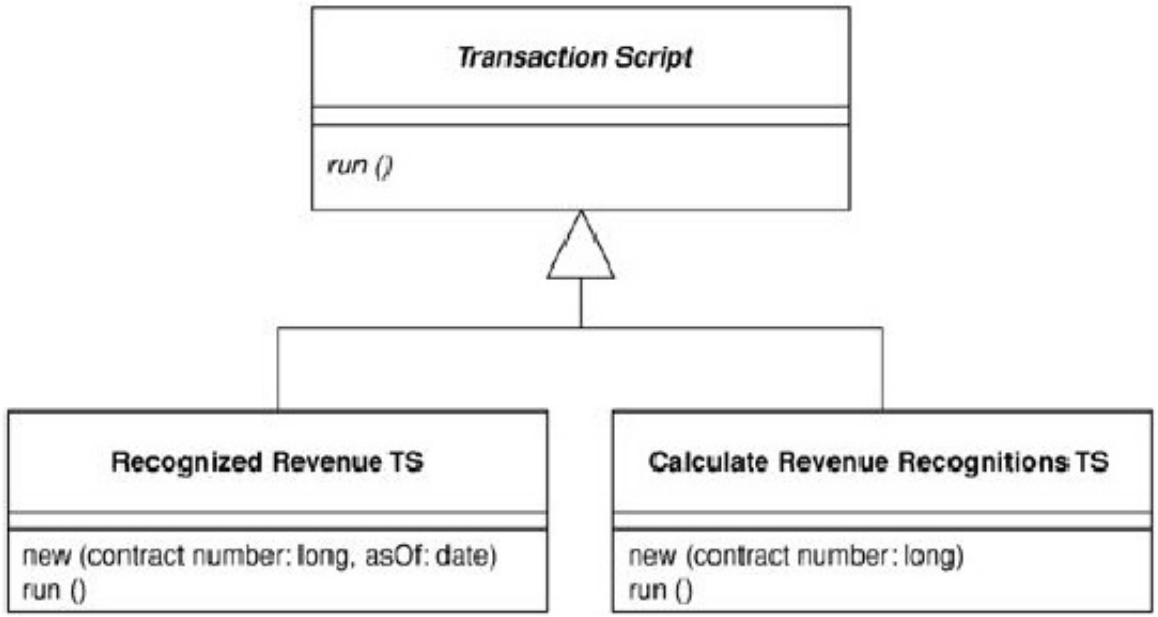
We write a single procedure for each use case or system operation or transaction.

Most business applications are just a series of database transactions.

Each transaction contains a bit of business logic, usually simple, such as selecting what information to display, validating input, doing calculations, and so on.

With Transaction Script, common functionality can be broken into subroutines, but each transaction gets its own function or method.

Scripts should be in classes separate from the presentation and data source, either one class per script or multiple scripts grouped functionally into classes.



Example of one class per script with GoF Command pattern (Fowler 2002, Fig. 9.1).

Use transaction scripts for simple problems that don't need fancy object models.

Advantages:

- Simplicity, with an easy-to-understand procedural model
- It works very well with simple data sources using a Row Data Gateway or a Table Data Gateway (see the database mapping patterns)
- The transaction boundaries are obvious (start a transaction at the beginning and commit the transaction at the end of each procedure).

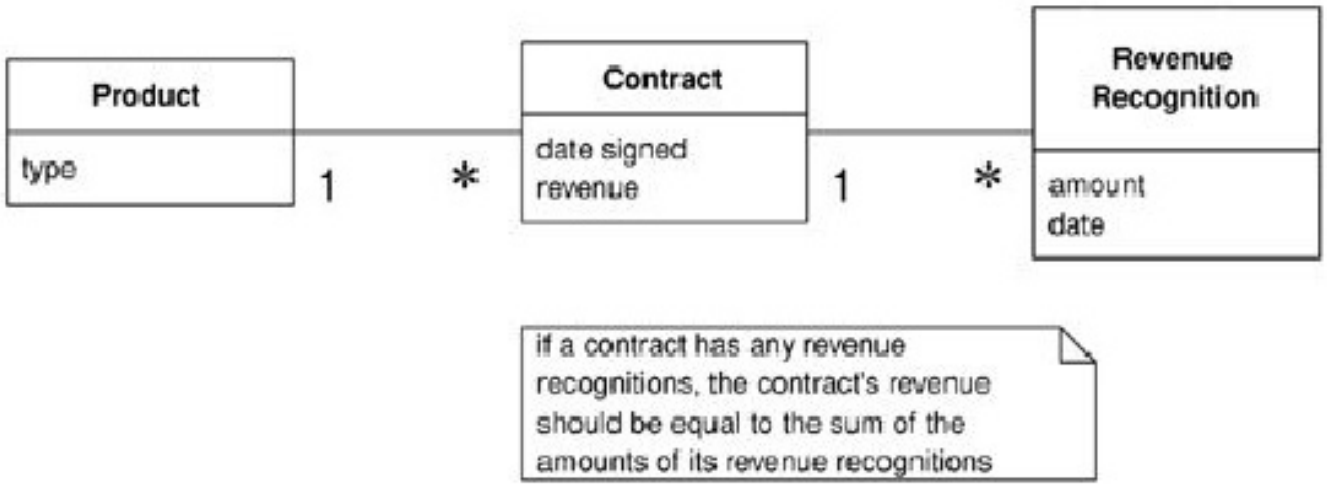
Disadvantages:

- Duplication across multiple actions eventually creates a tangle of routines with no structure.

Now we'll look at a transaction script implementation in Java for a simple example application.

When can a business book revenue? The rules are complex and depend on regulations, corporate policy, and many other factors.

One way to handle this is to model the RevenueRecogntions for every Contract we sign.



Conceptual model for revenue recognitions on a contract (Fowler 2002, Fig. 9.2)

Suppose we use the data model:

```
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))
```

Domain logic patterns

Transaction Script example code

Here's how we might map to the data source using the Table Data Gateway pattern to wrap SQL queries (more on data source mapping later):

```
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;
```

This uses MfDate (Martin Fowler date!) and JDBC prepared statements.

Transaction script to get the sum of recognitions up to some date for a contract:

```
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);
      }
   }
```

Some notes of interest in the previous TS:

- `db` is the database gateway object.
- `Money` is a PEAA Base Pattern.
- We see iteration over a JDBC ResultSet and `getBigDecimal()` being used to extract the SQL `decimal field` as an arbitrary-precision decimal number.
- This script could be trivially implemented with an SQL aggregate procedure — the point is to understand how it could be done in a TS.

Domain logic patterns

Transaction Script example code

Next, `calcuateRevenueRecognitions` is a TS that adds the appropriate revenue recognitions to the database based on the product type.

```
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));
```

Domain logic patterns

Transaction Script example code

```
        }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);
}
```

Some notes of interest:

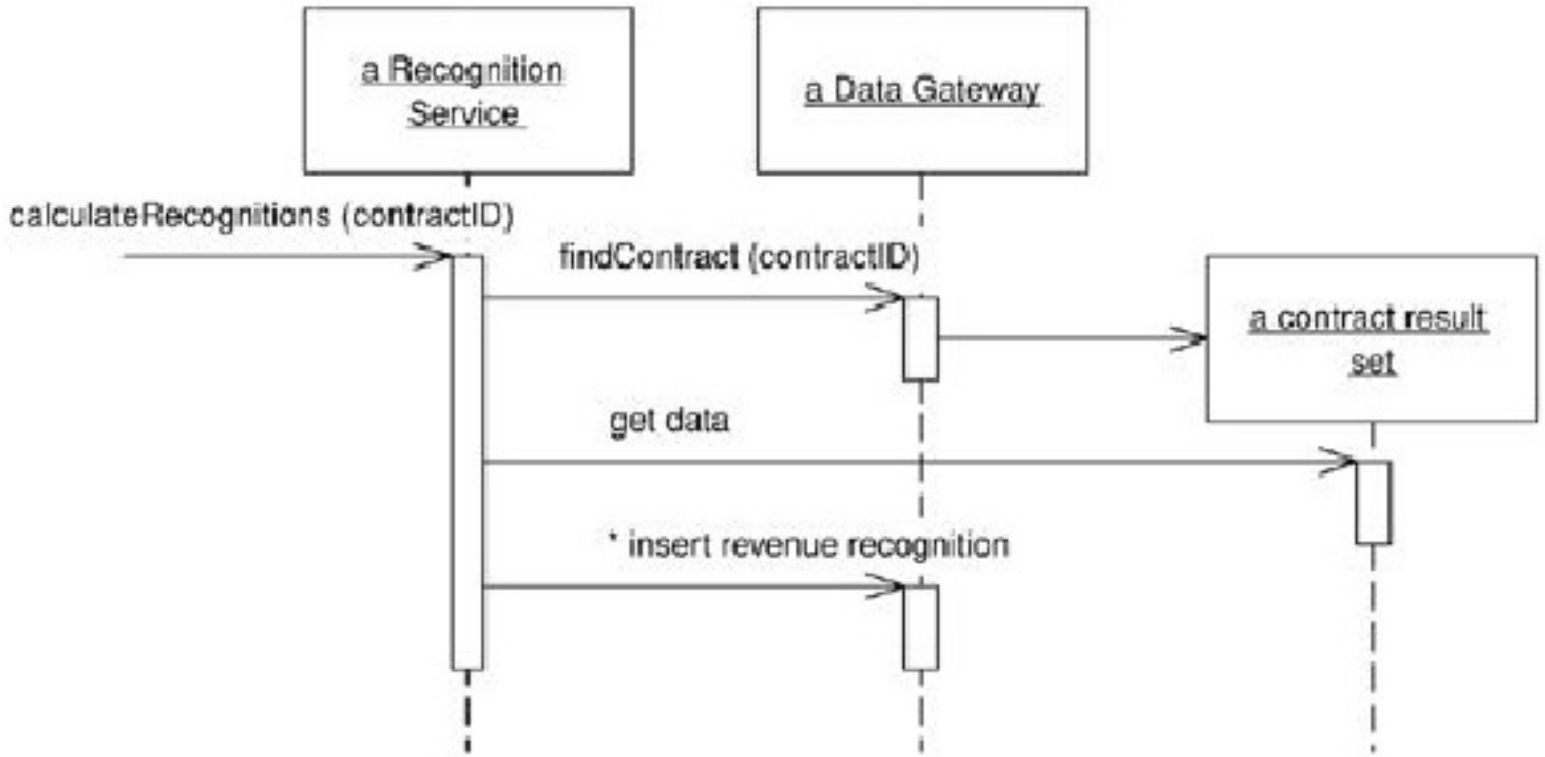
- allocate() is a Money method that doesn't lose pennies.
- The TS needs to know about all the possible types of products (Word processors, Spreadsheets, and Databases).

The gateway needs appropriate finders and inserters:

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

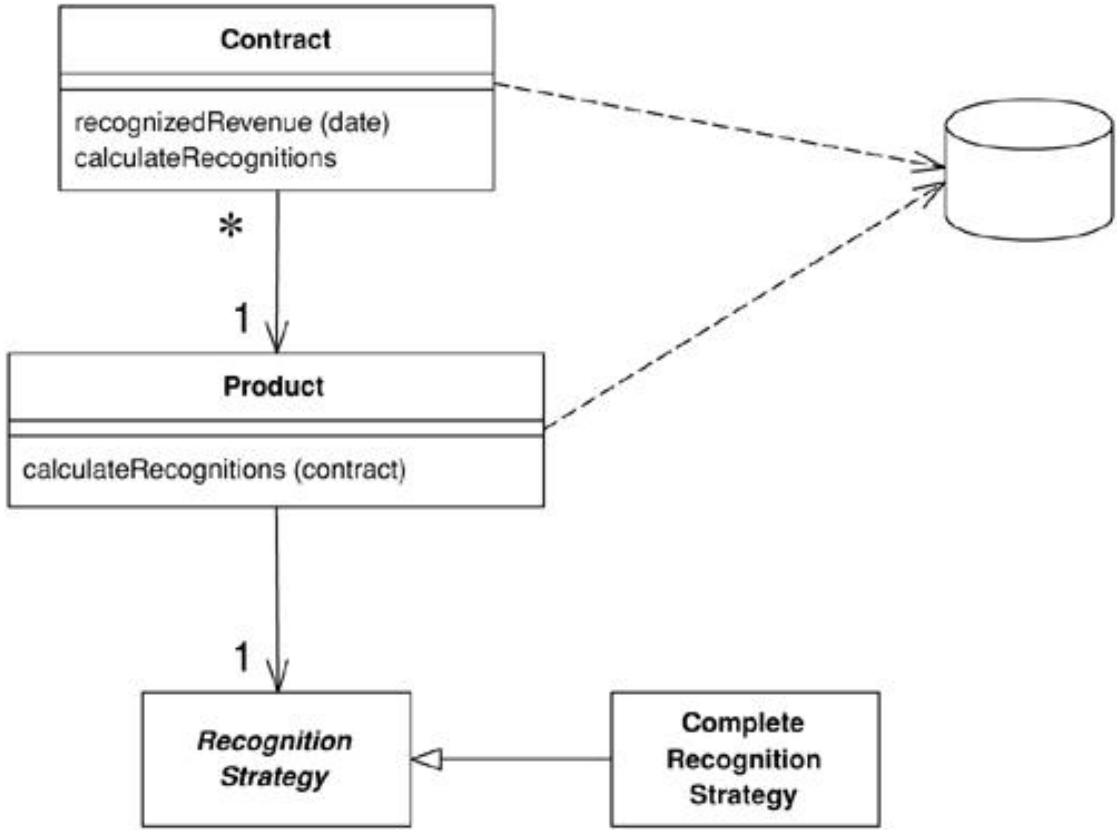
Domain logic patterns

Transaction Script interaction



Fowler (2002), Fig. 2.1

A Domain Model is an object model of the domain that incorporates both behavior and data.



Fowler (2002), p. 116

Domain logic patterns

Domain Model

We use OOAD to build a model of the domain, organized around the nouns in the domain.

Domain objects normally encapsulate both data and behavior, and come in two main kinds:

- Objects mimicking the data the business deals with
- Objects encapsulating the rules running the business.

Domain logic patterns

Domain Model

There tend to be two kinds of Domain Model:

- Simple models look like the database design, with one domain object for each database table, and can use the straightforward Active Record database mapping pattern.
- More complex models are better for complex logic but are harder to map, requiring the Data Mapper pattern.

Notes of interest for more complex domain models:

- Break up the logic and put it in the classes it naturally belongs to.
- In Java EE, we use POJOs (Plain Old Java Objects) for the domain model, and provide appropriate annotations to have them treated as persistent entities.
- Read GoF and Fowler's *Analysis Patterns* for useful patterns for organizing the domain classes.
- Use Data Mapper for the data source mapping and (maybe) implement a Service Layer interface to the presentation layer.

Advantages:

- The logic is organized, with each type of object handling its own validation, calculations, and so on.
- Domain models can scale well as the data source gets more complex.

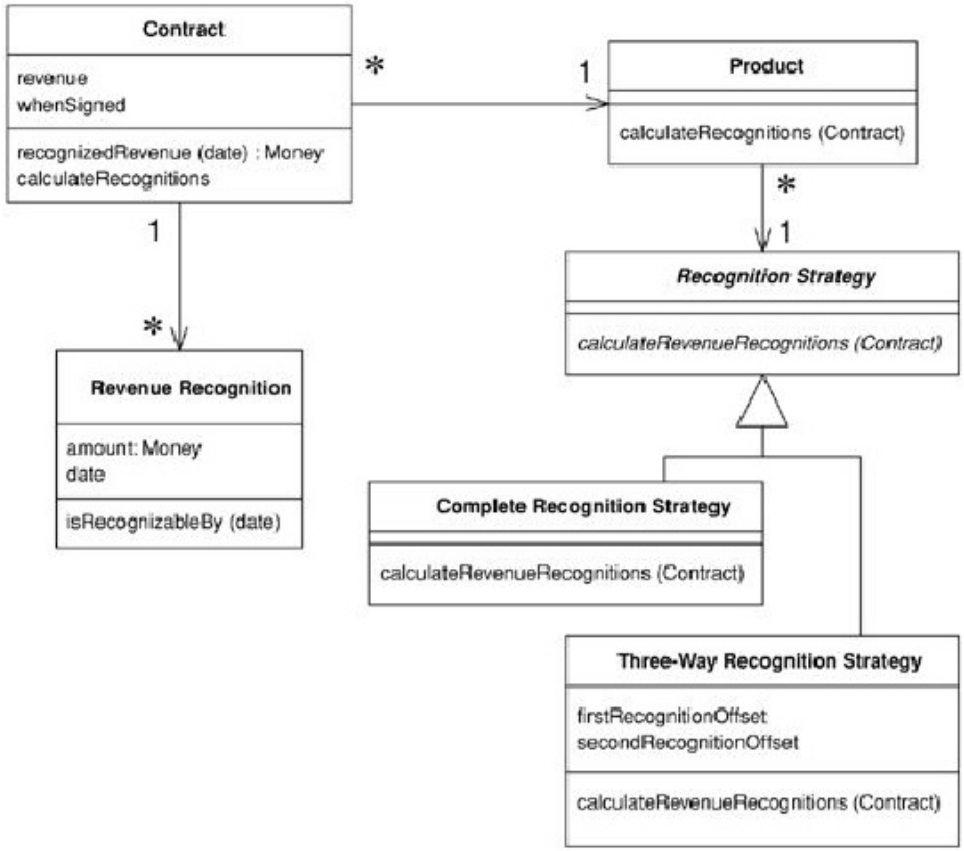
Disadvantages:

- It can be difficult to trace the behavior for one action
- Transaction boundaries might be more difficult to determine. You wouldn't want one domain object starting a transaction and another committing it.
- According to Fowler, it takes developers a long time to get used to the approach.

