DATA ACCESS

Lecture 6

PREVIOUSLY

- Modelling the business logic layer
  - Domain driven design
  - Volatility driven design

CONTENT

Hybrid Data Source Pattern
- Active Record
- Table Module

Data Access
- Gateways (DAO + DTO)
- Data Mapper

Object-Relational **Structural** Patterns

Object-Relational **Behavioral** Patterns
- Lazy Load
- Identity Map

REFERENCES

Martin Fowler et. al, Patterns of Enterprise Application Architecture, Addison Wesley, 2003 [Fowler]

Microsoft Application Architecture Guide, 2009 [MAAG]

Paulo Sousa, Instituto Superior de Engenharia do Porto, Patterns of Enterprise Applications by example

http://jayurbain.com/msoe/se380/outline.html

Sun: Core J2EE Pattern Catalog

http://www.oracle.com/technetwork/java/dataaccessobject-138824.html

ORGANIZING THE BUSINESS LOGIC ACCORDING TO FOWLER

Key architectural decisions, which influence structure of other layers.

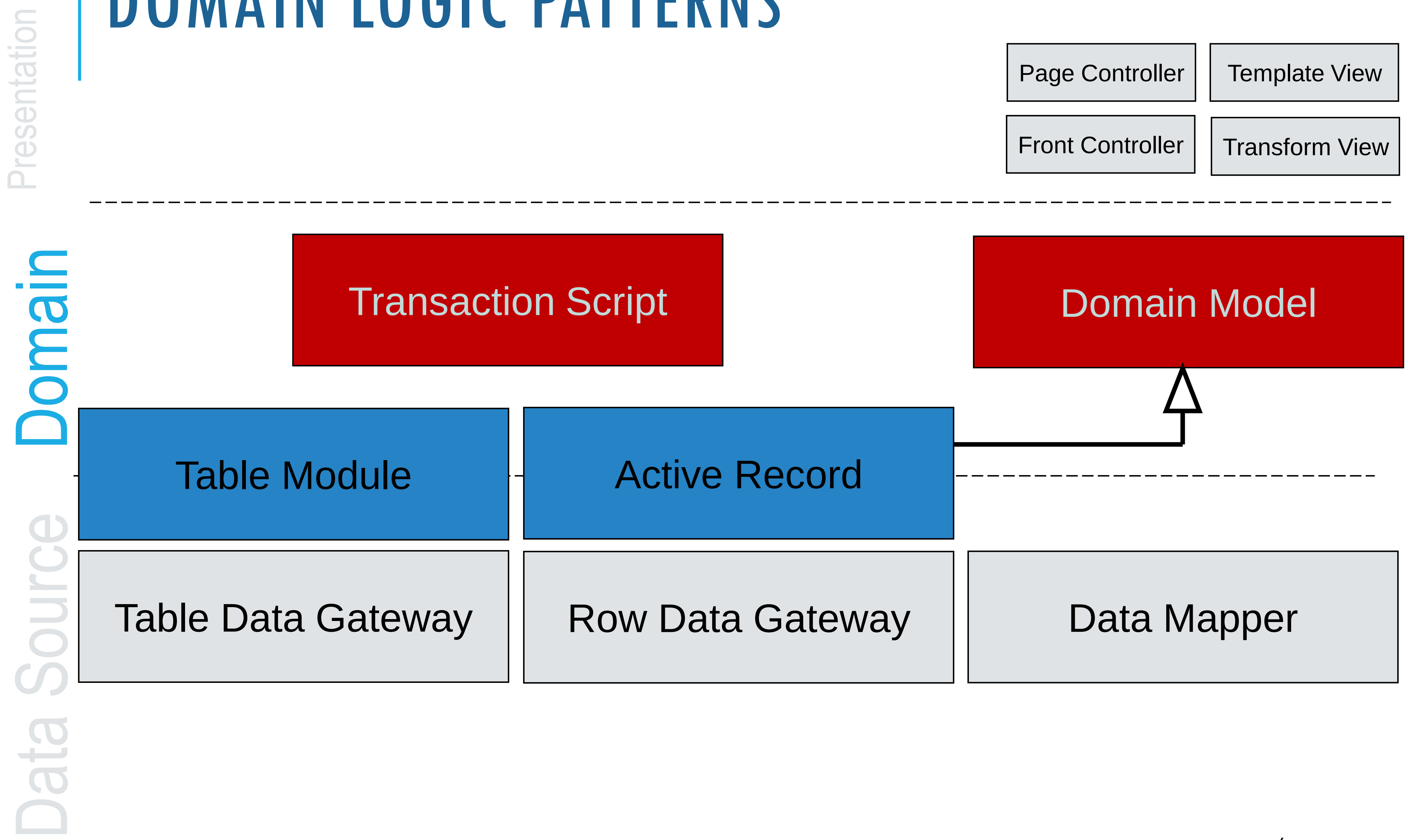
Pure patterns

- Transaction Script (**functional** driven)
- Domain Model (**domain** driven)

Hybrid patterns

- Active Record
- Table Module (**data** driven)

DOMAIN LOGIC PATTERNS



TRANSACTION SCRIPT

- A TS organizes the business logic primarily as a single procedure where each procedure handles a single request from the presentation. [Fowler]
- The TS may make calls directly to the DB or through a thin DB wrapper.
- Think of a script for a use case or business transaction.

Remember “use-case controller”?

TRANSACTION SCRIPT

… is essentially a procedure that takes the

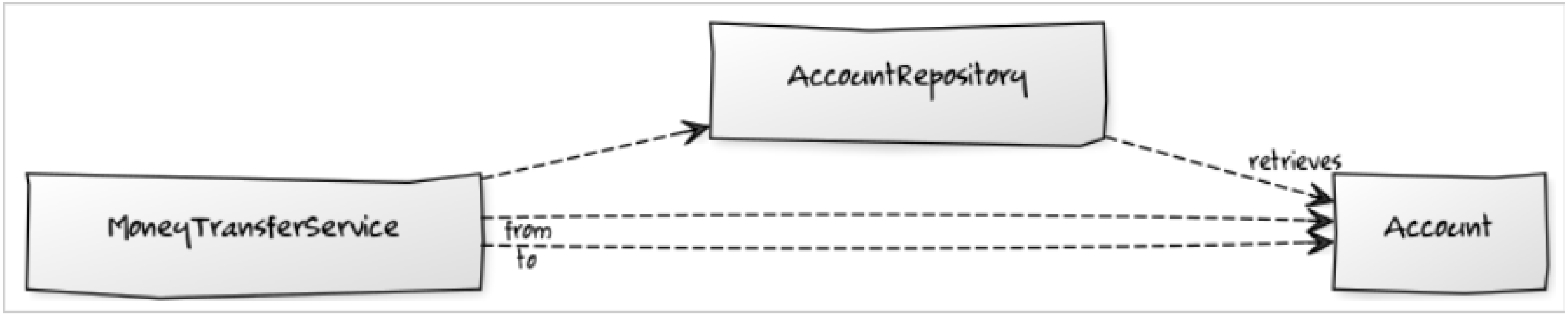
- input from the presentation,
- processes it with validations and calculations,
- stores data in the database,
- (invokes any operations from other systems, and)
- replies with more data to the presentation perhaps doing more calculation to help organize and format the reply data.