Domain logic patterns

Domain Model example code

We use the same revenue recognition system even though the logic is not really complex enough to merit a Domain Model.



Domain Model for revenue recognition using GoF Strategy pattern (Fowler, 2002, Fig. 9.3)

Domain logic patterns

Domain Model example code

```
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);
   }
```

Domain logic patterns

Domain Model example code

Now, calculating the revenue recognized on a particular date involves Contract and RevenueRecognition, by iterating over the current revenueRecognitions.

```
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;
   }
```

The complexity is higher than the TS because multiple objects are involved. But associating behavior only with the objects that "need to know" scales well and reduces coupling.

Domain logic patterns

Domain Model example code

```
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;
   }
```

Domain logic patterns

Domain Model example code

```
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));
   }
```

Domain logic patterns

Domain Model example code

```
class RecognitionStrategy...
   abstract void calculateRevenueRecognitions(Contract contract);

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

Domain logic patterns

Domain Model example code

```
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)));
}
```

Domain logic patterns

Domain Model example code

Here is some test code demonstrating use of the model:

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

When we calculate recognitions on a contract, Contract doesn't need to know about the strategy subclasses.

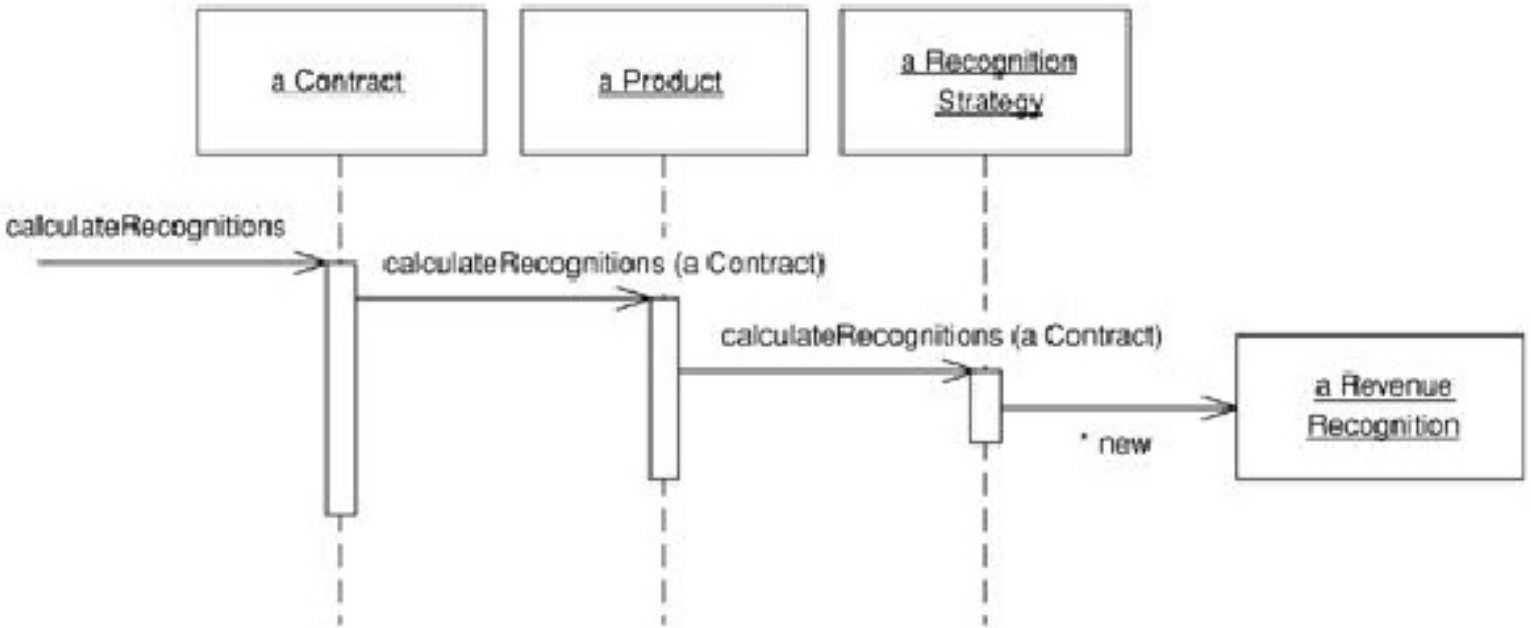
```
class Contract...
   public void calculateRecognitions() {
      product.calculateRevenueRecognitions(this);
   }

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

Some points about the example:

- We see how a request gets forwarded from place to place until we reach an object qualified to handle it. This is typical for domain models (and in all OO systems).
- Strategy is effective for refactoring procedural code full of conditionals.

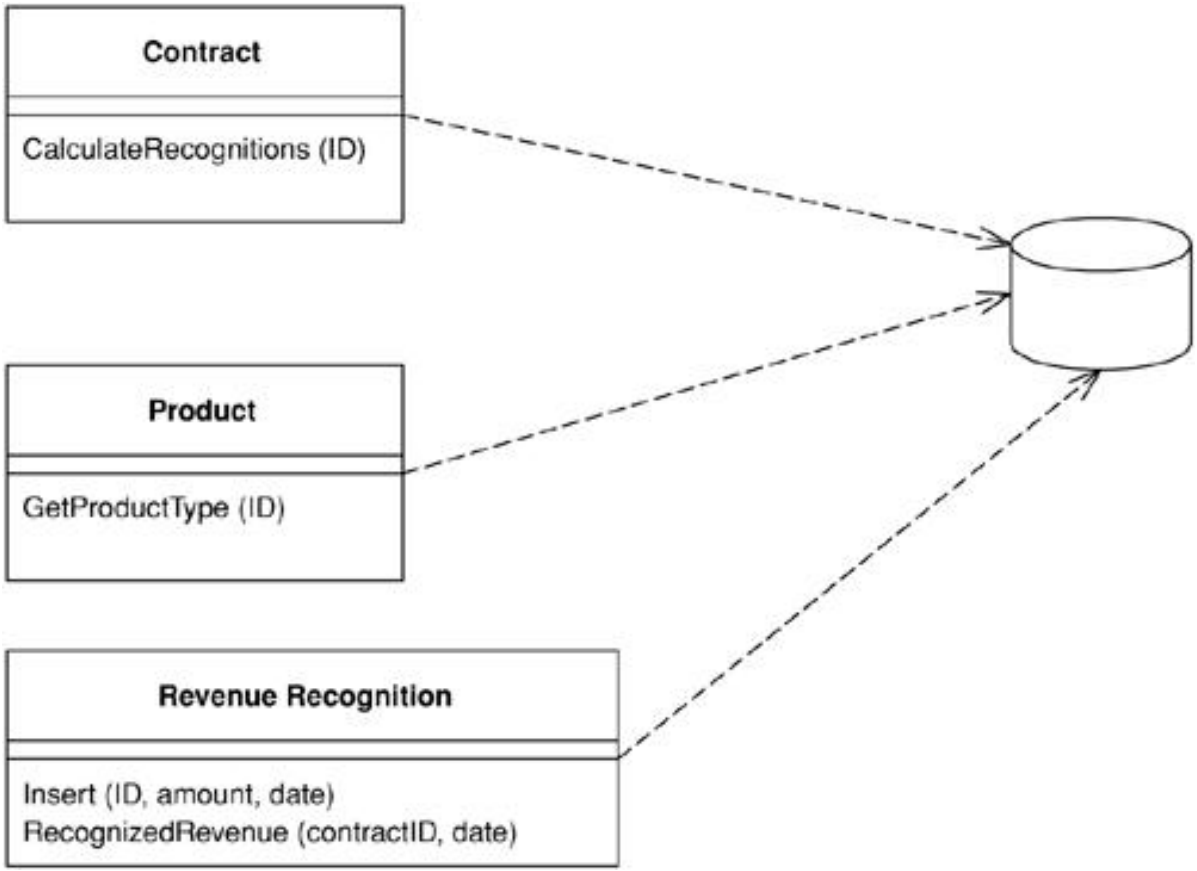
Domain logic patterns

Domain model



Fowler (2002), Fig. 2.2

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



Fowler (2002), p. 125

Table modules are similar to domain classes in a domain model, but are more closely tied to the database structure.

We have a one-to-one mapping between database tables and classes in the domain logic layer, and clients only use one instance of the class for each table.

This simplifies the data source mapping at the cost of a less flexible organization of the domain logic.

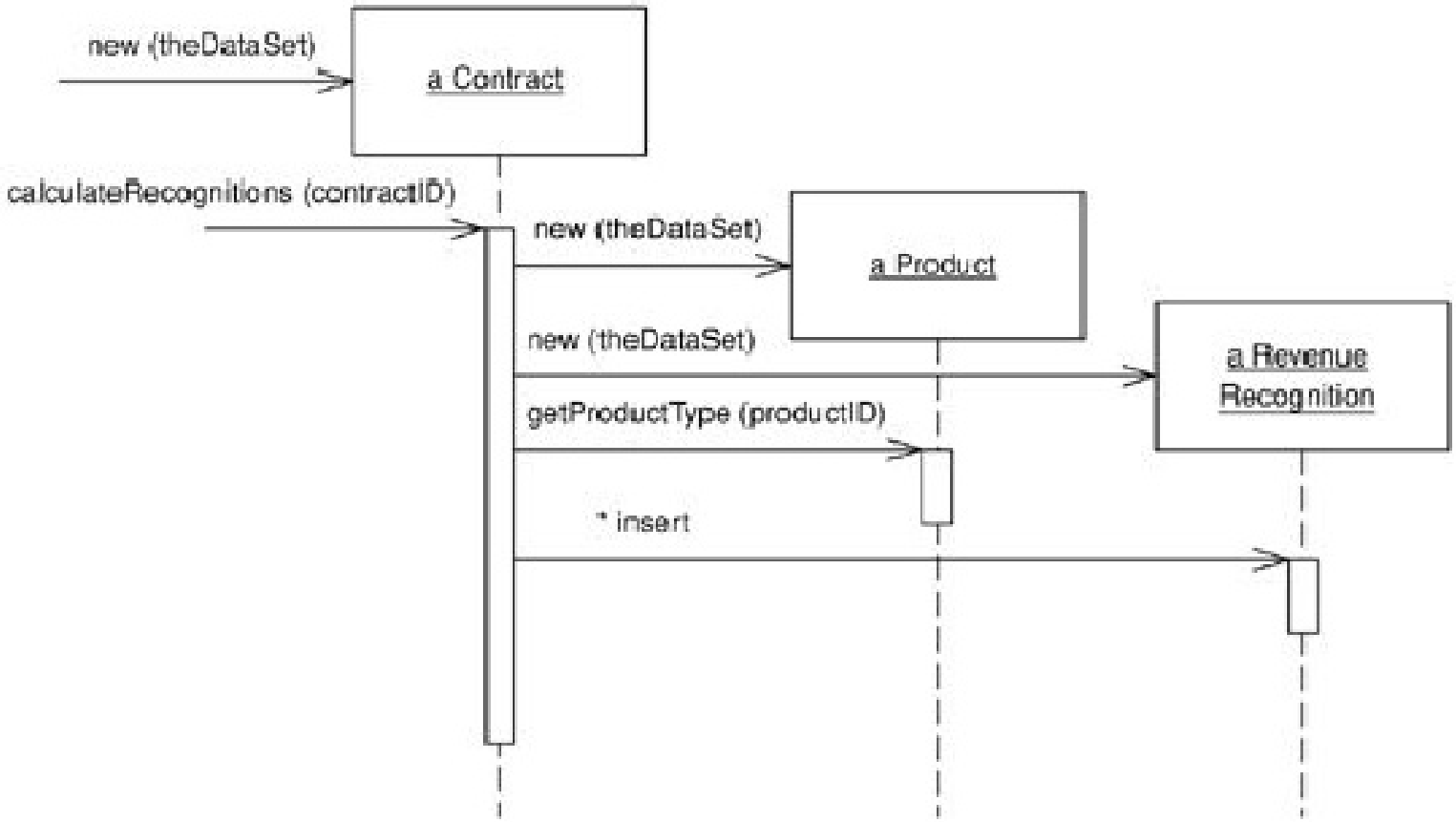
If we want to use data from a particular table as a client, we

- Issue a query to the database;
- Get the answer back as a Record Set;
- Use the record set to instantiate a table module object;
- Invoke operations on the table module object, passing an identifier when necessary to identify a particular row.

The table module might be a collection of static methods or an instance. Instances allow inheritance and initialization with a particular Record Set.

Domain logic patterns

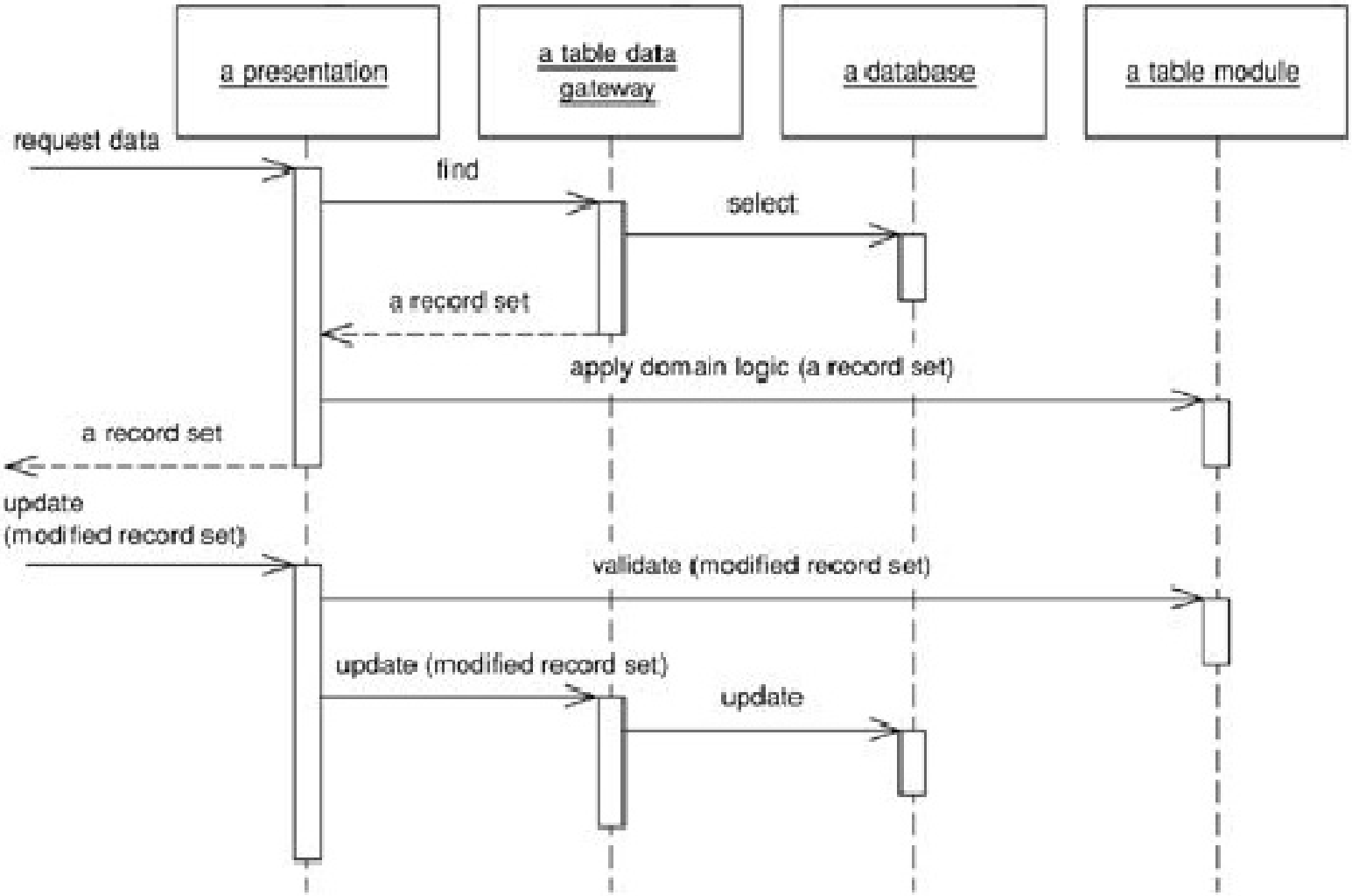
Table Module



Working with table modules instatiated from the same Result Set (Fowler, 2002, Fig. 9.4)

Domain logic patterns

Table Module



Layers of interaction with a Table Module (Fowler, 2002, Fig. 9.5)

Domain logic patterns

Table Module

Advantages:

- More structured than Transaction Scripts
- Easier to understand the flow than with a Domain Model
- Less complex data source mapping than a Domain Model
- GUI environments often provide direct support for displaying Record Sets, so Table Modules in the domain logic layer provide simple, direct mappings between the presentation and data source layers, with room for additional manipulation and validation of the data as it flows through. .NET works this way.

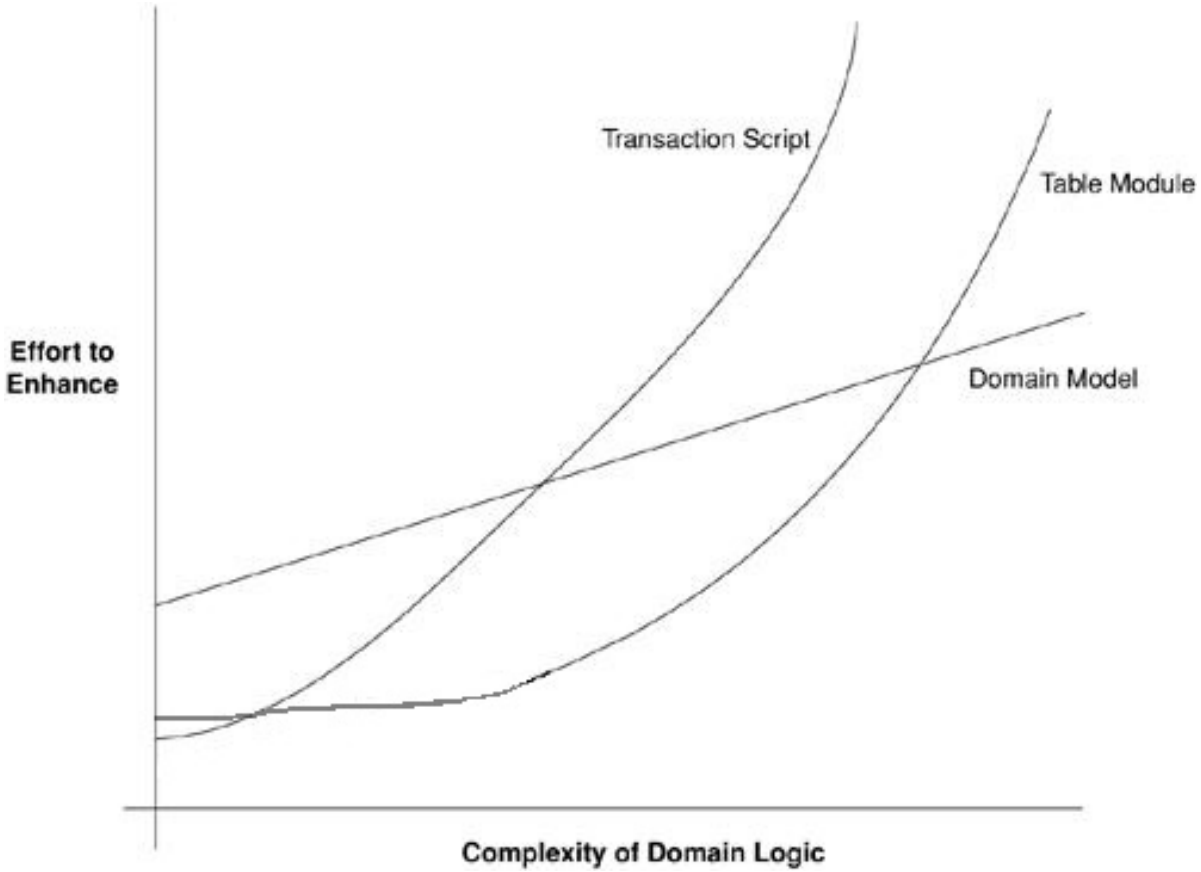
Disadvantages:

- For extremely simple applications, Transaction Scripts are easier to implement.
- For complex applications, Table Modules are not as flexible as a Domain Model

For a C# implementation of the revenue recognition example, see text.

Domain logic patterns

Your choice



Tradeoffs for domain logic styles (Fowler, 2002, Fig. 2.4)

Domain logic patterns

Your choice

The proper choice of domain logic organization depends on the complexity of the domain logic and the tool support available for a particular paradigm.

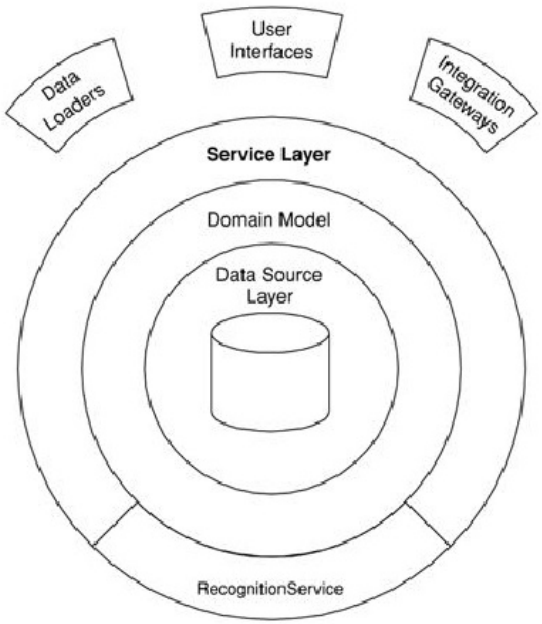
A mixture of the three patterns is often used in the same application.

Domain logic patterns

Service Layer

A Service Layer 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.

We provide an API decoupling presentation from the domain logic, making it easy to support multiple interfaces.



Fowler, 2002, p. 133

Besides decoupling, the service layer can be a good place to do things that cut across domain objects in use cases:

- Transaction control
- Security

Domain logic patterns

Rationale for Service Layer

Enterprise applications often have different interfaces to the same functionality, for example:

- A rich client GUI
- An integration gateway for other applications

In these cases we might break business logic into two parts:

- Domain logic, which is purely about the problem domain (e.g. revenue recognition strategies);
- Application logic or workflow logic.

A Service Layer might factor some or all application logic into the service layer, making the domain logic layer more reusable across different applications.

How much business logic to put in the service layer?

- At one extreme, we have transaction scripts with a simple Active Record domain model.
- At the other extreme, the service layer is a facade.
- More commonly, application logic is implemented by operation scripts in the service layer and domain logic is implemented in the domain objects.

We must group the external operations into service layer classes themselves called services, usually ending with the name Service.

Keep services coarse grained, minimizing the number of calls.

This will make it possible to allow remote access later.

To find services, start with the use case model and/or user interface specification. Usually there is a 1-to-1 correspondence between CRUD use cases and Service Layer operations.

For Java implementation, Fowler recommends EJB stateless session beans to implement application logic in operation scripts in the service layer, which then delegate to POJO domain objects for the domain logic.

Don't use a service layer if you think your business logic will only have one client (the UI) forever, but if you might have more than one client, a service layer is worthwhile.

To illustrate, we do a POJO Service Layer without transaction support.

To make it interesting, we add some application logic to the revenue recogntion application. When revenue recognitions are calculated, we need to

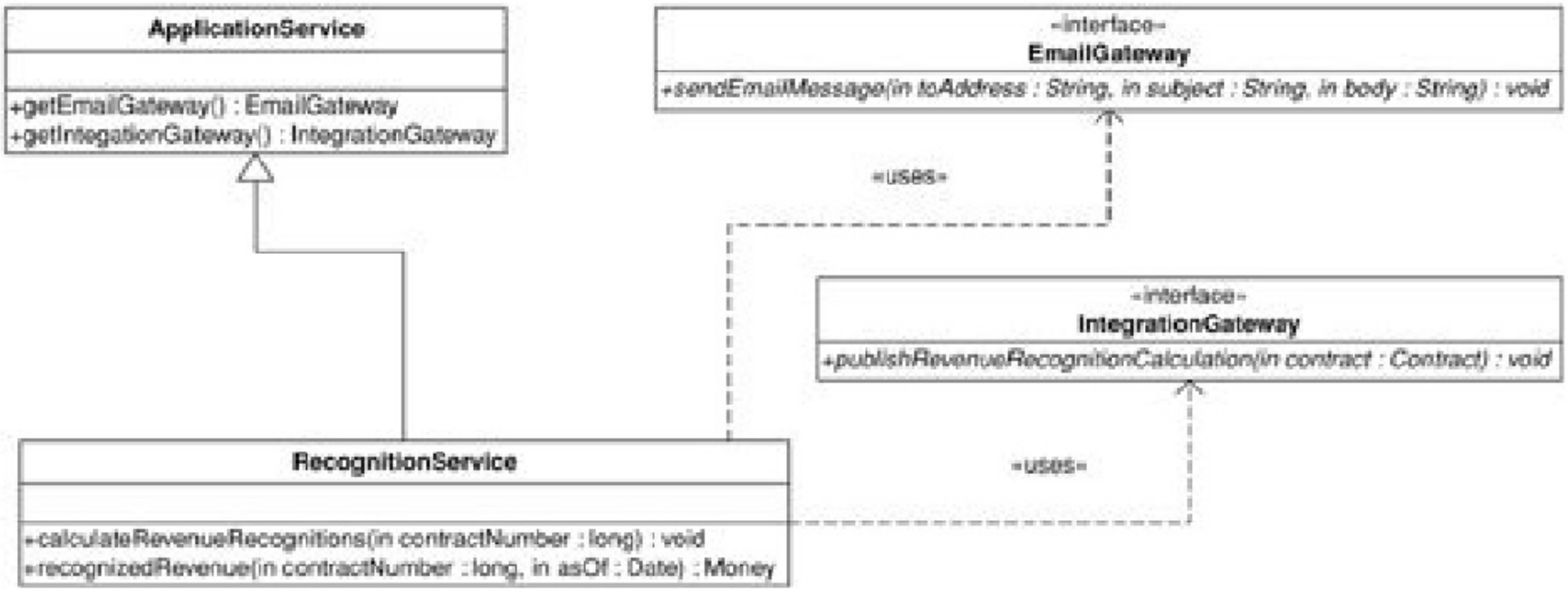
- Send an email to a contract administrator;
- Publish a message to any other integrated applications.

Now RecognitionService extends a Layer Supertype for our service layer and uses an EmailGateway and an IntegrationGateway.

(Layer Supertype is just a simple pattern for abstracting common functionality for a kind of object into a superclass.)

Domain logic patterns

Service Layer example code



POJO class diagram for a revenue recognition service (Fowler, 2002, Fig. 9.7)

Ignoring persistence, here is example code:

```
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);
}
```

Domain logic patterns

Service Layer example code

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

We're assuming that the contract class has methods to read contracts from the data source layer by ID.

We're also ignoring transactions for now — in fact, the `calculateRevenueRecognitions()` operation needs to be atomic.

EJB has built-in support for container-managed transactions. If stateless session beans are used, the transactional methods could be declared as so, with little change to the existing methods.

So we've seen the major architectural design choices for the business logic layer.

Thus far we've mostly ignored persistence mechanisms for the domain objects.

That's the next topic.

Outline

Mapping to relational databases

The data source layer needs to communicate with various pieces of infrastructure, usually including a database.

Object database technology is getting more mature:

- Open source frameworks like Zope, based on an object database, are increasing in popularity and market share
- They seem to save developer time compared to RDBMS mapping by up to 1/3

But most applications are still based on RDBMS technology.

- SQL is (almost) standard and well-understood by most developers
- RDBMSs have the backing of the big companies

Mapping to relational databases

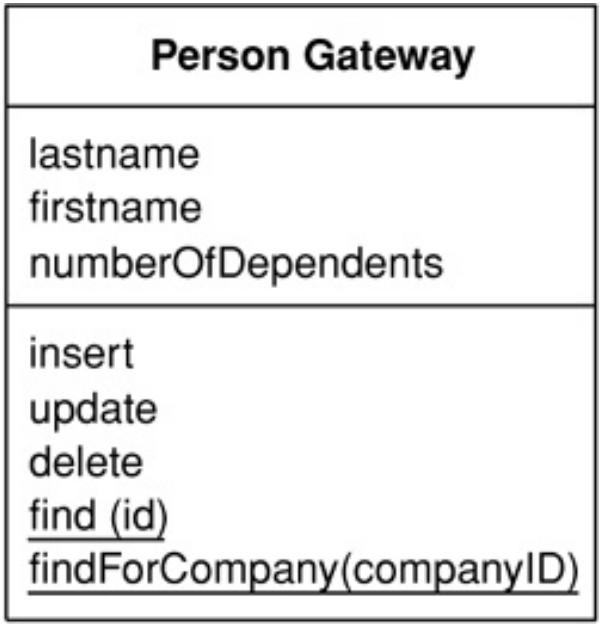
Basic choices

First, you should separate the SQL from the domain logic, putting it in separate classes.

Oftentimes we have one class per database table. The classes act as Gateways to the tables. We have two Gateway patterns:

- Row Data Gateway has one instance for each row returned by a query
- Table Data Gateway returns a Record Set (a generic data structure provided by the development environment, e.g. the JDBC `ResultSet`)

Example row data gateway:

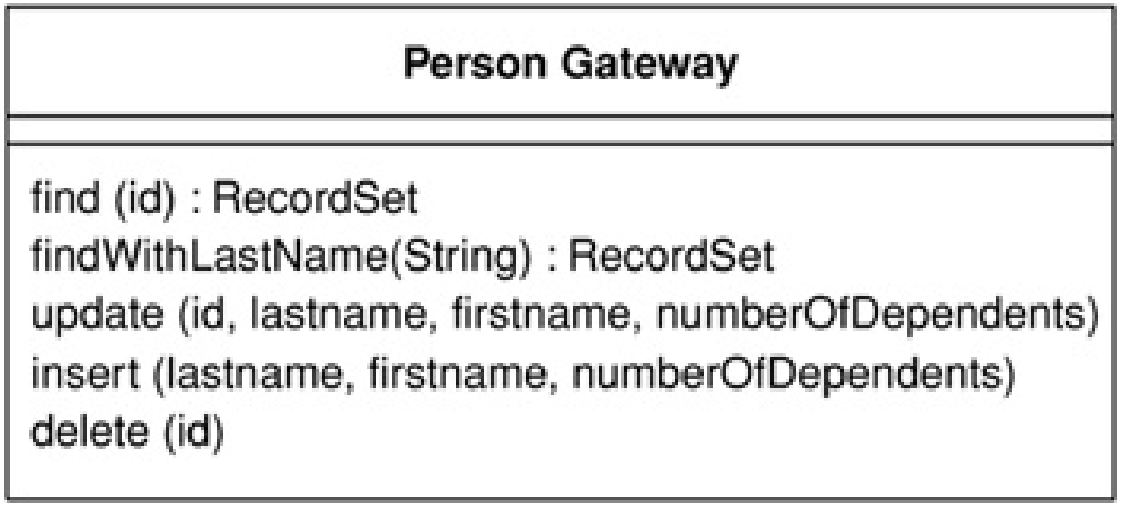


Fowler (2002), Fig. 3.1

Mapping to relational databases

Table Data Gateway

Example table data gateway:



Fowler (2002), Fig. 3.2

Table Data Gateway

- Works best when you have a table module for the domain logic.
- Works fine when we have views or other arbitrary queries on the database.