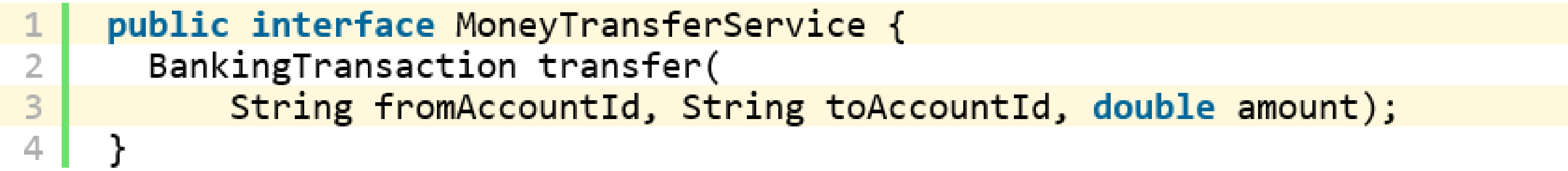
[Fowler]

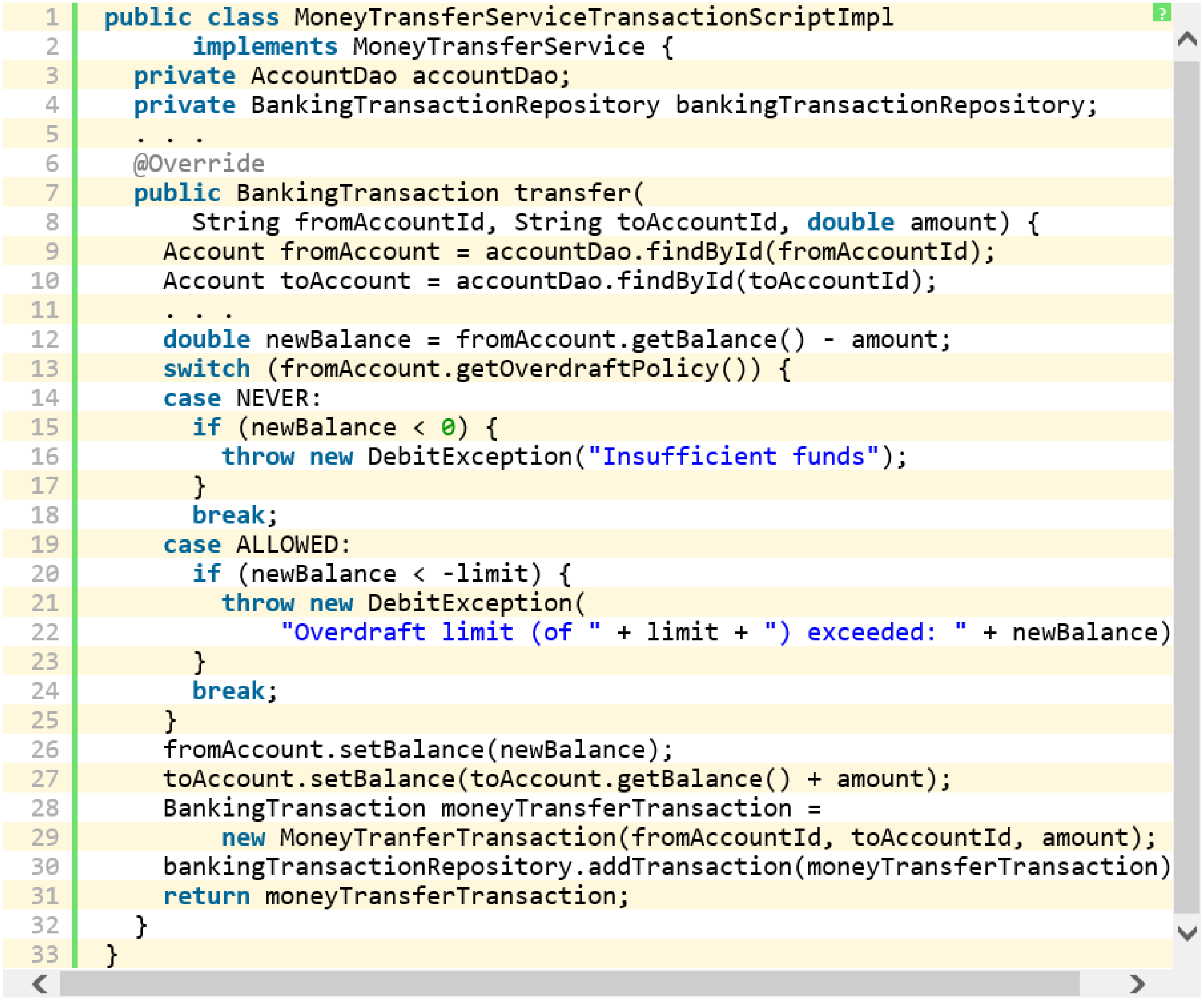
EXAMPLE

Banking application

Money transfer functionality




```java
public interface MoneyTransferService {
  BankingTransaction transfer(
      String fromAccountId, String toAccountId, double amount);
}
```



```java
public class MoneyTransferServiceTransactionScriptImpl
   implements MoneyTransferService {
 private AccountDao accountDao;
 private BankingTransactionRepository bankingTransactionRepository;
 . . .
 @Override
 public BankingTransaction transfer(
   String fromAccountId, String toAccountId, double amount) {
  Account fromAccount = accountDao.findById(fromAccountId);
  Account toAccount = accountDao.findById(toAccountId);
  . . .
  double newBalance = fromAccount.getBalance() - amount;
  switch (fromAccount.getOverdraftPolicy()) {
  case NEVER:
   if (newBalance < 0) {
    throw new DebitException("Insufficient funds");
   }
   break;
  case ALLOWED:
   if (newBalance < -limit) {
    throw new DebitException(
      "Overdraft limit (of " + limit + ") exceeded: " + newBalance)
   }
   break;
  }
  fromAccount.setBalance(newBalance);
  toAccount.setBalance(toAccount.getBalance() + amount);
  BankingTransaction moneyTransferTransaction =
    new MoneyTranferTransaction(fromAccountId, toAccountId, amount);
  bankingTransactionRepository.addTransaction(moneyTransferTransaction)
  return moneyTransferTransaction;
 }
}
```

```java
public enum OverdraftPolicy {
  NEVER, ALLOWED
}
```