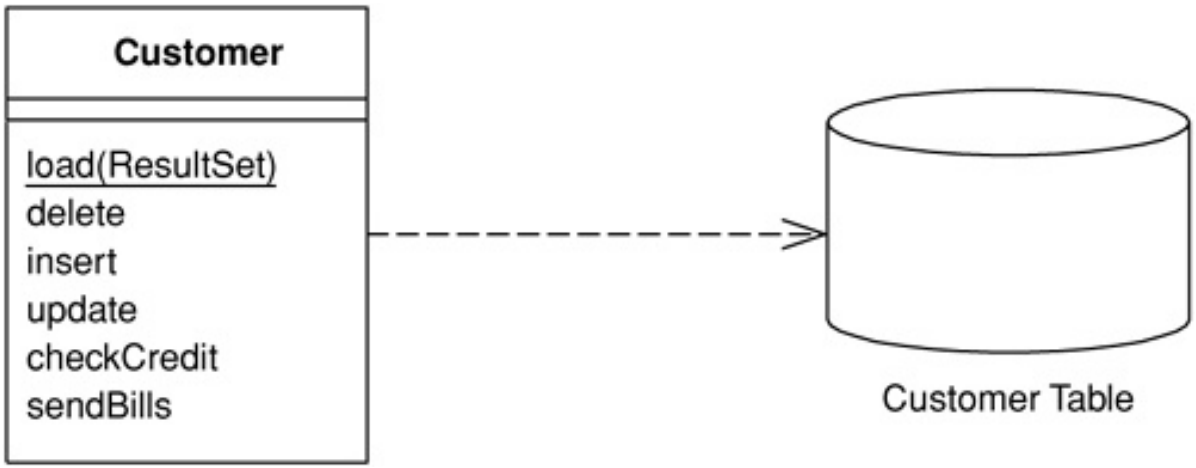
Row Data Gateway

- Works best with a simple Domain Model where domain classes correspond well to the database structure.
- Are not so good for large applications with complex domain models, where we want to decouple the domain class structure from the database structure, and we may want inheritance (especially when we apply GoF and other domain logic patterns).

For simple domain models where row data gateways are appropriate, we often combine domain logic with the row data gateway.

Active Record

An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.



Fowler (2002), Fig. 3.3

Typical Active Record methods:

- Construct an instance of Active Record from a SQL result row set
- Construct a new instance for later insertion into the table
- Static finder methods to wrap commonly-used SQL queries and return Active Record objects (separating finders into another class is good for testing, however)
- Update the database and insert into it the data in the Active Record
- Get and set the fields (these might transform SQL data types to more reasonable types)
- Implement some pieces of the business logic

Mapping to relational databases

Active Record

Example Active Record code for a Person class:

```
class Person...
   private String lastName;
   private String firstName;
   private int numberOfDependents;
```

An ID field goes in the superclass, and the database has the same structure:

```
create table people (ID int primary key, lastname varchar,
                     firstname varchar, number_of_dependents int)
```

Loading an object requires finding it first (in our case with a static method)...

Mapping to relational databases

Active Record

```
class Person...
   private final static String findStatementString =
         "SELECT id, lastname, firstname, number_of_dependents" +
         "  FROM people" +
         "  WHERE id = ?";
   public static Person find(Long id) {
      Person result = (Person) Registry.getPerson(id);
      if (result != null) return result;
      PreparedStatement findStatement = null;
      ResultSet rs = null;
      try {
         findStatement = DB.prepare(findStatementString);
         findStatement.setLong(1, id.longValue());
         rs = findStatement.executeQuery();
         rs.next();
         result = load(rs);
         return result;
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {
         DB.cleanUp(findStatement, rs);
      }
```

Mapping to relational databases

Active Record

```
// We need a Long, not a long, for the Registry's index
public static Person find(long id) {
    return find(new Long(id));
}
public static Person load(ResultSet rs) throws SQLException {
    Long id = new Long(rs.getLong(1));
    Person result = (Person) Registry.getPerson(id);
    if (result != null) return result;
    String lastNameArg = rs.getString(2);
    String firstNameArg = rs.getString(3);
    int numDependentsArg = rs.getInt(4);
    result = new Person(id, lastNameArg, firstNameArg, numDependentsArg);
    Registry.addPerson(result);
    return result;
}
```

We use an Identity Map (class `Registry`) to make sure we don't load the same object twice.

Updating an object is a simple instance method:

```
class Person...

   private final static String updateStatementString =
         "UPDATE people" +
         "  set lastname = ?, firstname = ?, number_of_dependents = ?" +
         "  where id = ?";
   public void update() {
      PreparedStatement updateStatement = null;
      try {
         updateStatement = DB.prepare(updateStatementString);
         updateStatement.setString(1, lastName);
         updateStatement.setString(2, firstName);
         updateStatement.setInt(3, numberOfDependents);
         updateStatement.setInt(4, getID().intValue());
         updateStatement.execute();
      } catch (Exception e) {
         throw new ApplicationException(e);
      } finally {
         DB.cleanUp(updateStatement);
      }
   }
```

Insertion is also straightforward:

```
class Person...
   private final static String insertStatementString =
         "INSERT INTO people VALUES (?, ?, ?, ?)";
   public Long insert() {
      PreparedStatement insertStatement = null;
      try {
         insertStatement = DB.prepare(insertStatementString);
         setID(findNextDatabaseId());
         insertStatement.setInt(1, getID().intValue());
         insertStatement.setString(2, lastName);
         insertStatement.setString(3, firstName);
         insertStatement.setInt(4, numberOfDependents);
         insertStatement.execute();
         Registry.addPerson(this);
         return getID();
      } catch (Exception e) {
         throw new ApplicationException(e);
      } finally {
         DB.cleanUp(insertStatement);
      }
   }
```

Finally the business logic goes right in the Active Record class:

```
class Person...
   public Money getExemption() {
      Money baseExemption = Money.dollars(1500);
      Money dependentExemption = Money.dollars(750);
      return baseExemption.add(dependentExemption.multiply(this.getNumberOfDependen
   }
```

Some advantages of Active Record:

- It works well when the domain model and business logic are simple.
- It is a good pattern for gradual refactoring of a set of Transaction Scripts.

Some disadvantages:

- It cannot handle complex mappings from objects to relations.
- It couples the domain logic to the database schema.

Some tips on reading data with finder methods:

- Finder methods wrap SQL select statements, e.g., `find(id)` or `findForCustomer(customer)`.
- A Table Data Gateway approach lets you put finders in the same classes with the related insert and update methods.
- For a Row Data Gateway or Data Mapper you could make the finders static methods, but that would couple your domain objects to the database solution.
- A better solution: completely separate pluggable finder objects implementing a "finder" interface that could be talking to any database or a mock service.
- Finder methods work on the database, not on objects in memory, so it's best to do all the finding before you start any creations, updates, and deletes in memory.

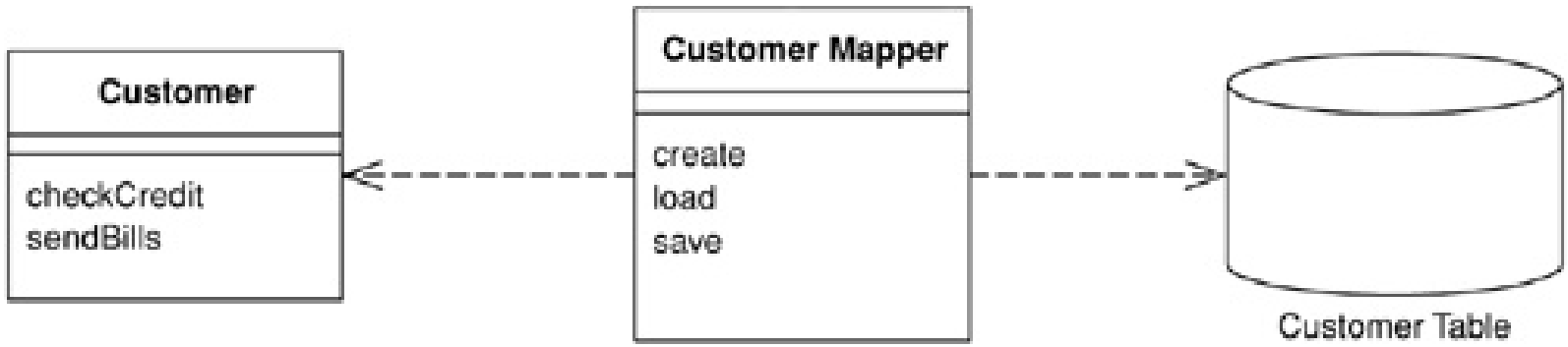
Other tips about reading data:

- Avoid issuing multiple SQL queries to get multiple rows from the same table!
- Large applications will eventually run into database performance issues. Then the queries have to be examined, profiled, and tuned.

For more complex decoupled domain models, Active Record will not suffice.

Data Mapper

A layer of mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself.



Fowler (2002), Fig. 3.4

Fowler recommends never using separate gateways with a domain model. He either uses Active Record or Data Mapper.

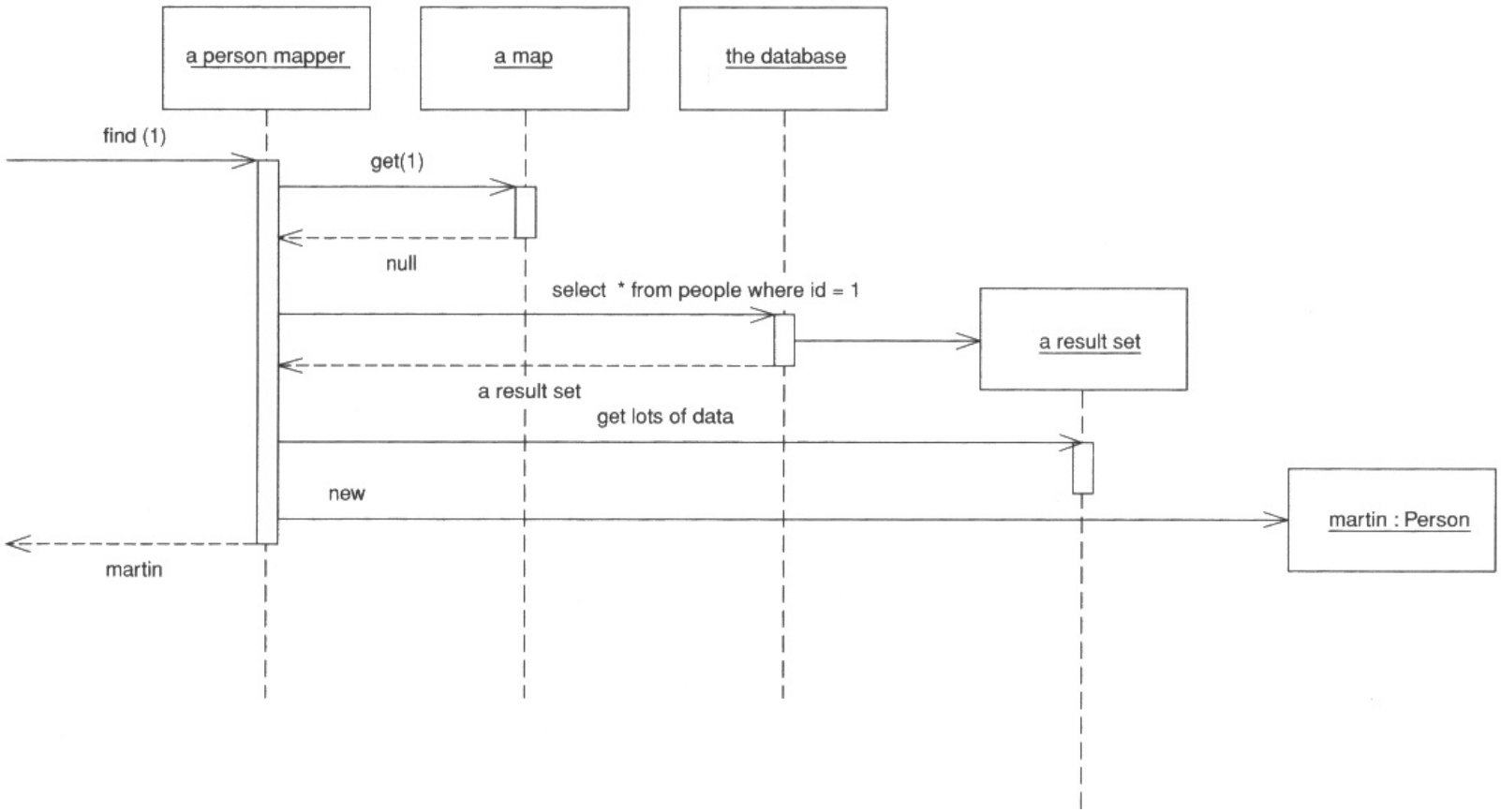
To implement Data Mapper you can:

- Roll your own!
- Use a third party object-relational (O/R) mapping package.

For Java EE, use Java Persistence API (JPA) implementations like Hibernate EntityManager and Toplink JPA.

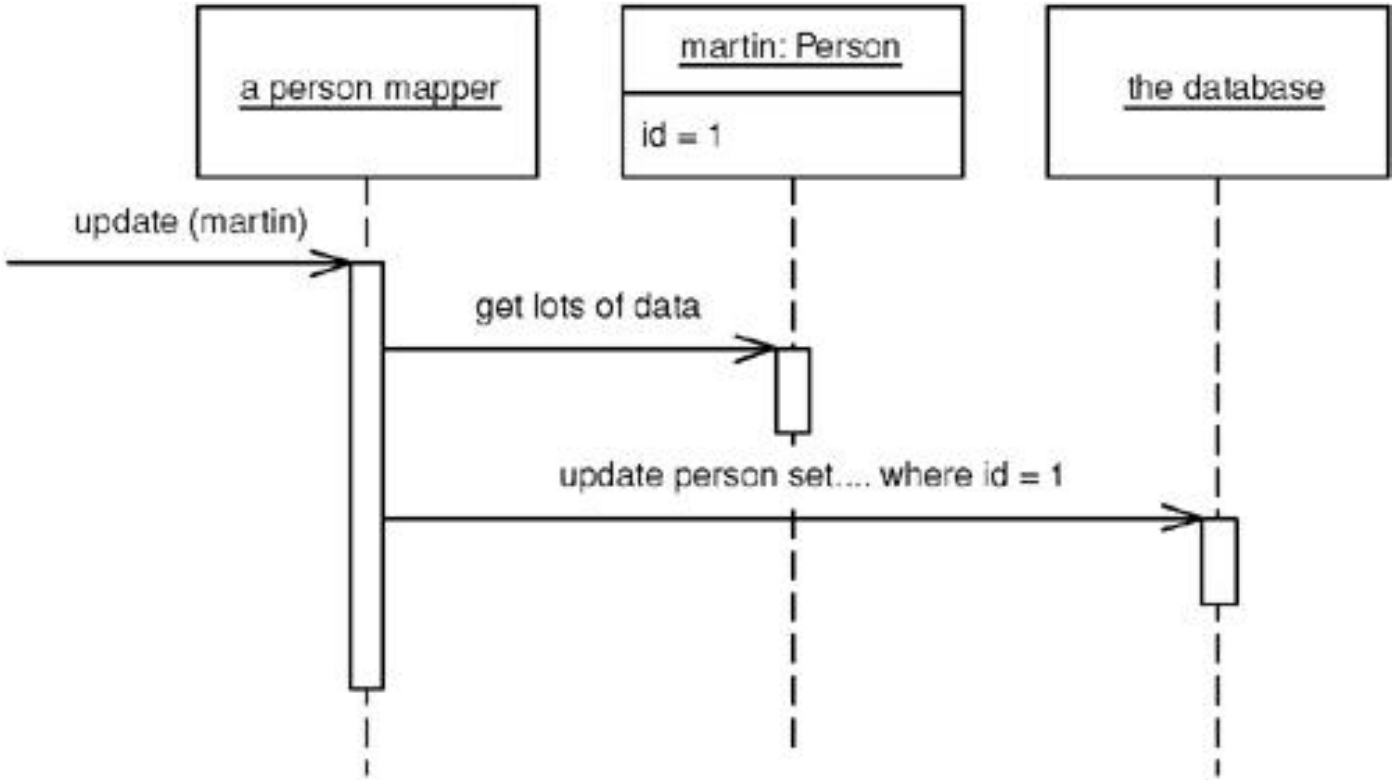
Here we discuss how to roll your own, so you can understand the generic O/R mapping solutions.

Simple example: `PersonMapper` maps `Person` entities to the database with the help of an Identity Map (more on this later):



Fowler (2002), Fig. 10.3

Simple update example with a data mapper:



Fowler (2002), Fig. 10.4

Mapping to relational databases

Data Mapper

Big questions for a data mapper include how much data to load into memory, when to load it, and how to handle transactions.

- To handle transactions, we need a place to register objects that are "dirty" (changed in memory but not saved to the database). This is called a Unit of Work.
- To prevent `find()`-style methods from making multiple copies of the same object in memory, it needs to cooperate with a registry called an Identity Map.
- To prevent a load of one object cascading to hundreds of associated objects, we need a Lazy Load facility.

We consider each of these patterns in turn.

Mapping to relational databases

Unit of Work

How do we get objects to load and save themselves to the database?

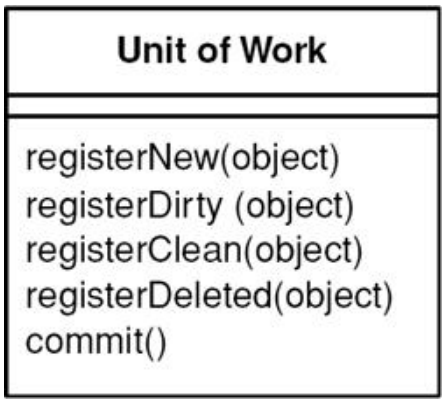
One way is to add load() and save() methods to each object via a Layer Supertype. But still, you must

- Ensure that the database and object state are consistent
- Ensure that updates in one process don't affect reads in another process.

Unit of Work solves this problem.

Unit of Work

Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.

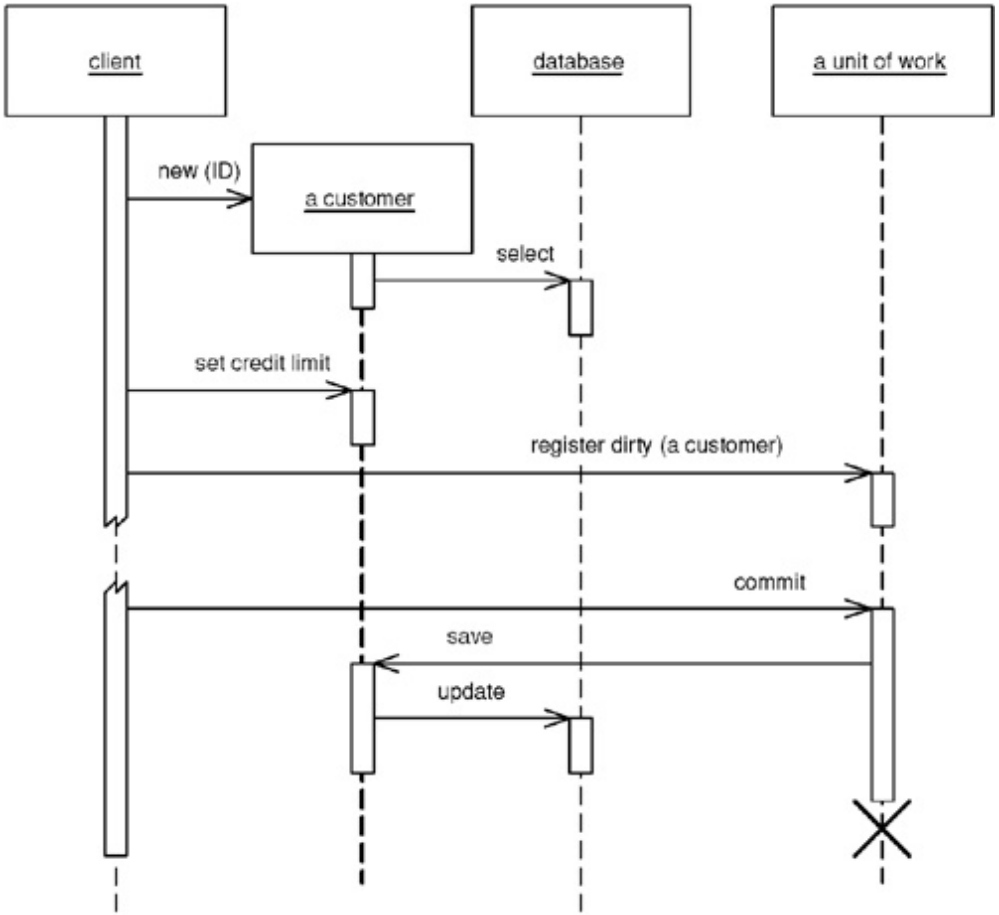


Fowler (2002), p. 184

A unit of work object:

- Acts as a controller for a transaction (open transaction, commit)
- Keeps track of the objects that have been loaded and modified
- Ensures that updated objects are properly committed to the database

Approach 1: client is responsible for registering each dirty entity:



Fowler (2002), Fig. 11.1

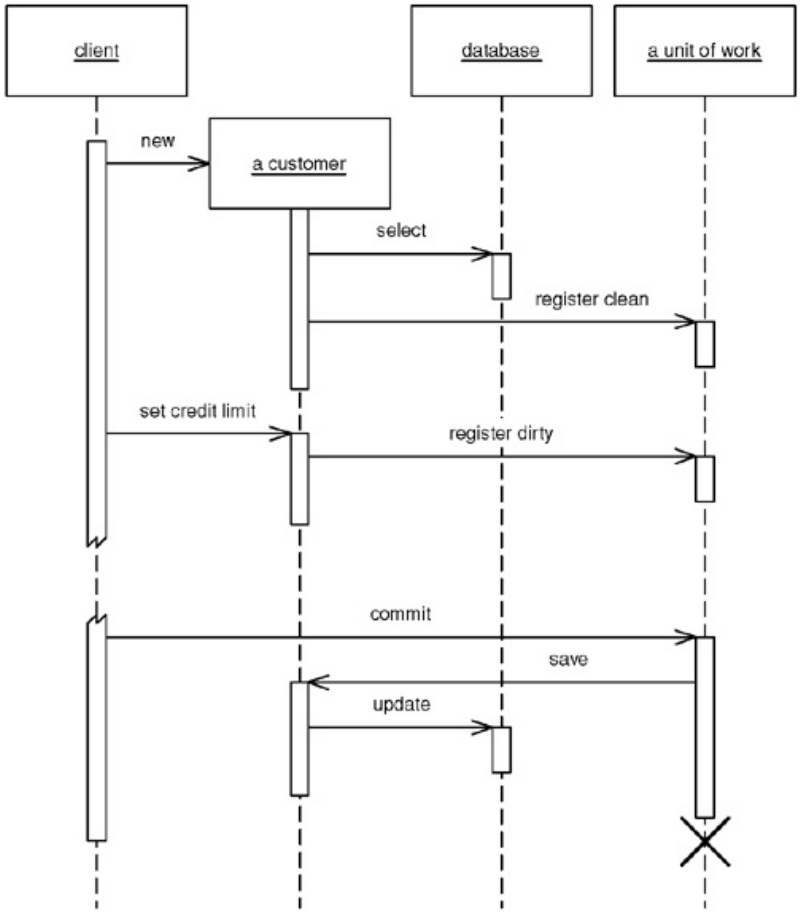
Mapping to relational databases

Unit of Work

Approach 1 is caller registration.

Problems occur when we forget to register our updates.

Approach 2: each entity is responsible for registering itself as dirty in each update method:



Fowler (2002), Fig. 11.2

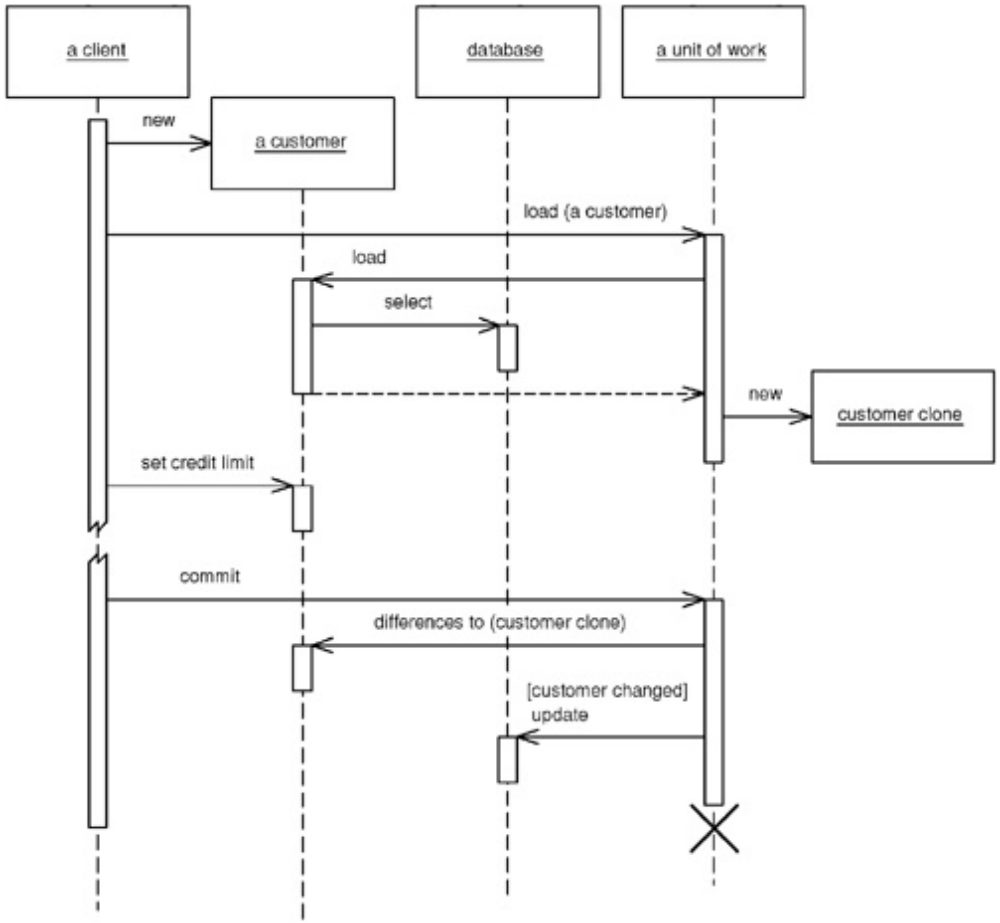
Approach 2 is object registration.

Each object is responsible for registering itself as clean or dirty.

We might still forget to register updates when developing the object.

Automatic code injection through aspect-oriented programming (AOP) is an appropriate way to solve this problem.

Approach 3: the unit of work copies each object on load and compares at commit time to determine dirty status:



Fowler (2002), Fig. 11.3

Mapping to relational databases

Unit of Work

Approach 3 is unit of work controller.

The approach is more heavyweight but protects against forgetfulness.

Oracle's TopLink product uses this pattern.

Mapping to relational databases

Identity Map

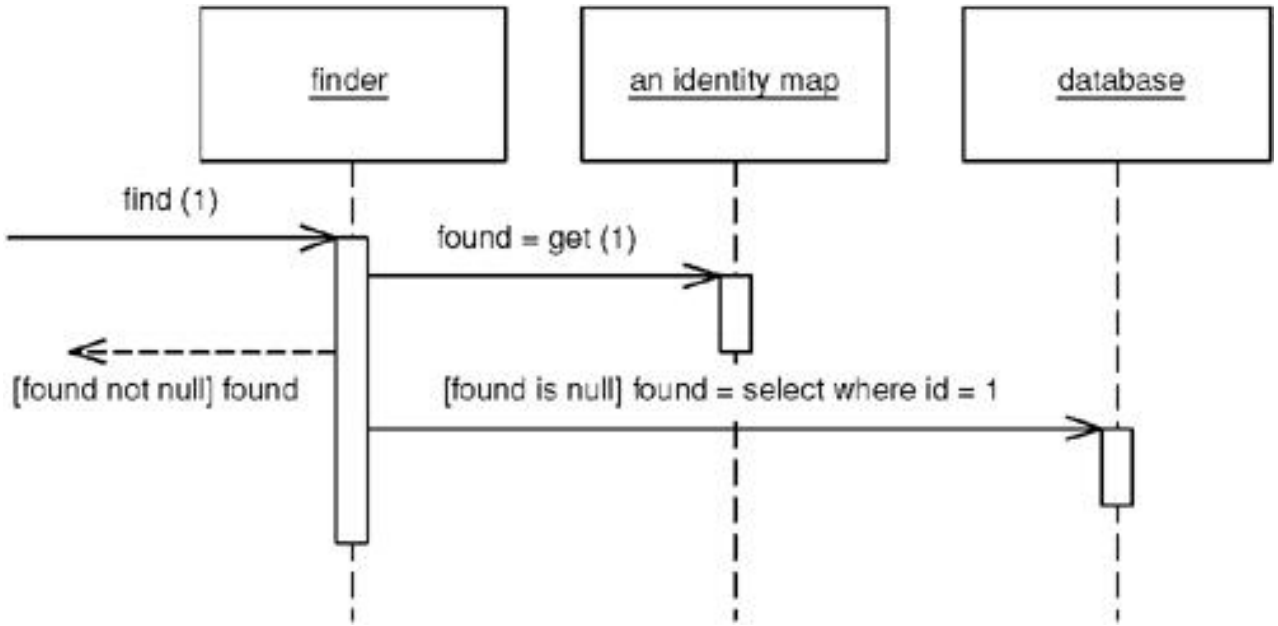
What happens if you read the same object twice?

- You will have two in-memory objects corresponding to one database row.
- If you make updates, the copies become inconsistent.

Identity Map solves this problem.

Identity Map

Ensures that each object gets loaded only once by keeping every loaded object in a map. Looks up objects using the map when referring to them.



Fowler (2002), p. 195

An Identity Map is like a database cache but the cache is for correctness, not performance.

Mapping to relational databases

Lazy Load

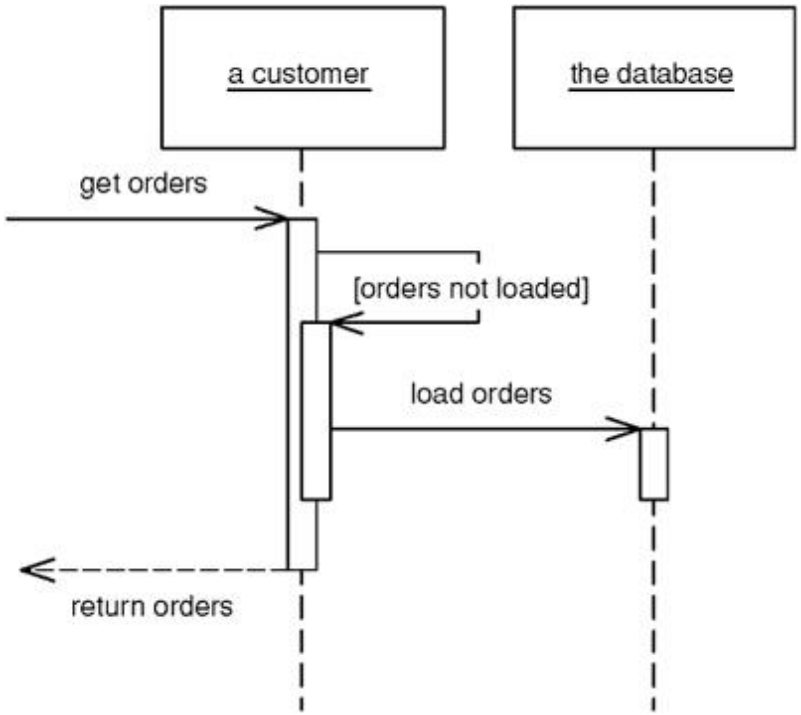
When you load an object having links to to other objects, what to do? If we read linked objects also, we might need too many unnecessary queries.

Lazy Load solves this problem:

- When a read object has links, we point them to placeholder objects
- We load the referenced object later, only when necessary.

Lazy Load

An object that doesn't contain all of the data you need but knows how to get it.



Fowler (2002), p. 200

Mapping to relational databases

Lazy Load

The idea of Lazy Load is to stop loading linked objects at some point in the graph, leaving markers at each point we cut off.

There are four main implementation styles:

- Lazy initialization
- Virtual proxy
- Value holder
- Ghost

Mapping to relational databases

Lazy Load

The simplest Lazy Load implementation is lazy initialization:

- Unloaded fields are left as null.
- We force all access to lazily-loaded fields to be through the getter method.
- The getter first checks the field's value. If the field is null, we load it.

This method is simple, but it couples the object to the database since the object needs to know how to retrieve its own fields.

Lazy intialization thus works well with Active Record and the Gateway patterns but is not suitable for Data Mapper.

Mapping to relational databases

Lazy Load

The virtual proxy (a GoF pattern) adds a layer of indirection:

- The non-loaded object is replaced with a placeholder (the virtual proxy) that looks like the desired object but doesn't contain any data.
- When any method on the virtual proxy is called, it realizes it doesn't exist and loads itself from the database.

If identity of the virtual proxy object is important, then bookkeeping becomes difficult. Thus it's recommended to only use virtual proxies for complex value objects like collections where identity is not important.

Another approach to Lazy Load is to use a value holder in which one object wraps another, which isn't loaded until necessary.

The last approach is to use a ghost:

- We create the real object itself, but only fill in the ID.
- Whenever a field is accessed, we load the state of the whole object.
- The ghost can be placed in an Identity Map, avoiding identity problems.

Mapping to relational databases

Lazy Load

- Lazy Load can cause performance problems if the database accesses are too fine-grained.
- Lazy loading can be implemented consistently using AOP.
- For efficiency, the application might try to arrange things so the lazy loader fetches just the right object graph for the active use case.
- Lazy loading is configured for EJB3 entities using the `Fetch` attribute of the relationship annotation for each field.

Now, back to Data Mapper.

Data Mapper becomes necessary when you want persistent objects that contain collections, use inheritance, and so on.

A Data Mapper is a layer completely separate from the domain objects; the domain objects don't need to know about the database schema or SQL.

Mapping to relational databases

Data Mapper

In a moment we'll look at a finder method that loads instances from the database by ID.

The first issue we will come across is that the mapper class is in a different class and package from the domain object.