```java
// @Entity
public class Account {
  // @Id
  private String id;
  private double balance;
  private OverdraftPolicy overdraftPolicy;
  . . .
  public String getId() { return id; }
  public void setId(String id) { this.id = id; }
  public double getBalance() { return balance; }
  public void setBalance(double balance) { this.balance = balance; }
  public OverdraftPolicy getOverdraftPolicy() { return overdraftPolicy; }
  public void setOverdraftPolicy(OverdraftPolicy overdraftPolicy) {
    this.overdraftPolicy = overdraftPolicy;
  }
}
```

ANALYSIS

Strengths

- Simplicity

Weaknesses

- complicated transaction logic
- duplicated logic

DOMAIN MODEL (EA PATTERN)

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

A DM creates a web of interconnected objects, where each object represents some meaningful individual, whether as large as a corporation or as small as a single line in an order form.

DOMAIN MODEL (EA PATTERN)

Realization (via design classes) of
UML Domain Model (conceptual classes).

- E.g. person, book, shopping cart, task, sale item, …

Domain Model classes contain logic for handling validations and calculations.

- E.g. a shipment object calculates the shipping charge for a delivery.

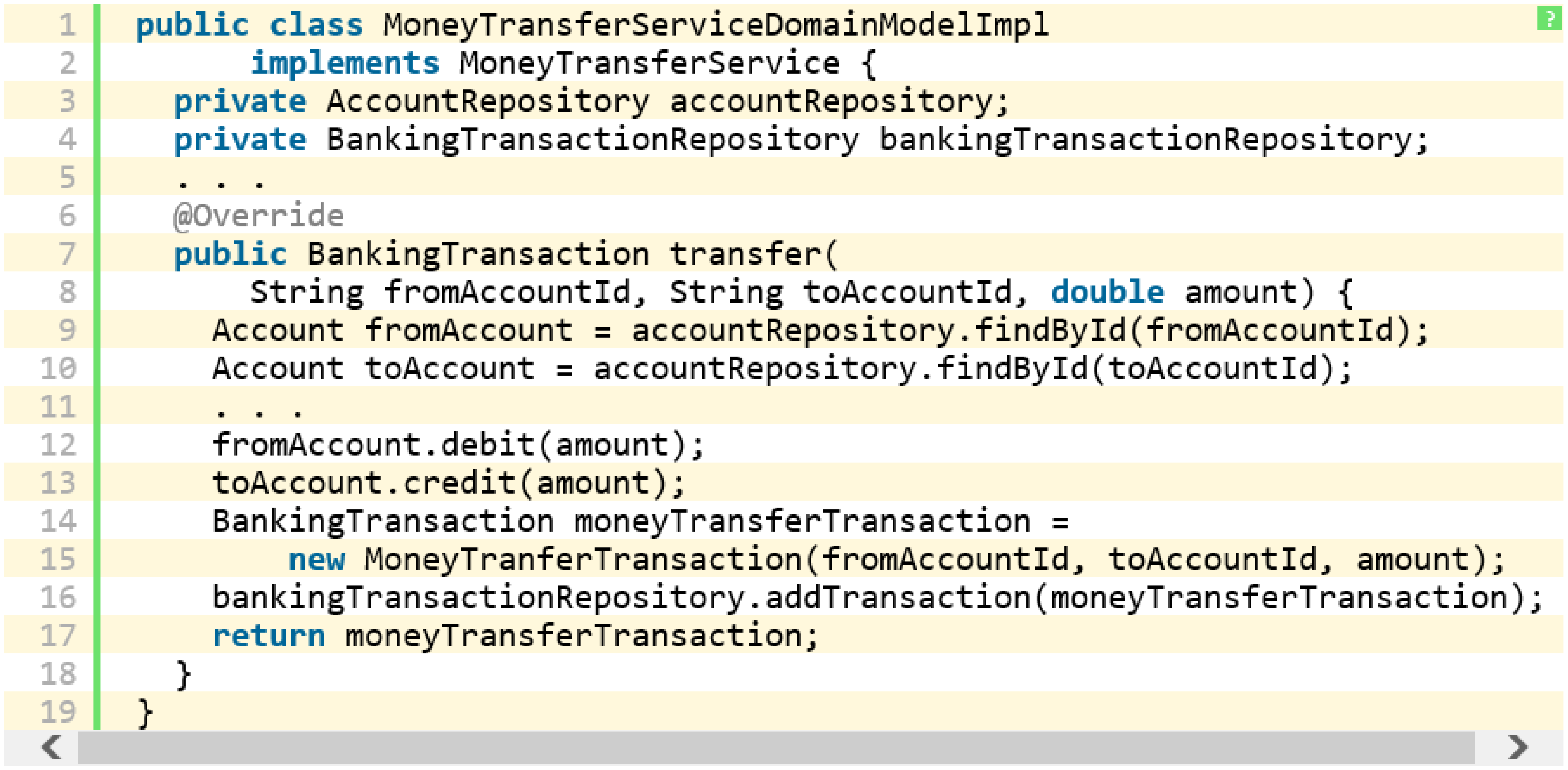
DM FEATURES

Business logic is organized as an OO model of the domain

- Describes both data and behavior
- Different from database model
  - Process, multi-valued attributes, inheritance, design patterns
  - Harder to map to the database

Risk of bloated domain objects

MONEY TRANSFER EXAMPLE



```java
public class MoneyTransferServiceDomainModelImpl
      implements MoneyTransferService {
  private AccountRepository accountRepository;
  private BankingTransactionRepository bankingTransactionRepository;
  . . .
  @Override
  public BankingTransaction transfer(
      String fromAccountId, String toAccountId, double amount) {
    Account fromAccount = accountRepository.findById(fromAccountId);
    Account toAccount = accountRepository.findById(toAccountId);
    . . .
    fromAccount.debit(amount);
    toAccount.credit(amount);
    BankingTransaction moneyTransferTransaction =
        new MoneyTranferTransaction(fromAccountId, toAccountId, amount);
    bankingTransactionRepository.addTransaction(moneyTransferTransaction);
    return moneyTransferTransaction;
  }
}
```

```java
// @Entity
public class Account {
  // @Id
  private String id;
  private double balance;
  private OverdraftPolicy overdraftPolicy;
  . . .
  public double balance() { return balance; }
  public void debit(double amount) {
    this.overdraftPolicy.preDebit(this, amount);
    this.balance = this.balance - amount;
    this.overdraftPolicy.postDebit(this, amount);
  }
  public void credit(double amount) {
    this.balance = this.balance + amount;
  }
}
```

```java
public interface OverdraftPolicy {
  void preDebit(Account account, double amount);
  void postDebit(Account account, double amount);
}
```

```java
public class NoOverdraftAllowed implements OverdraftPolicy {
  public void preDebit(Account account, double amount) {
    double newBalance = account.balance() - amount;
    if (newBalance < 0) {
      throw new DebitException("Insufficient funds");
    }
  }
  public void postDebit(Account account, double amount) {
  }
}
```

```java
public class LimitedOverdraft implements OverdraftPolicy {
  private double limit;
  . . .
  public void preDebit(Account account, double amount) {
    double newBalance = account.balance() - amount;
    if (newBalance < -limit) {
      throw new DebitException(
          "Overdraft limit (of " + limit + ") exceeded: " + newBalance);
    }
  }
  public void postDebit(Account account, double amount) {
  }
}
```

CHOOSING A BUSINESS LOGIC PATTERN

Which one to choose?

- Influenced by the complexity of domain logic.

Application is simple access to data sources

→ Transaction Script, (or Active Record, Table Module)

Significant amount of business logic

→ Domain Model

TS is simpler:

- Easier and quicker to develop and maintain.
- But can lead to duplication in logic / code.

DM – difficult access to relational DB

DATA ACCESS

- Communication between BLL and Data source:
  - Retrieve data from DB
  - Save data to DB
- Responsibilities:
  - Wrap the access to the DB
    - **Connect** to DB
    - **Execute** queries
    - **Map** data structure to domain model structure
  - **Store** the raw data
    - Set of records
    - Record

MONOLITHIC APPROACH

All responsibilities in one class

- Responsibilities:
  - Wrap the access to the DB
    - Connect to DB
    - Execute queries

    + Business Logic
  - Store the raw data
    - Set of records => **Table Module**
    - Record =>**Active Record**

TABLE MODULE

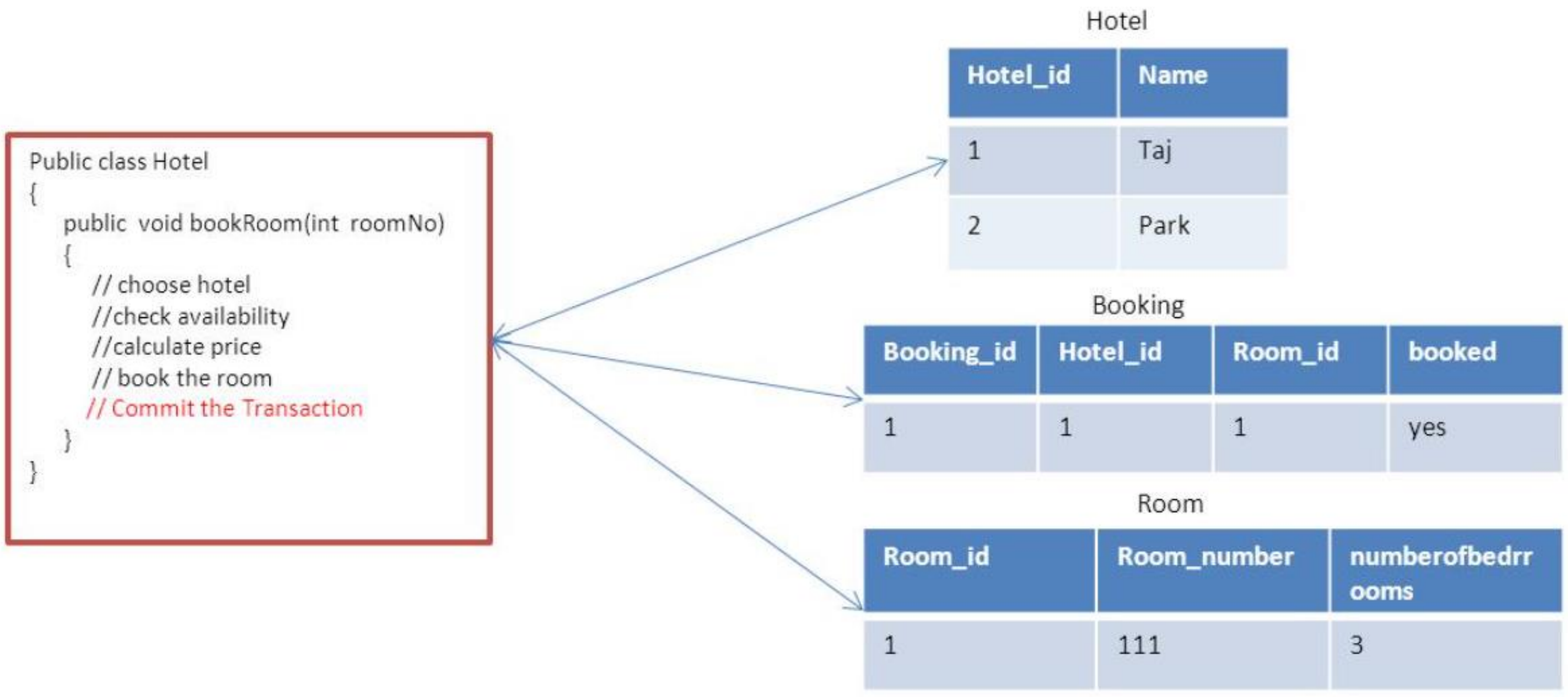
Provides a single object for all the behavior on a table

Organizes domain logic with **one class per table**

Table Module has no notion of an identity for the objects that it's working with

$\Rightarrow$Id references are necessary

TRANSACTION SCRIPT VS TABLE MODULE



```
Public class Hotel
{
    public  void bookRoom(int  roomNo)
    {
        // choose hotel
        //check availability
        //calculate price
        // book the room
       // Commit the Transaction
    }
}
```