This means that to support `save()`, we need special public getters for all fields that return the raw attributes. These should only be called in the context of a database operation.

Similarly, for `load()`, we need a rich constructor that sets all fields, or special public setters for all fields.

Now we develop a Data Mapper for `Person`, noting that Data Mapper is overkill for such a simple case:

```
class Person...

   private String lastName;
   private String firstName;
   private int numberOfDependents;
```

Here is the database table:

```
create table people (ID int primary key, lastname varchar,
                     firstname varchar, number_of_dependents int)
```

Person-specific code is placed in PersonMapper:

```
class PersonMapper...

   protected String findStatement() {
      return "SELECT " + COLUMNS +
         "  FROM people" +
         "  WHERE id = ?";
   }
   public static final String COLUMNS = " id, lastname, firstname, " +
                                        "number_of_dependents ";
   public Person find(Long id) {
      return (Person) abstractFind(id);
   }
   public Person find(long id) {
      return find(new Long(id));
   }
```

Generic code goes in a Layer Supertype assuming the object has an ID:

```
class AbstractMapper...

   protected Map loadedMap = new HashMap();
   abstract protected String findStatement();
   protected DomainObject abstractFind(Long id) {
      DomainObject result = (DomainObject) loadedMap.get(id);
      if (result != null) return result;
      PreparedStatement findStatement = null;
      try {
         findStatement = DB.prepare(findStatement());
         findStatement.setLong(1, id.longValue());
         ResultSet rs = findStatement.executeQuery();
         rs.next();
         result = load(rs);
         return result;
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {
         DB.cleanUp(findStatement);
      }
   }
```

Load behavior is split between the concrete mapper and the superclass:

```
class AbstractMapper...
   protected DomainObject load(ResultSet rs) throws SQLException {
      Long id = new Long(rs.getLong(1));
      if (loadedMap.containsKey(id)) return (DomainObject) loadedMap.get(id);
      DomainObject result = doLoad(id, rs);
      loadedMap.put(id, result);
      return result;
   }
   abstract protected DomainObject doLoad(Long id, ResultSet rs) throws SQLExceptio

class PersonMapper...
   protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException {
      String lastNameArg = rs.getString(2);
      String firstNameArg = rs.getString(3);
      int numDependentsArg = rs.getInt(4);
      return new Person(id, lastNameArg, firstNameArg, numDependentsArg);
   }
```

Mapping to relational databases

Data Mapper

That was find by ID. Here is a simple version of find by last name:

```
class PersonMapper...
   private static String findLastNameStatement =
         "SELECT " + COLUMNS +
         "  FROM people " +
         "  WHERE UPPER(lastname) like UPPER(?)" +
         "  ORDER BY lastname";
   public List findByLastName(String name) {
      PreparedStatement stmt = null;
      ResultSet rs = null;
      try {
         stmt = DB.prepare(findLastNameStatement);
         stmt.setString(1, name);
         rs = stmt.executeQuery();
         return loadAll(rs);        // implemented by the Layer Supertype
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {
         DB.cleanUp(stmt, rs);
      }
   }
```

Mapping to relational databases

Data Mapper

```
class AbstractMapper...
   protected List loadAll(ResultSet rs) throws SQLException {
      List result = new ArrayList();
      while (rs.next())
         result.add(load(rs));
      return result;
   }
```

Actually, note that the method should also check the Identity Map for each person in the result set.

Mapping to relational databases

Data Mapper

Instead of duplicating the find code across every mapper, we can refactor to make a generic finder:

```
class AbstractMapper...
   public List findMany(StatementSource source) {
      PreparedStatement stmt = null;
      ResultSet rs = null;
      try {
         stmt = DB.prepare(source.sql());
         for (int i = 0; i < source.parameters().length; i++)
            stmt.setObject(i+1, source.parameters()[i]);
         rs = stmt.executeQuery();
         return loadAll(rs);
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {
         DB.cleanUp(stmt, rs);
      }
   }
```

The generic finder requires a wrapper for the SQL query and its parameters:

```
interface StatementSource...
   String sql();
   Object[] parameters();
```

The implementation of the `StatementSource` interface can be an inner class (next page)...

This approach makes creating new finder methods easy: just instantiate the `StatementSource` for your query and pass to `findMany()`.

Mapping to relational databases

Data Mapper

```
class PersonMapper...
   public List findByLastName2(String pattern) {
      return findMany(new FindByLastName(pattern));
   }
   static class FindByLastName implements StatementSource {
      private String lastName;
      public FindByLastName(String lastName) {
         this.lastName = lastName;
      }
      public String sql() {
         return
            "SELECT " + COLUMNS +
            "  FROM people " +
            "  WHERE UPPER(lastname) like UPPER(?)" +
            "  ORDER BY lastname";
      }
      public Object[] parameters() {
         Object[] result = {lastName};
         return result;
      }
   }
```

Updates are specific to the object we're mapping:

```
class PersonMapper...
   private static final String updateStatementString =
         "UPDATE people " +
         "  SET lastname = ?, firstname = ?, number_of_dependents = ? " +
         "  WHERE id = ?";
   public void update(Person subject) {
      PreparedStatement updateStatement = null;
      try {
         updateStatement = DB.prepare(updateStatementString);
         updateStatement.setString(1, subject.getLastName());
         updateStatement.setString(2, subject.getFirstName());
         updateStatement.setInt(3, subject.getNumberOfDependents());
         updateStatement.setInt(4, subject.getID().intValue());
         updateStatement.execute();
      } catch (Exception e) {
         throw new ApplicationException(e);
      } finally {
         DB.cleanUp(updateStatement);
      }
   }
```

Mapping to relational databases

Data Mapper

Inserts can be partly generic:

```
class AbstractMapper...
   public Long insert(DomainObject subject) {
      PreparedStatement insertStatement = null;
      try {
         insertStatement = DB.prepare(insertStatement());
         subject.setID(findNextDatabaseId());
         insertStatement.setInt(1, subject.getID().intValue());
         doInsert(subject, insertStatement);   // object-specific
         insertStatement.execute();
         loadedMap.put(subject.getID(), subject);
         return subject.getID();
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {
         DB.cleanUp(insertStatement);
      }
   }
   abstract protected String insertStatement();
   abstract protected void doInsert(DomainObject subject,
                                    PreparedStatement insertStatement)
         throws SQLException;
```

... and partly object-specific:

```
class PersonMapper...
   protected String insertStatement() {
      return "INSERT INTO people VALUES (?, ?, ?, ?)";
   }
   protected void doInsert(
         DomainObject abstractSubject,
         PreparedStatement stmt)
         throws SQLException
   {
      Person subject = (Person) abstractSubject;
      stmt.setString(2, subject.getLastName());
      stmt.setString(3, subject.getFirstName());
      stmt.setInt(4, subject.getNumberOfDependents());
   }
```

That's all for a basic approach to Data Mapper!

Fowler recommends separating the finders from the mappers so that the domain layer can use them. See last example in Chapter 10.

Now that we understand what Data Mapper is doing, we consider how to implement the object-relational mapping.

If you're using Row Data Gateway, Active Record, or Data Mapper, you need to know the common methods for mapping between objects and tables.

The main problem is links:

- Objects use a reference or memory address (pointer).
- Relational databases use foreign key references to another table.
- Objects use collections for many-valued fields.
- Relational databases should be normalized to be single-valued.

Example: an `Order` object would have a collection of `LineItems` that don't refer back to the order. The database would have a table of line items, each with a foreign key reference to the containing order.

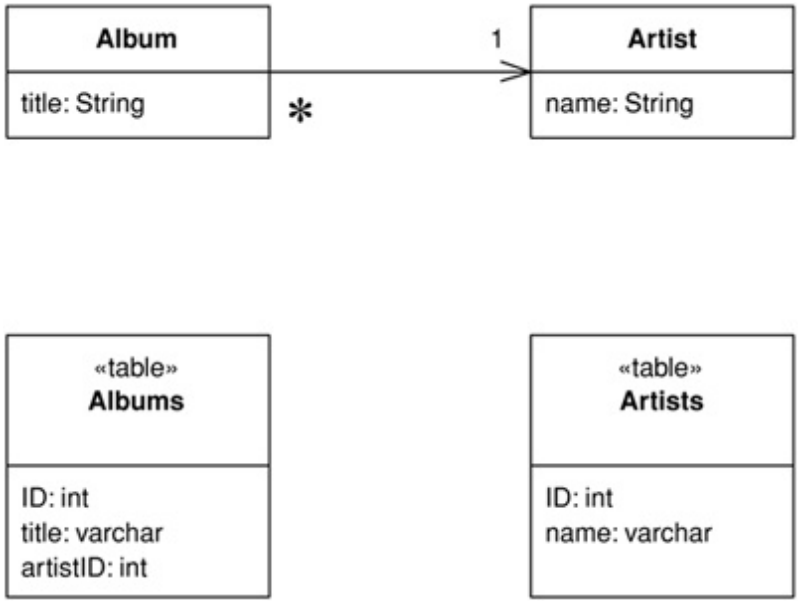
To solve the backward reference problem:

- We keep the relational identity of each object as an Identity Field
- We look up the Identity Field values when needed to map back and forth between object references and relational keys.

For a read:

- We use an Identity Map as a lookup table from relational keys to objects
- Then we use a Foreign Key Mapping to connect the inter-object reference

Using Foreign Key Reference for a many-to-one relationship:

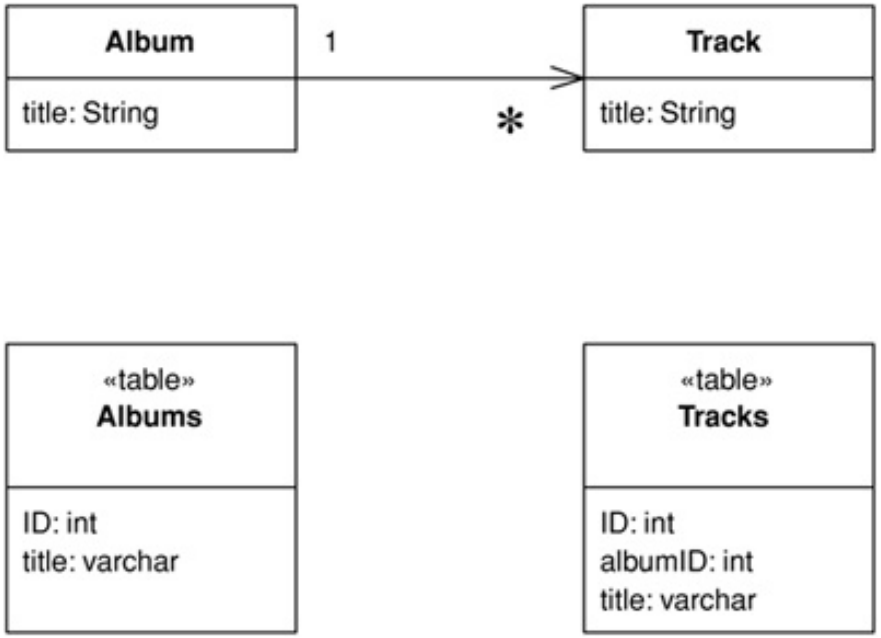


Fowler (2002), Fig. 3.5

When we load an `Album` and the `Artist` key is not in the Identity Map, we defer (Lazy Load) or use the Identity Field to load the `Artist` from the database.

When we save, we use the Identity Field to update the right row.

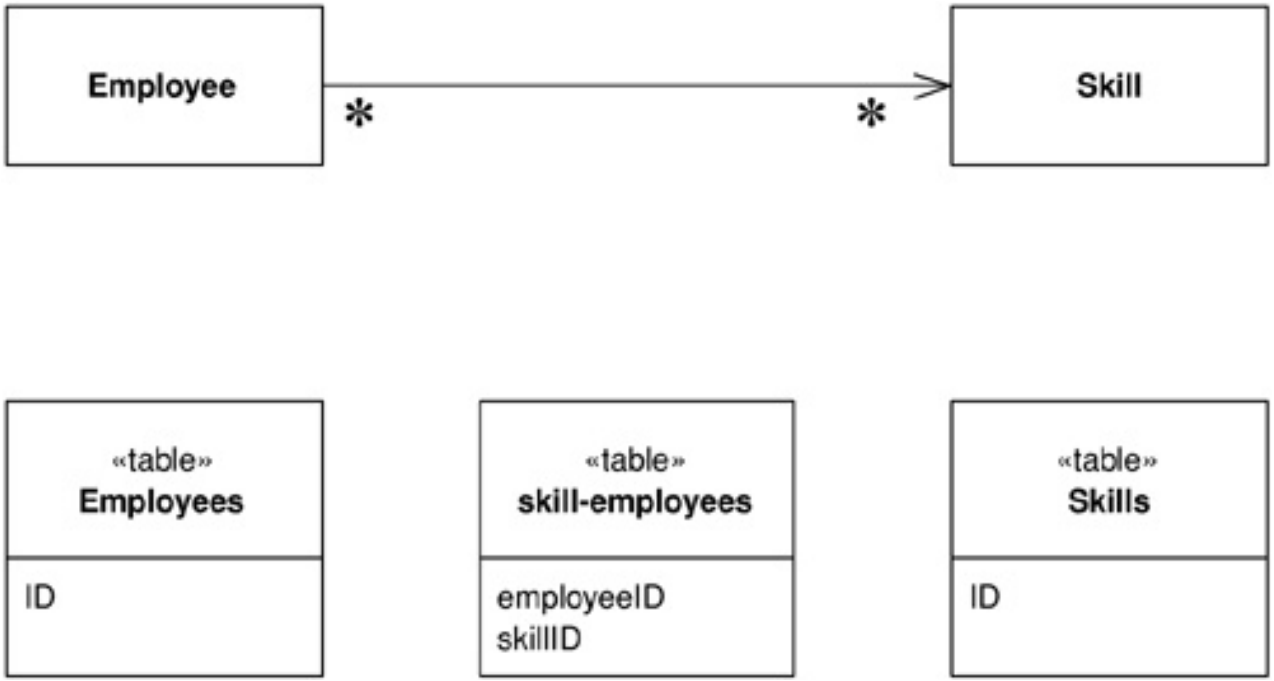
For 1-to-many, there is a collection on the "1" side in the domain model and the reference pattern is reversed in the data model:



Fowler (2002), Fig. 3.6

When we load an `Album`, we defer (Lazy Load) or issue a query to find all `Tracks` referring to our `Album`. For each returned row, we create a `Track` object and add it to the `Album`'s collection of `Tracks`.

For many-to-many relationships, we know we need a separate DB table for the mapping:



Fowler (2002), Fig. 3.7

This is Association Table Mapping.

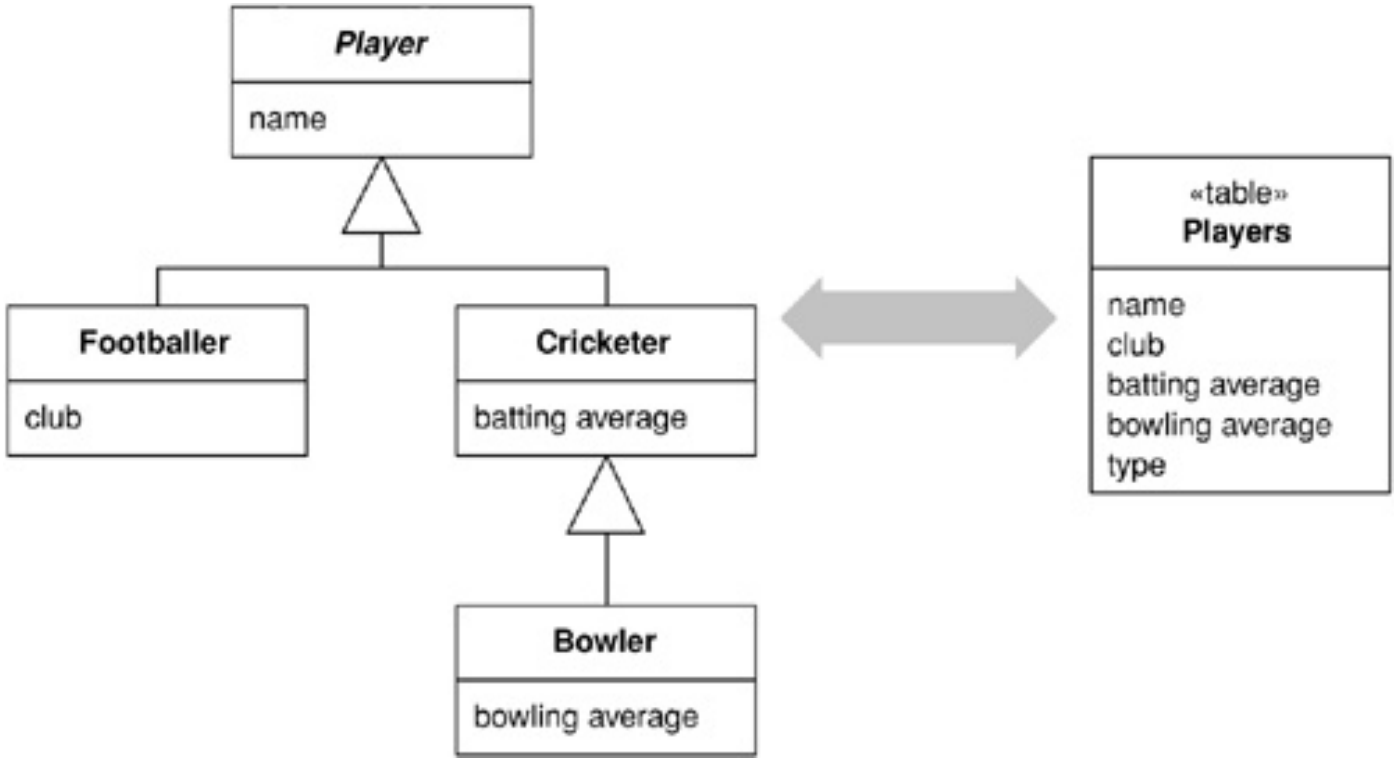
More O/R mapping tips:

- OO languages normally use arrays or lists, but relations are unordered. This means it is best to use unordered collections (sets) in the domain layer whenever possible.
- Small objects like `DateRange` and `Money` don't need to be stored in their own tables. In the Value Objects, when we read an object containing a small object, we get the fields of the small object directly from the row for the containing object, and store them in the containing object as an Embedded Value. When writing, we just get the relevant fields from the Value Object and put them directly into the table.
- This approach can be taken further by storing complex object networks as serialized LOBs (Large Objects), if we don't care about structured queries, or in XML, if we do care.