Hotel

Hotel_id	Name
1	Taj
2	Park

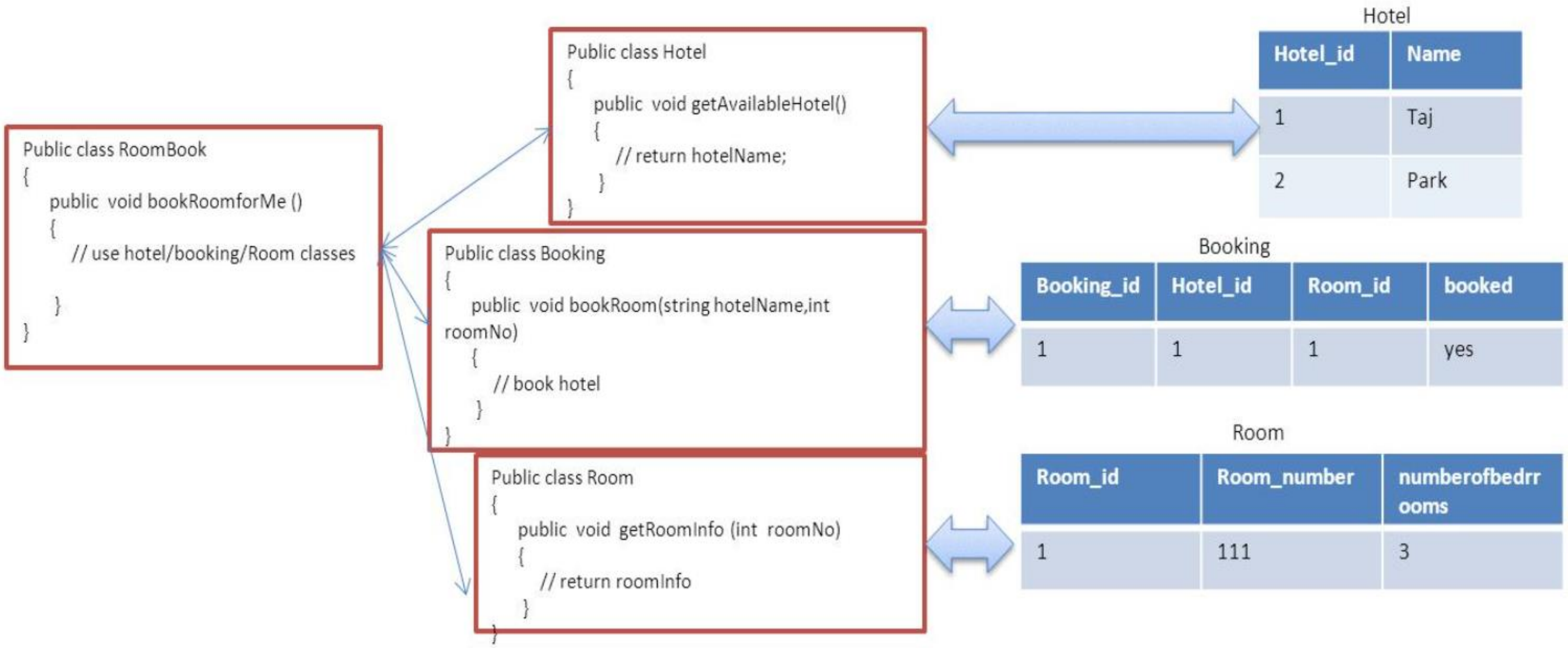
Booking

Booking_id	Hotel_id	Room_id	booked
1	1	1	yes

Room

Room_id	Room_number	numberofbedrrooms
1	111	3

TS VS TM



```
Public class RoomBook
{
    public  void bookRoomforMe ()
    {
       // use hotel/booking/Room classes

    }
}
```

```
Public class Hotel
{
    public  void getAvailableHotel()
    {
       // return hotelName;
     }
}
```

```
Public class Booking
{
    public  void bookRoom(string hotelName,int
roomNo)
    {
       // book hotel
    }
}
```

```
Public class Room
{
    public  void getRoomInfo (int  roomNo)
    {
       // return roomInfo
    }
}
```

Hotel

Hotel_id	Name
1	Taj
2	Park

Booking

Booking_id	Hotel_id	Room_id	booked
1	1	1	yes

Room

Room_id	Room_number	numberofbedrrooms
1	111	3

ACTIVE RECORD

Fowler: An object that wraps a record data structure of an external resource, such as a row in a database table, and adds some domain logic to that object.

An AR object carries both data and behavior.

Active Record is a Domain Model in which the classes match very closely the record structure of the underlying database.

CLASS OPERATIONS

- **construct** an instance of the *Active Record* from a SQL result set row
- **construct** a new instance for later insertion into the table
- static **finder** methods to wrap commonly used SQL queries and return *Active Record* objects
- methods to **update** the database and insert into the database with the data in the *Active Record*
- **getting** and **setting** methods for the fields
- methods that implement some pieces of **business logic**

IMPLEMENTATION

```
class Person{
 private String lastName;
 private String firstName;
 private int numberOfDependents;
…}


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

FIND + LOAD AN OBJECT

```
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) Repository.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); }
}
```

```
public static Person load(ResultSet rs) throws
 SQLException
 {
      Long id = new Long(rs.getLong(1));
      Person result = (Person) Repository.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);
      Repository.addPerson(result);
      return result;
}
```

SEPARATING RESPONSIBILITIES

Responsibilities:

- Wrap the access to the DB
  - Connect to DB
  - Execute queries

  } **Table Data Gateway, Row Data Finder, DAO**
  - Map DB structure to BL structure => **Data Mapper**

- Store the data
  - Set of records => **DTO, RecordSet**
  - Record => **DTO, Row Data Gateway**

GATEWAY

- Definition
  - *An object that encapsulates access to an external system or resource*
- Wise to separate SQL (and other forms of data access, query language) from domain logic, and place it in separate classes.
  - Separation of concerns.
- Common technique is to define **a class which maps exactly to a table** in the database => the gateways and the tables are thus *isomorphic*
- Contains **all the database mapping** code for an application, that is all the SQL for CRUD operations
- Contains **no domain logic**

TABLE DATA GATEWAY

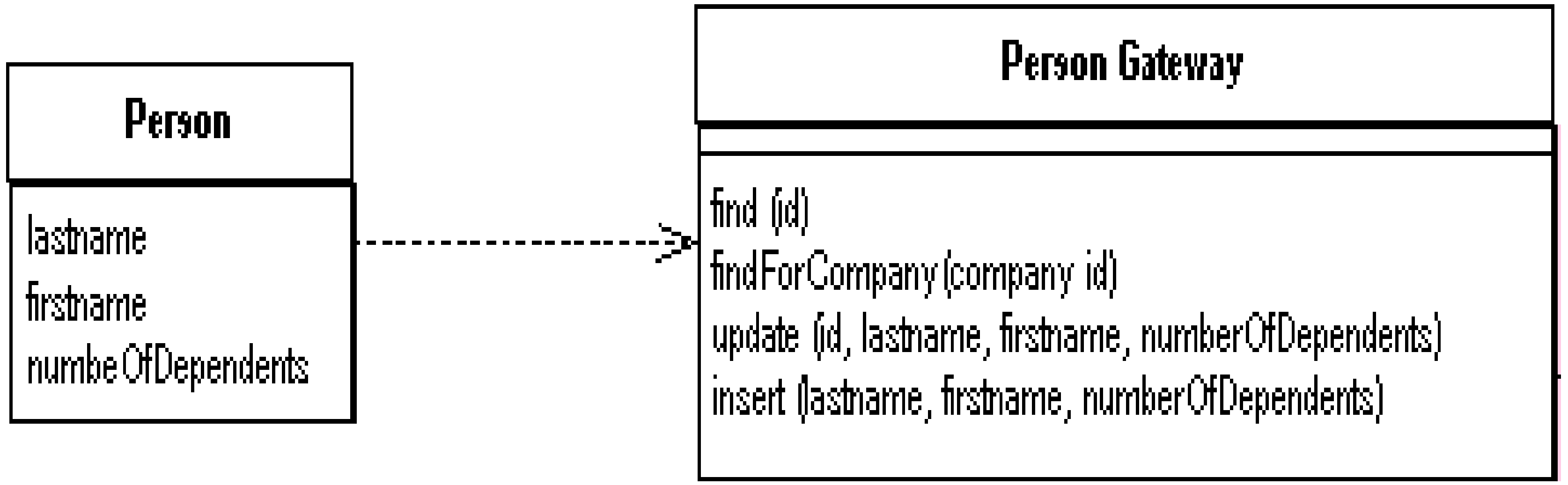
An object that acts as a gateway to a database table. One instance handles all the rows in the table. [Fowler]

Generic data structure of tables and rows that mimics the tabular nature of a DB. Can be used in many parts of application.

- Needs a single instance for ***a set of*** rows.
- Needs caching strategy, cursors to access database, manage result set.

Common to have GUI tools that work directly with record set