For inheritance, there are 3 options:

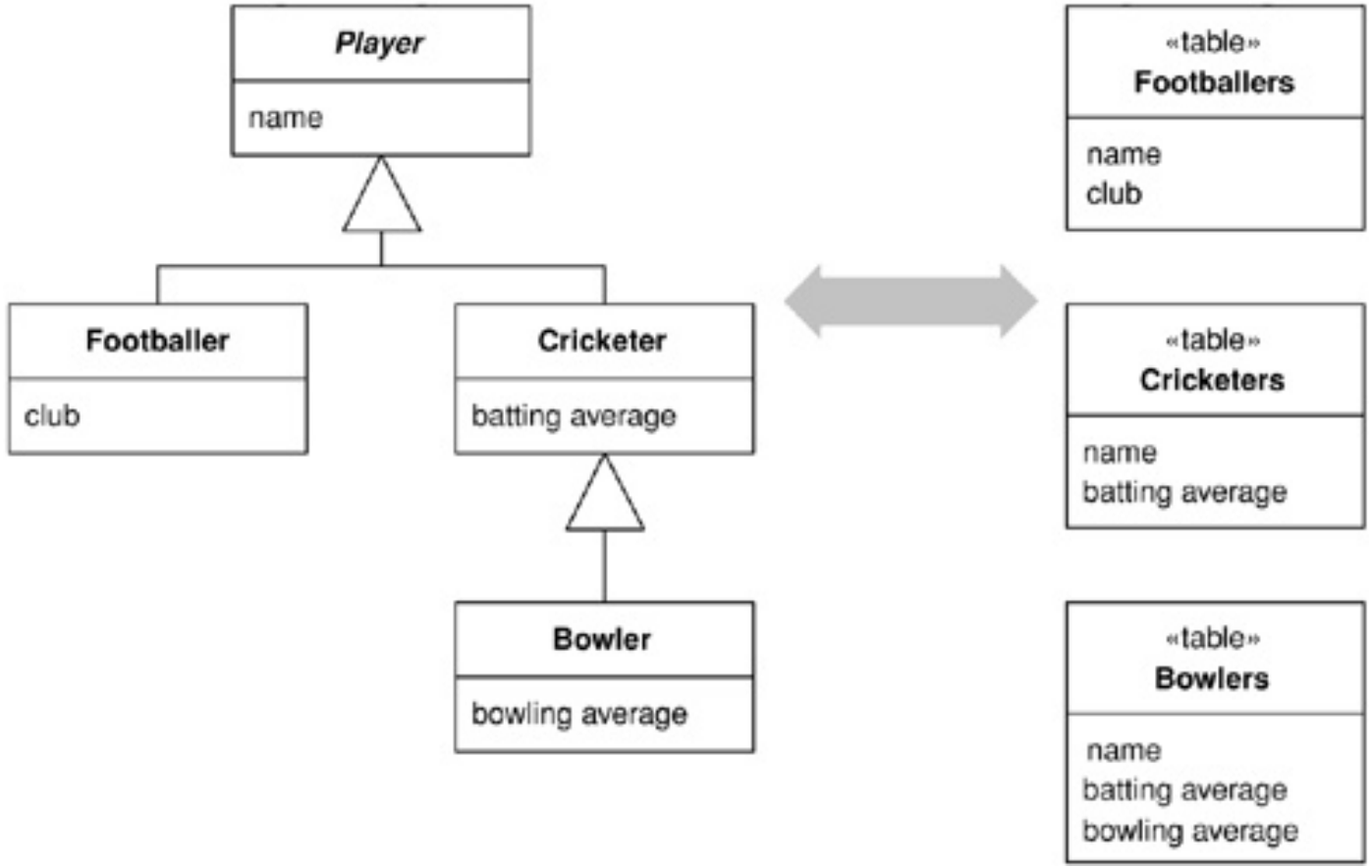
- One table for all classes in the hierarchy (Single Table Inheritance)
- One table per concrete class in the hierarchy (Concrete Table Inheritance)
- One table for each class (Class Table Inheritance).

Single Table Inheritance is convenient but wastes space and could create huge tables:



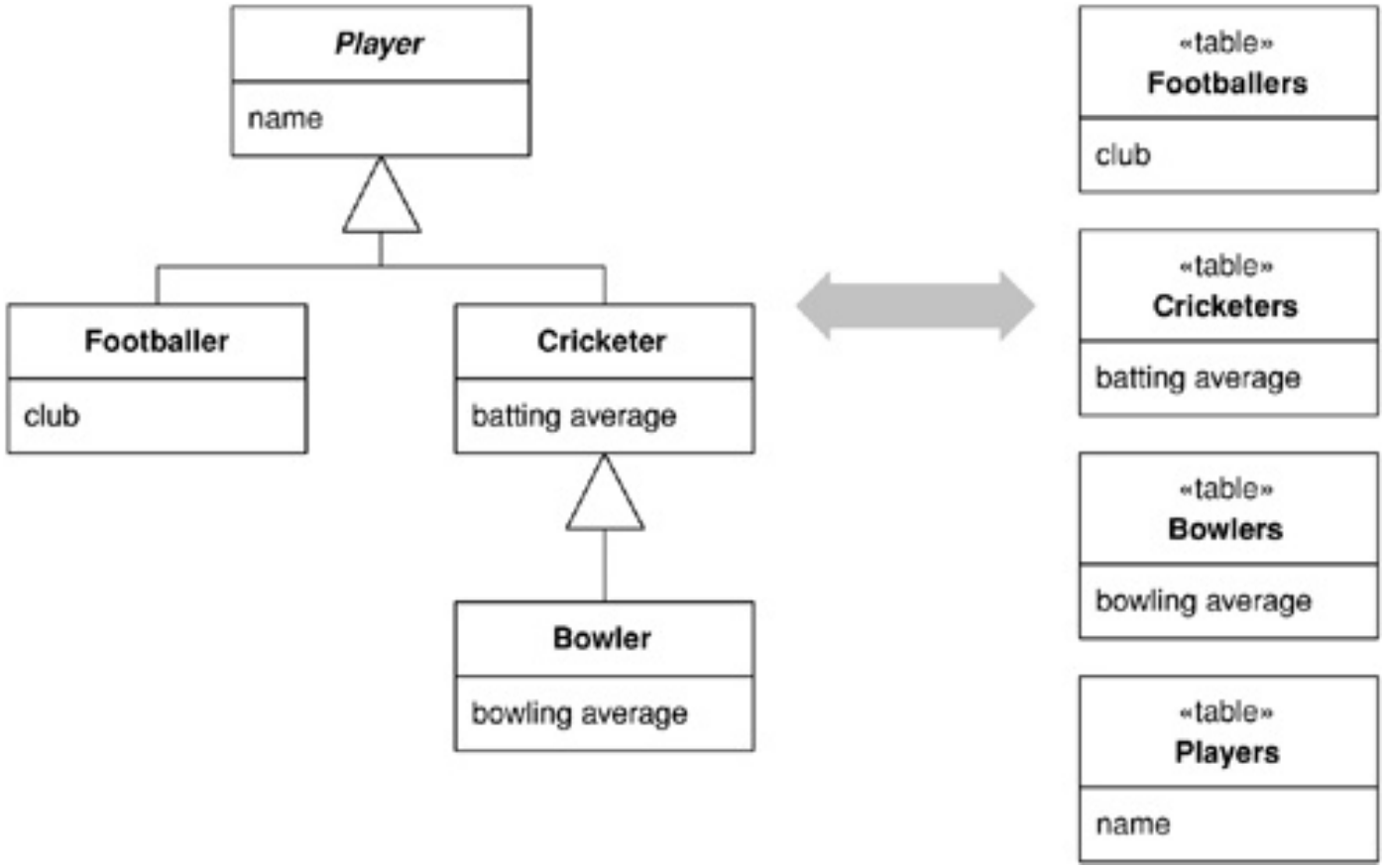
Fowler (2002), Fig. 3.8

Concrete Table Inheritance avoids waste of space but is brittle to change (think about the effects of adding a field):



Fowler (2002), Fig. 3.9

Class Table Inheritance needs joins for a single object, but is a simple mapping:



Fowler (2002), Fig. 3.10

The inheritance patterns can be mixed even within one hierarchy.

You might start with Single Table Inheritance but start to refactor when performance or wasted space becomes a problem.

There are variations for multiple inheritance or interfaces.

When you begin implementing your data source layer, you'll encounter one of three cases:

- You get to choose the DB schema.
- You have to map to an existing schema and aren't allowed to change that schema.
- You have to map to an existing schema, but changes might be possible.

Some tips for case 1 (you choose the DB schema):

- Avoid making your domain model look like the database — build it without worrying about the DB, so you can focus on keeping the domain logic simple.
- If it turns out that the domain model looks like a database design, then you can use Active Record.
- Build the database gradually, with each iteration, no more than 6 weeks in length, and preferably shorter.
- Design the Domain Model pieces first, then worry about the database, but do map to the database on every iteration.

Mapping to relational databases

Using metadata

For large applications if you are rolling your own Data Mapper you'll end up repeating a lot of code.

Eventually you might refactor with Metadata Mapping. In this pattern we describe the mapping between object fields and database columns. Example:

```
<field name="customer" targetClass="Customer" dbColumn="custID"
       targetTable="customers" lowerBound="1" upperBound ="999999"
       setter="loadCustomer" />
```

If you go far enough in using metadata to abstract the database, you have a Repository using Query Objects in which developers don't know or care whether their data is coming from the database or memory or both. The metadata approach is what's used by generic object-relational mapping systems.

Some tips on dealing with database connections:

- Use connection pooling. Try to explicitly close connections when you're done with them, or they might sit around forever or stay open until the garbage collector collects them.
- The best way to manage a connection is inside a Unit of Work — get the database connection when you start the transaction and release it when you commit or roll back.
- For quick things like read-only access to data outside a transaction, you can open a new connection for each command and close it immediately.

Mapping to relational databases

Other tips

A few more tips for database mapping code:

- Don't use select *
- Use column names, not numeric indices, to reference columns in a Record Set
- Use precompiled SQL queries then supply parameters
- Get optimization help from the project's database expert

That's all for database mapping! Next we'll see how the database mapping patterns are implemented on the Java EE platform, then come back to patterns for concurrency, session state, and distribution.

Outline

Concurrency

The issues

Concurrency is problematic because it is difficult to test:

- Everything depends on the relative order of events within different threads or processes.

Enterprise systems are different from real time systems, however:

- Low-level concurrency is managed by the online concurrency mechanisms of the RDBMS.
- But when a business transaction doesn't fit into a DB or system transaction, we need to worry about offline concurrency (concurrency control for data manipulated over multiple system transactions).
- In addition to multi-request business transactions, we sometimes have to worry about concurrency in a multi-threaded application server.

Concurrency

The issues

There are many concepts we need to understand about concurrency:

- Requests, transactions, and sessions.
- Processes and threads.
- Isolation and immutability.
- Optimistic and pessimistic locking.
- ACID guarantees.
- Types of transactions.
- Isolation models.
- Locking patterns.
- Application server concurrency patterns.

As an example, consider the essential concurrency problems in a source code control system.

We have two essential concurrency problems:

- Lost updates occur when two users try to commit modifications to the same file. The result is that the first committer's update can be lost.
- Inconsistent reads occur when one user can see the partial transactions of other users. Two resources that are read at different times may be inconsistent with each other.

We'll see that for short transactions, the system can provide serializable isolation, preventing these problems.

For long-running transactions, we need weaker isolation.

Correctness or safety means that the final state should always be the same as if there was never more than one person working on the data at the same time.

Correctness is guaranteed by only allowing one person to work on the data at the same time (locking everyone else out) but that reduces liveness, or how much concurrency is allowed.

In practice, we will have to find an appropriate tradeoff between safety and liveness.

Think of the system as the receiver of a series of requests from users.

Usually we think of requests in isolation, but they could be related. An example: a client places an order then cancels.

A transaction is a series of requests that should be treated as one request.

We have transactions on both "sides" of the application server:

- The user interacts with the application to form business transactions.
- The application interacts with the database to form system transactions.

A session is a long-running interaction between the client and server possibly involving more than one request.

A session can potentially contain multiple business transactions. The opposite is also possible but ill-advised!

A process is an execution context that has its own dedicated memory address space.

A thread is an execution context in which some resources (usually the memory address space) can be shared with other execution contexts.

An isolated thread is one that doesn't share memory with other threads (but still shares other resources, making it more efficient).

It would be nice if we could have one unique process for every session for the lifetime of the session, but more typically we have to actively associate requests with new processes or threads then with sessions.

The two methods for overcoming concurrency problems in enterprise applications (mainly lost updates and inconsistent reads) are isolation and immutability:

- Isolation allows a process to enter an isolated zone in which concurrency isn't a problem. Conceptually, we work in a memory address space separate from all other processes, and lock all external resources.
- Immutability means recognizing data that cannot be modified. All concurrency problems involve updates, and immutable data cannot be updated, so immutability eliminates concurrency problems.

When we have resources that can't be isolated within a session's process, we use either optimistic or pessimistic locking.

Optimistic locking (conflict detection):

- Concurrent reads and writes are allowed, and we generate exceptions when we see conflicts.
- Example: A and B check out a file at the same time. A commits first, and his commit goes through, but when B tries to commit there will be a conflict, and B will be responsible for fixing it.

Pessimistic locking (conflict prevention):

- Once a resource is read for update, nobody else is allowed to edit it until the transaction is finished.
- Example: when A checks a file out, B can't check out until A commits.

Optimistic locking works well when conflicts are rare and can be resolved, as in source code control.

For business data, conflict resolution is usually too difficult, so optimistic approaches typically rollback and retry.

When neither conflict resolution nor rollback is appropriate, we have to use pessimistic approaches.

Solving inconsistent reads with pessimistic techniques:

- Each resource has a read lock and a write lock.
- To read data, we say we need a read lock.
- To write data, we say we need a write lock.
- Any number of clients can obtain a read lock simultaneously.
- If anyone has a read lock, nobody can get a write lock.
- If anyone has the write lock, nobody else is allowed any lock.

Concurrency

Preventing inconsistent reads

One way of solving inconsistent reads using optimistic techniques:

- The system maintains a version number for system states.
- When reading, we make sure every resource we read has the same version number.

Alternatively, we can use timestamps on each resource.

Issues with optimistic techniques:

- Versioning and timestamps are feasible in source control, where the updates are infrequent and the version history needs to be kept anyway.
- Versioning is almost nonexistent in databases, where updates are frequent and history maintenance would be too expensive. (That aside, sometimes you do need precise logging of your business transactions, but normally you have to design that yourself.)

Locking means one thread might have to wait for another before it can proceed.

A deadlock occurs when there is a cyclic wait for a set of locks.

Techniques to address a deadlock:

- Detection and recovery: choose a victim and force it to roll back (difficult and drastic), or put a timeout on every lock request (making you a victim when your request times out — simple but more drastic).
- Prevention: force all processes to get all their locks at once in the beginning before they can continue, or put an ordering on the locks and force each process to grab them in order.

The advice is to use simple conservative schemes for enterprise applications, even if it causes unnecessary victimization.