TDG

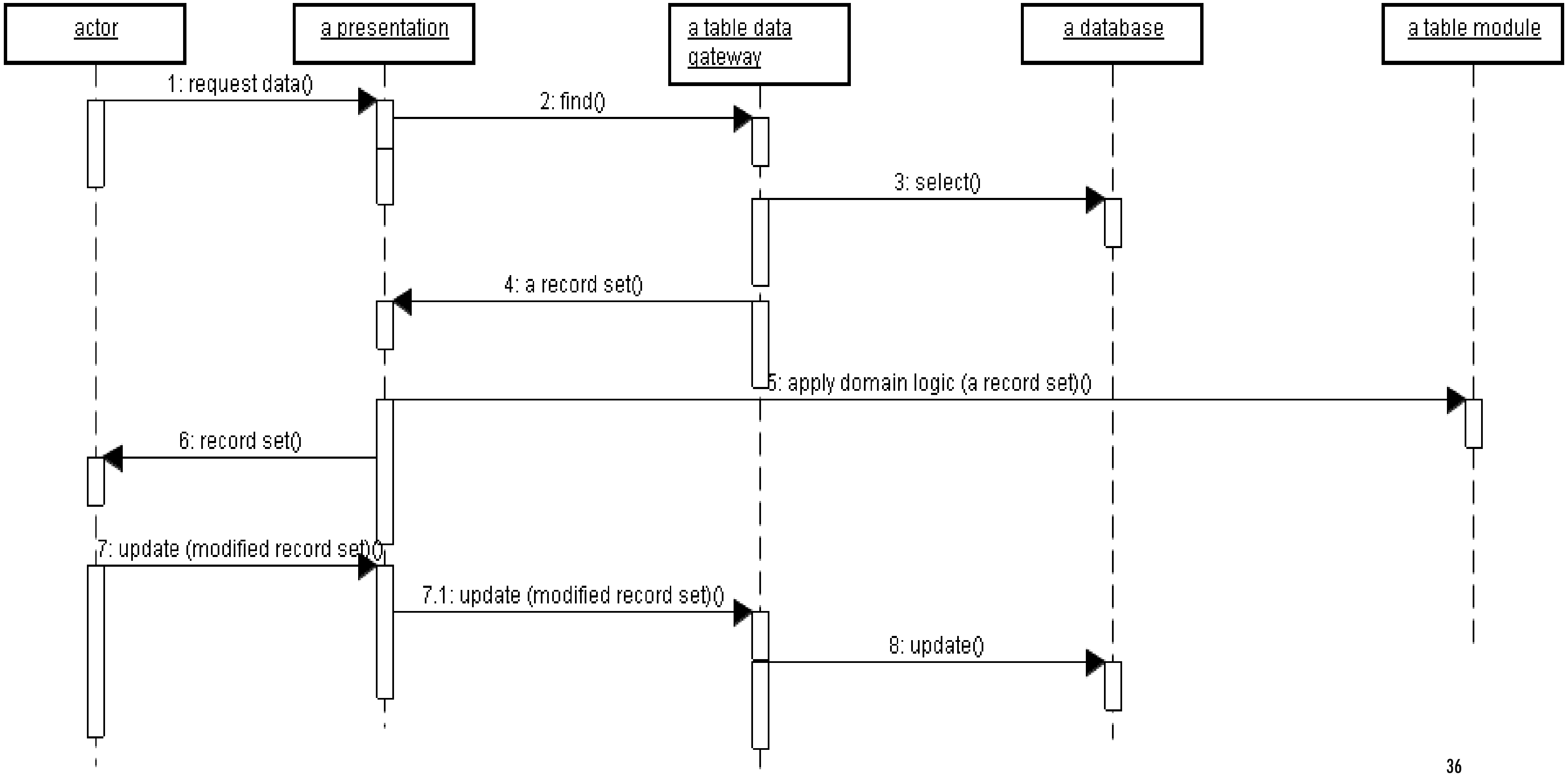

Person
lastname
firstname
numbeOfDependents
Person Gateway
find (id)
findForCompany (company id)
update (id, lastname, firstname, numberOfDependents)
insert (lastname, firstname, numberOfDependents)

- find(id) : RecordSet
- findWithLastName(String) : RecordSet
- Iterate through RecordSet
- Update, delete, insert, …

FEATURES

- No attributes
- Just CRUD methods
- Challenge: how it returns information from a query ?
  - Data Transfer Object (DTO)
  - RecordSet
- Goes well with Table Module
- Suitable for Transaction Scripts

SEPARATING THE RESPONSIBILITIES



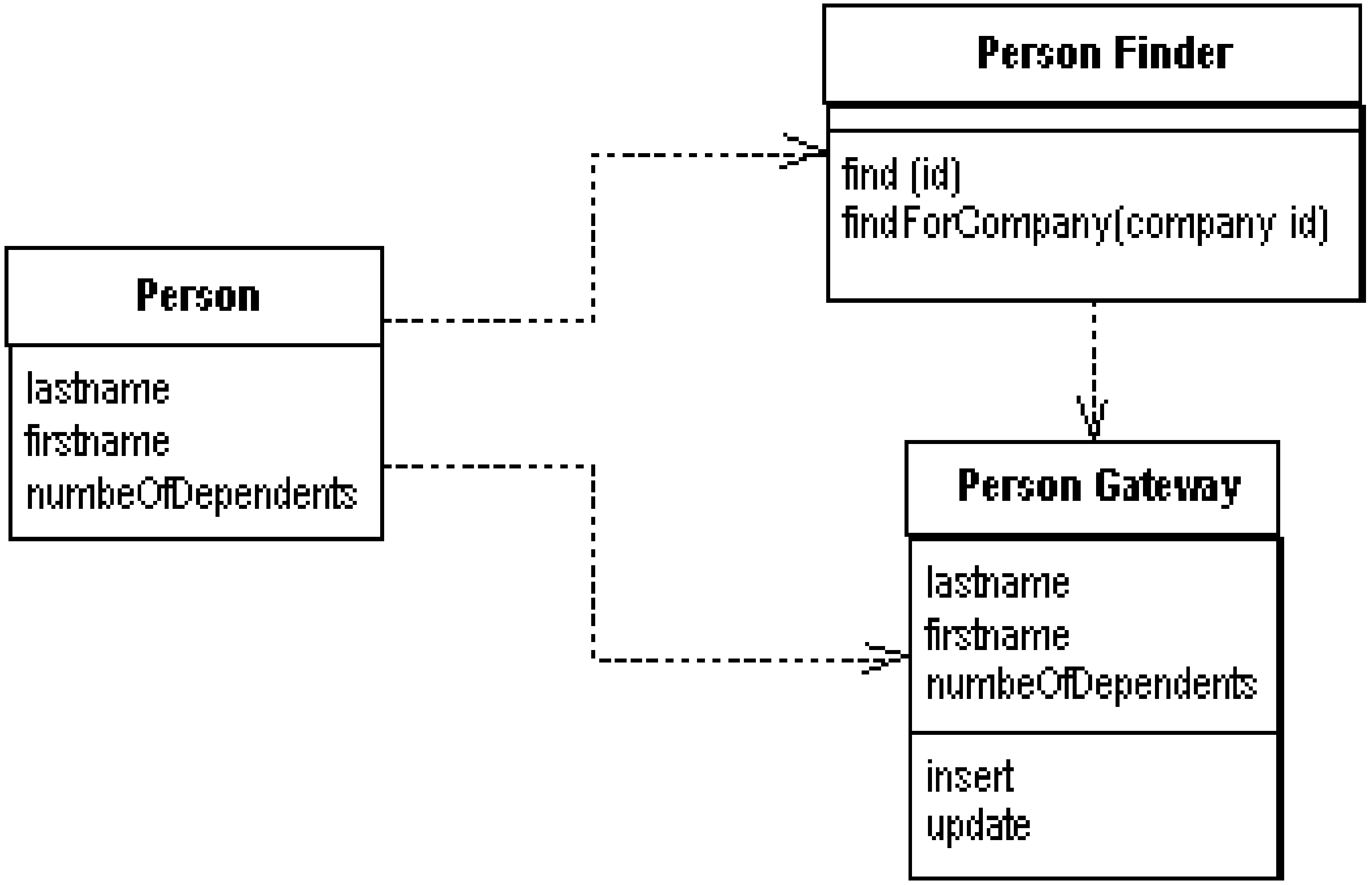
ROW DATA GATEWAY

- An object that acts as a **single record** in the data source
- Allows in memory storage of object instance without need to access DB.
- Needs an instance of an object for each row.
- Typical approach for *object-relationa*l mapping tools, e.g., Hibernate.

Fowler RDG combines two responsibilities:

- Class …Finder with find(id) - gateway method which returns the 'object' (i.e. SELECT statements)
- Class …Gateway which stores the 'object' data

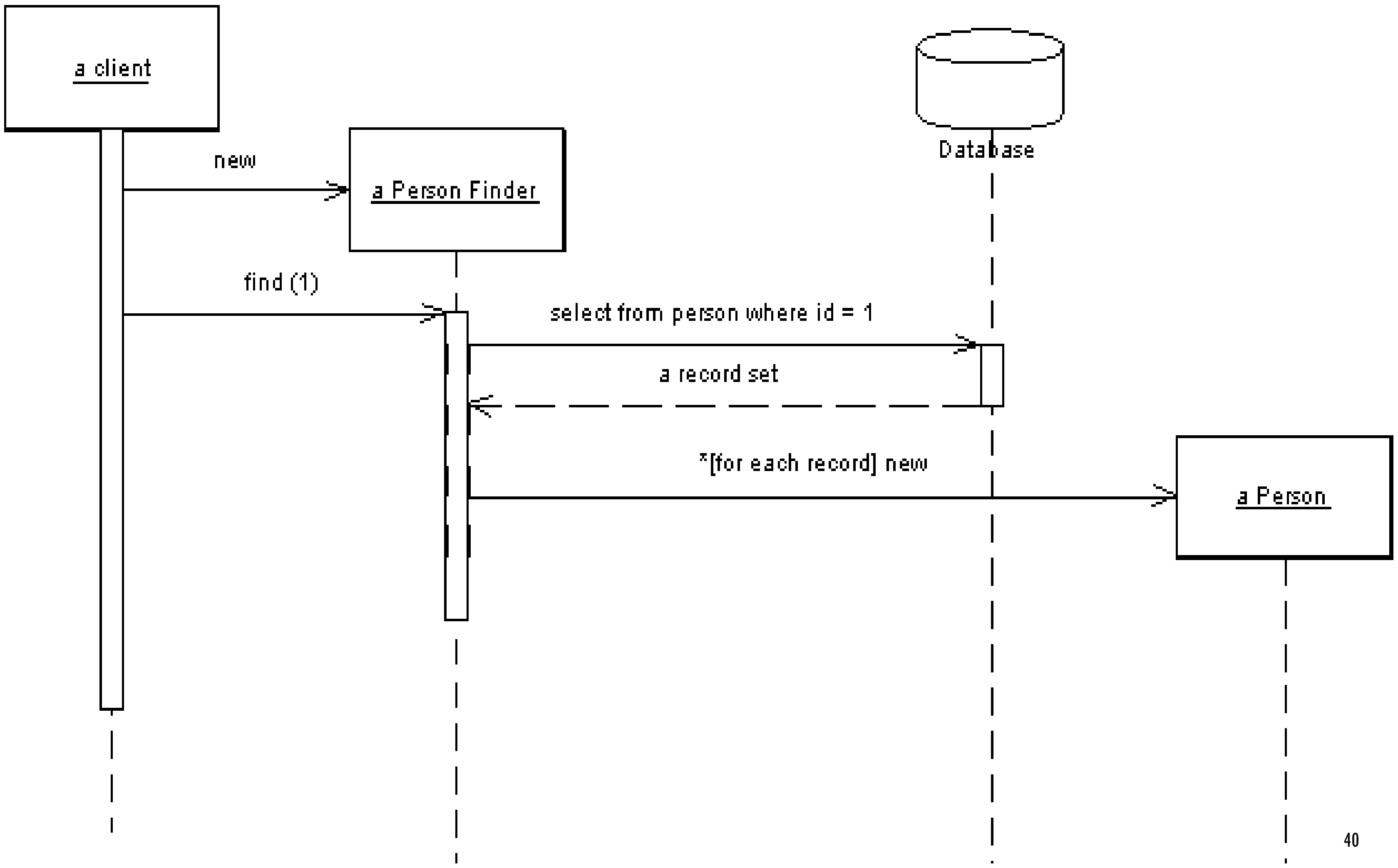
ROW DATA GATEWAY



HOW IT WORKS?

- Separate data access code from business logic
- Type conversion from the data source types to the in-memory types
- Works particularly well for Transaction Scripts
- Where to put the find operations that generate the *Row Data* ?
  - separate finder objects
  - each table in a relational database will have:
    - one finder class
    - one gateway class for the results

RDG BEHAVIOUR



IMPLEMENTATION