Our main tool is the transaction. A transaction is a bounded sequence of work with start points and end points well defined.

Software transactions are described in terms of ACID properties:

- Atomicity (all-or-none): every step must complete successfully, or all the steps must roll back. Partial completion is not allowed. If there's a system crash in the middle of a bank transfer we must go back to the original state.
- Consistency: the resources must be consistent and noncorrupt at both the beginning and end of the transaction.
- Isolation: the result of a transaction must not be visibile until it has completed successfully.
- Durability: any result of a committed transaction must be permanent, i.e., survive any kind of crash.

Concurrency

Transactional resources

We are most familiar with transactions in databases, but they are also common in many other places. Some terms:

- A transactional resource is anything using transactions to control concurrency. "Transactional resource" and "database" can be used interchangeably.
- Transactions spanning multiple requests are called long transactions.
- A request transaction is one started when a request is received and completed at the end of request processing.
- A late transaction isn't opened until all reads are complete. This improves efficiency but allows inconsistent reads.

Whenever possible, use request transactions, not long transactions. This maximizes throughput.

Some development environments allow us to tag a method as transactional, so that a transaction is automatically begun at the beginning of the request and completed at the end.

Concurrency

Isolation ↔ liveness tradeoff

The "I" (Isolation) in ACID is the most difficult to achieve in practice:

- We have already seen how pessimistic locking makes it easy to achieve strong isolation.
- But pessimistic locking reduces liveness.
- We cannot allow resources to be locked indefinitely by long transactions.

Concurrency

Isolation ↔ liveness tradeoff

SQL defines four levels of isolation that are useful for us.

In addition to strong (serializable) isolation, we consider weaker forms of isolation:

- Repeatable read isolation
- Read committed isolation
- Read uncommitted isolation

Concurrency

Isolation ↔ liveness tradeoff

Serializable isolation:

- Transactions can be executed concurrently.
- Each transaction works on a snapshot of the global state that is correct and consistent at the beginning of the transaction.
- The result is guaranteed to correspond to one of the possible serialized results.

Note that SQL's serializable isolation is not strict serialization: it prevents inconsistent reads, but allows lost updates.

Repeatable read isolation:

- You may see some but not all new items being inserted by other users while you do a read (phantom reads).
- However, you'll always get the same result if you repeat the read within the same transaction.

Concurrency

Isolation ↔ liveness tradeoff

Read committed isolation:

- You may see updates performed by other users during your transaction.
- Reads are thus in general unrepeatable.
- But you are guaranteed to only see updates committed by other users.

Read uncommitted isolation (non-isolation, really):

- You see updates performed by other users during your transaction.
- You can even see updates from transactions that have not yet been committed.

Read uncommitted is also called dirty read. If another user's transaction rolls back, we may have read data that should never have existed at all!

Summary:

Isolation level	Dirty Read	Unrepeatable Read	Phantom Read
Read uncommitted	Yes	Yes	Yes
Read committed	No	Yes	Yes
Repeatable read	No	No	Yes
Serializable	No	No	No

Advice: always use serializable isolation unless performance is an issue, in which case you might weaken the isolation for some or all transactions.

The SQL standards for isolation are mainly applicable to system transactions.

But we want ACID guarantees for our business transactions. For example:

- User logs in to online banking system.
- User sets up some bill payments.
- User clicks OK to commit, or Cancel to roll back.

How to achieve ACID business transactions?

Simple solution for ACID business transactions: execute the business transaction within a single system transaction.

Use this approach whenever possible.

But if we have many concurrent users, we need to implement offline concurrency. The business transaction will be broken down into a series of system transactions.

Example solution for the online banking example using offline concurrency:

1. User logs in.
2. System sets up an isolated Unit of Work for the bill payment business transaction.
3. Each new payment is registered with the Unit of Work.
4. If user cancels, the Unit of Work is destroyed.
5. If user authorizes, the system executes a system transaction.
6. If integrity violations occur, we roll back and ask the user to start again.

We see that isolation is difficult to enforce for business transactions.

- Our own uncommitted updates need to be isolated from other users, probably as part of the session state (more on this soon).
- Preventing lost updates might require locking.
- Preventing inconsistent reads might also require locking, unless it is possible to perform all necessary reads in an initial system transaction, at the beginning of the business transaction.

Normally we design so that every business transaction occurs in a session. Sessions then become sequences of business transactions, from the user's point of view.

Wherever possible, let the transaction system deal with concurrency.

When there's a mismatch between the system transactions and business transactions, then you have to roll your own. The patterns include

- Optimistic Offline Lock: use optimistic concurrency control across the business transactions. Easiest to program and high liveness. Limitation is you don't know the transaction will fail until you try to commit. This can decrease user confidence in your system.
- Pessimistic Offline Lock: lock everything in advance. More difficult to program, less liveness, earlier detection of trouble.
- Coarse-Grained Lock: works with either basic approach, reducing complexity by managing a group of objects together rather than individual locks.
- Implicit Lock: business transaction developers don't have to worry about locking; the objects do it themselves (in a Layer Supertype) or the framework handles it transparently.

An Optimistic Offline Lock *prevents conflicts between concurrent business transactions by detecting a conflict and rolling back the transaction.*

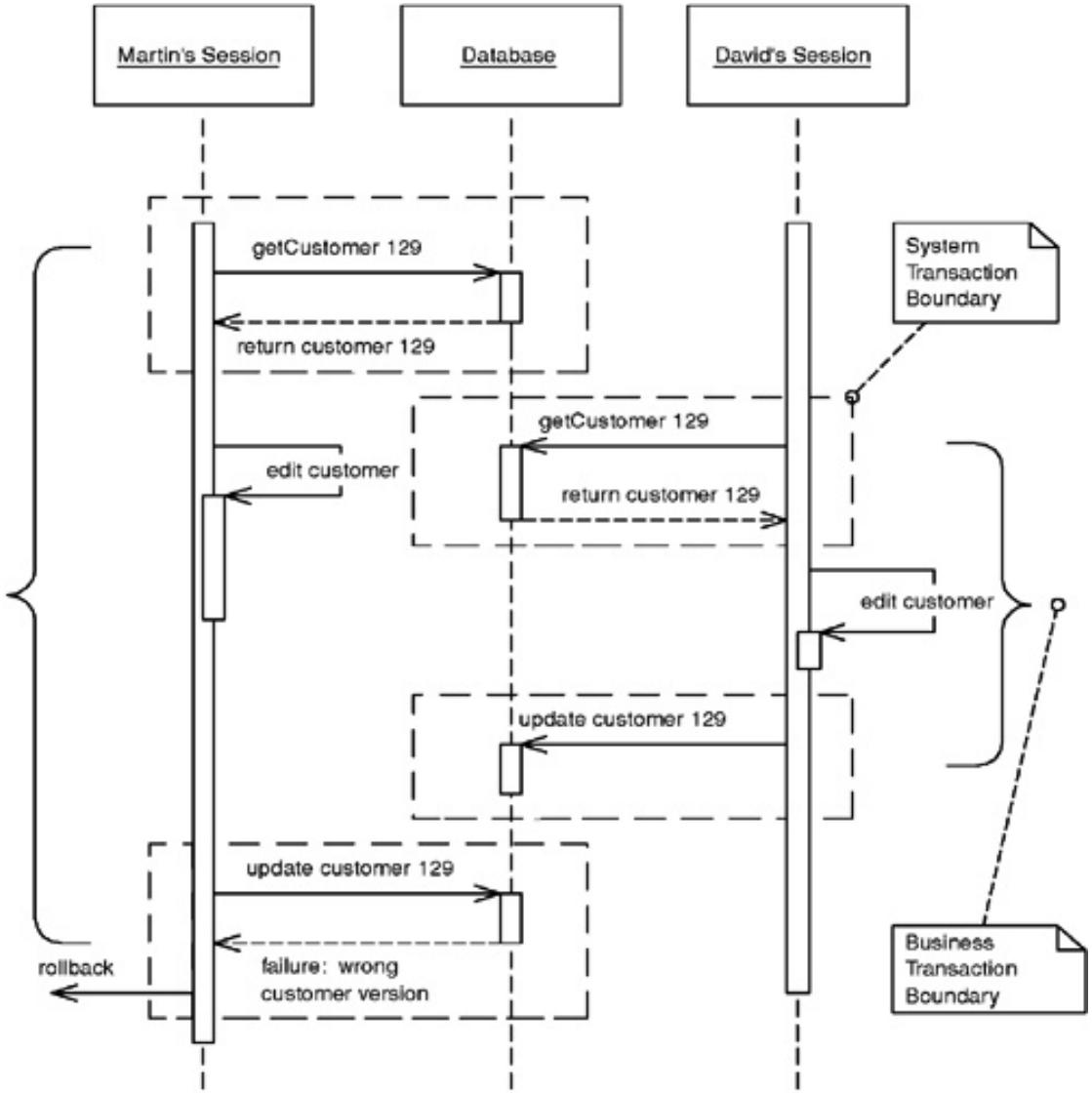
If the chance of a conflict is low, Optimistic Offline Lock is a good choice.

The basic approach:

- Associate a version number with every record.
- Increment the version number on each update.
- Check the version number before allowing an update.
- If using SQL, we can `UPDATE` with a `WHERE` clause specifying the version number — if the number of affected rows is 0, we have a conflict.

Concurrency

Optimistic Offline Lock



Fowler (2002), p. 416

A Pessimistic Offline Lock *prevents conflicts between concurrent business transactions by allowing only one business transaction at a time to access data.*

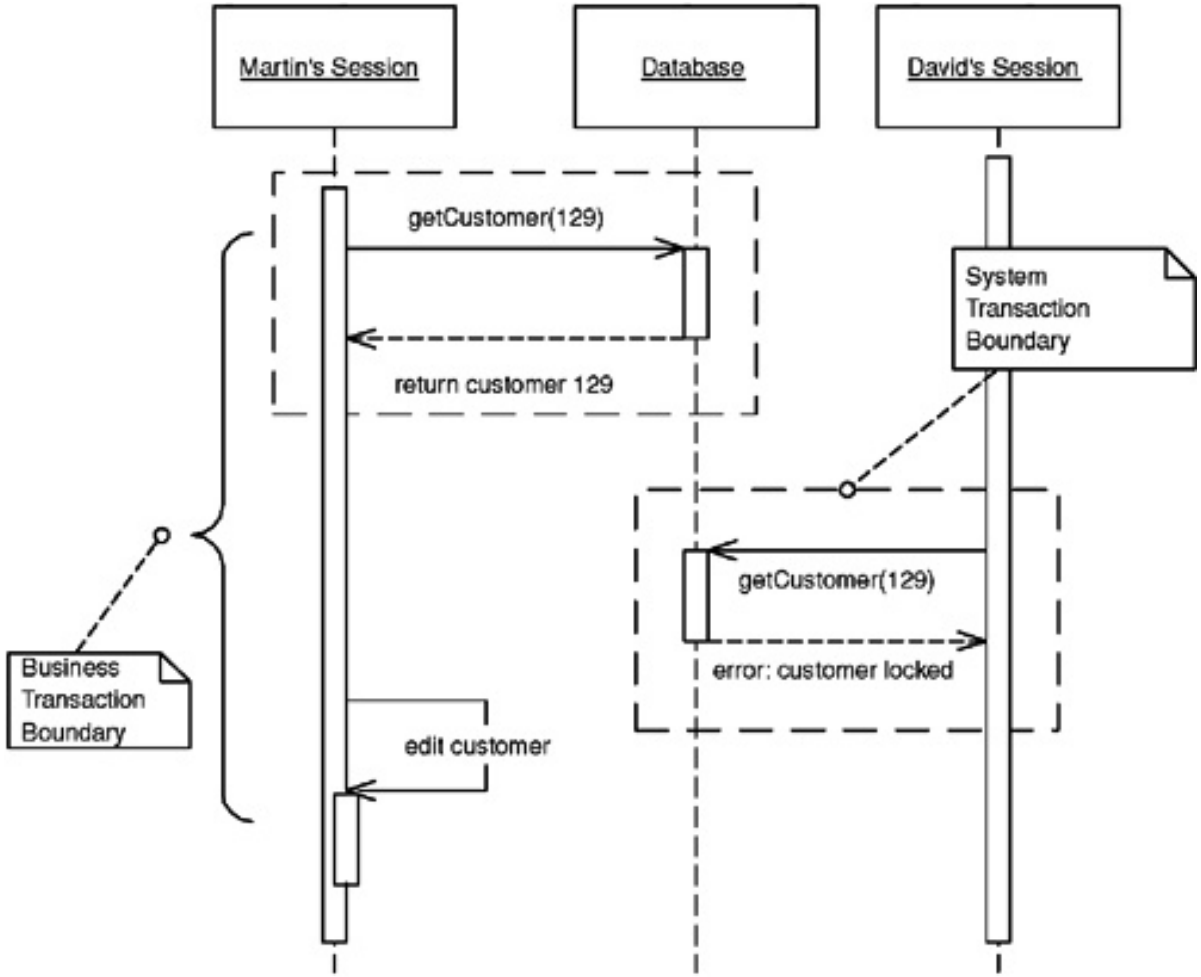
If chances of conflict are high, or rollback is too painful for your application, you need a Pessimistic Offline Lock.

Each resource you might lock corresponds to a row in a lock table, and each lock has an owner.

We write a lock manager to acquire and release locks.

We lock all resources at the beginning of the business transaction and release them at the end of the business transaction.

See Chapter 16 for implementation strategies.



Fowler (2002), p. 426

How do we handle simultaneous requests to the application server? Database transactions can't help us. Hopefully the application server environment will.

Process per session has each session running in its own process, isolated from others. This is great but uses a lot of resources.

Pooled process per request means we allocate an existing idle process to each new request. Session information can be saved at the end of each request and restored at the beginning of the request. Isolation is good, but we have to worry about releasing resources at the end of each request.

Thread per request means each request is handled by a single thread within one process. More requests with less hardware, but dangerous.

Fowler recommends process per request due to the increased isolation and equal scalability with thread per request.

You can also get good isolation even with threads if using an environment like EJB that gives your threads a private playground.

Thread per requrest needs to allocate all objects locally so as not to affect other threads. Avoid static or global variables so you don't have to synchronize them.

For global memory, use the REGISTRY pattern:

- A REGISTRY is a SINGLETON that knows how to find common objects and services.
- REGISTRY is easy to scale to multiple application server environments.

Outline

Session state

What is session state?

In a good OO design, almost all objects have state.

Query objects might be stateless. But even query objects might want to do things like keep a history of queries.

However, statelessness for the application server is a good thing:

- All state is in the database.
- It means it doesn't matter which object serves an incoming request.
- It maps well onto HTTP's statelessness.

Problem: many interactions are inherently stateful.

Example: a shopping cart in an e-commerce application. If we don't have it we can't make money!

Session state

Session state and business transactions

Record data

Long-term persistent data visible to all sessions.

Session state

State information only related to a single session

If we want to turn session state into record data, we have to commit it.

Typical structure of a long business transaction:

1. Query record data.
2. Manipulate session state.
3. Repeat 1–2 over several requests.
4. Commit.

Session state

Session state and business transactions

Should session state updates be ACID?

- Consider consistency: While we are working on a loan application, the intermediate data will not be conistent according to the business logic. We often need different consistency rules for session state and record data.
- Consider isolation: While we are performing executing the steps in the business transaction, the related record data should not change. As already discussed, in practice, we will most likely use a weak isolation model for session state.

Session state

Patterns for session state

The session state patterns:

- Client Session State stores the session data on the client.
  - For Web presentation, we use URL encoding, cookies, or hidden fields.
  - For rich client presentation, we use presentation-native objects and Data Transfer Object.
- Server Session State stores session data on the application server, either in memory or persistently in a container-managed database.
- Database Session State means mapping the session data just like normal persistent data.

Session state

Choosing a pattern for session state

The decision depends on many factors.

Do I have to implement it myself?

- All application servers support Server Session State, and many support Database Session State. This may be the deciding factor.

How much data is there?

- If a lot, Client Session State is ruled out.

Does it need to be protected?

- If yes, Client Session State can still be made to work but is probably a bad idea.

Session state

Choosing a pattern for session state

Who will use it?

- Server Session State puts the data closest to where it will be used.
- Client Session State requires reconstructing serialized objects, and Database Session State requires an extra database query.

Do we need to support clustered application servers?

- If so, we might rule out Server Session State, unless we also have server affinity, sticky sessions, or easy migration of session state.

Will we have a lot of partially completed transactions?

- If so, Client Session State might be the winner, especially if the amount of session state is small.

Do I care about crashes?

- If so, Database Session State will be most robust.

You can mix the approaches for different kinds of data. But you will always need at least Client Session State for the session ID.

Fowler prefers Server Session State, with a session ID stored client side, for most applications. However:

- Client Session State is recommended when the session data is very small.
- Database Session State is recommended for extra reliability.