```
class PersonGateway...
{
 private String lastName;
 private String firstName;
 private int
numberOfDependents;

public String getLastName()
{ return lastName; }

 public void setLastName(String
lastName)
 { this.lastName = lastName; }
…
```

```
public void insert() {..}
public void update() {…}
public static PersonGateway
 load(ResultSet rs) {…}
 ..
}
```

PERSONFINDER

```
class PersonFinder...

 private final static String findStatementString = "SELECT id,
 lastname, firstname, number_of_dependents " + " from people " +
 " WHERE id = ?";

public PersonGateway find(Long id)
 {
       PersonGateway result =
 PersonGateway)Registry.getPerson(id);
       if (result != null) return result;
       try
       {
       PreparedStatement findStatement =
 DB.prepare(findStatementString);
       findStatement.setLong(1, id.longValue());
       ResultSet rs = findStatement.executeQuery();
       rs.next();
       result = PersonGateway.load(rs);
       return result;
       } catch (SQLException e) { throw new
 ApplicationException(e);
 }
```

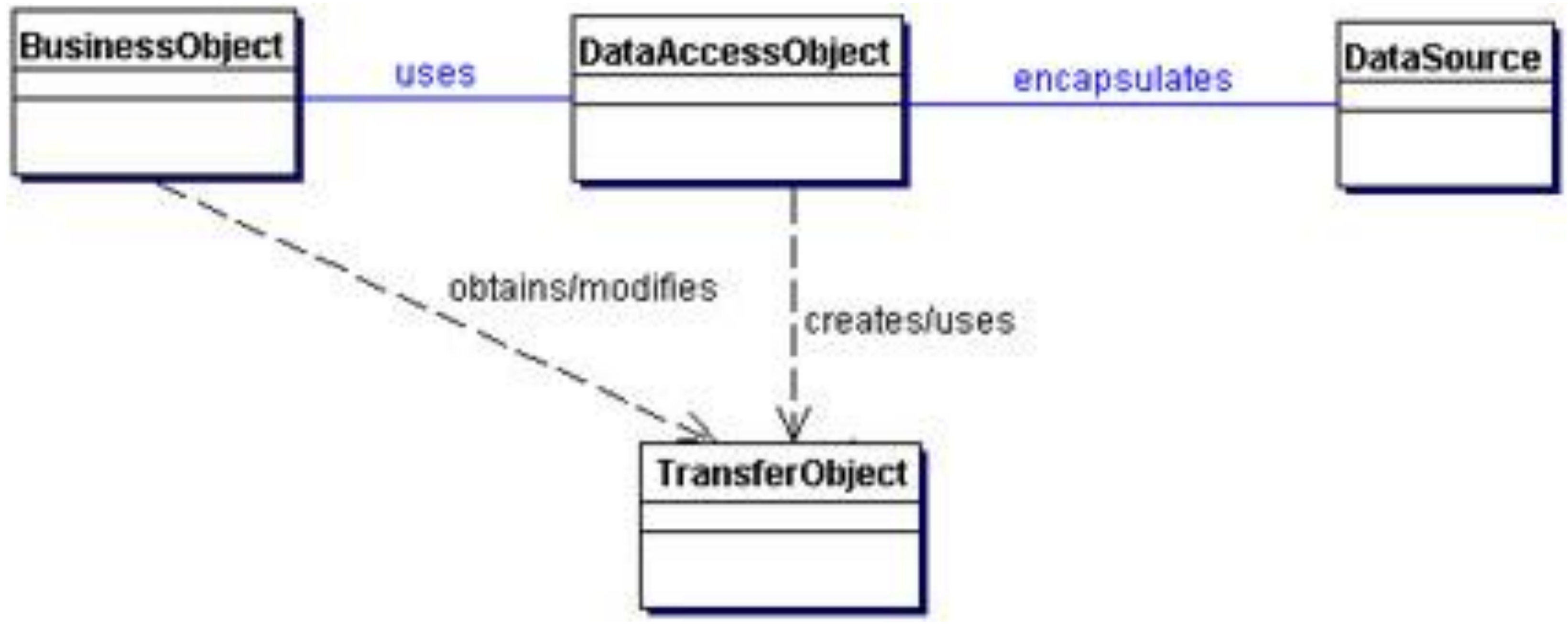
CLASS PERSONGATEWAY

```
private static final 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); } }
  …
```

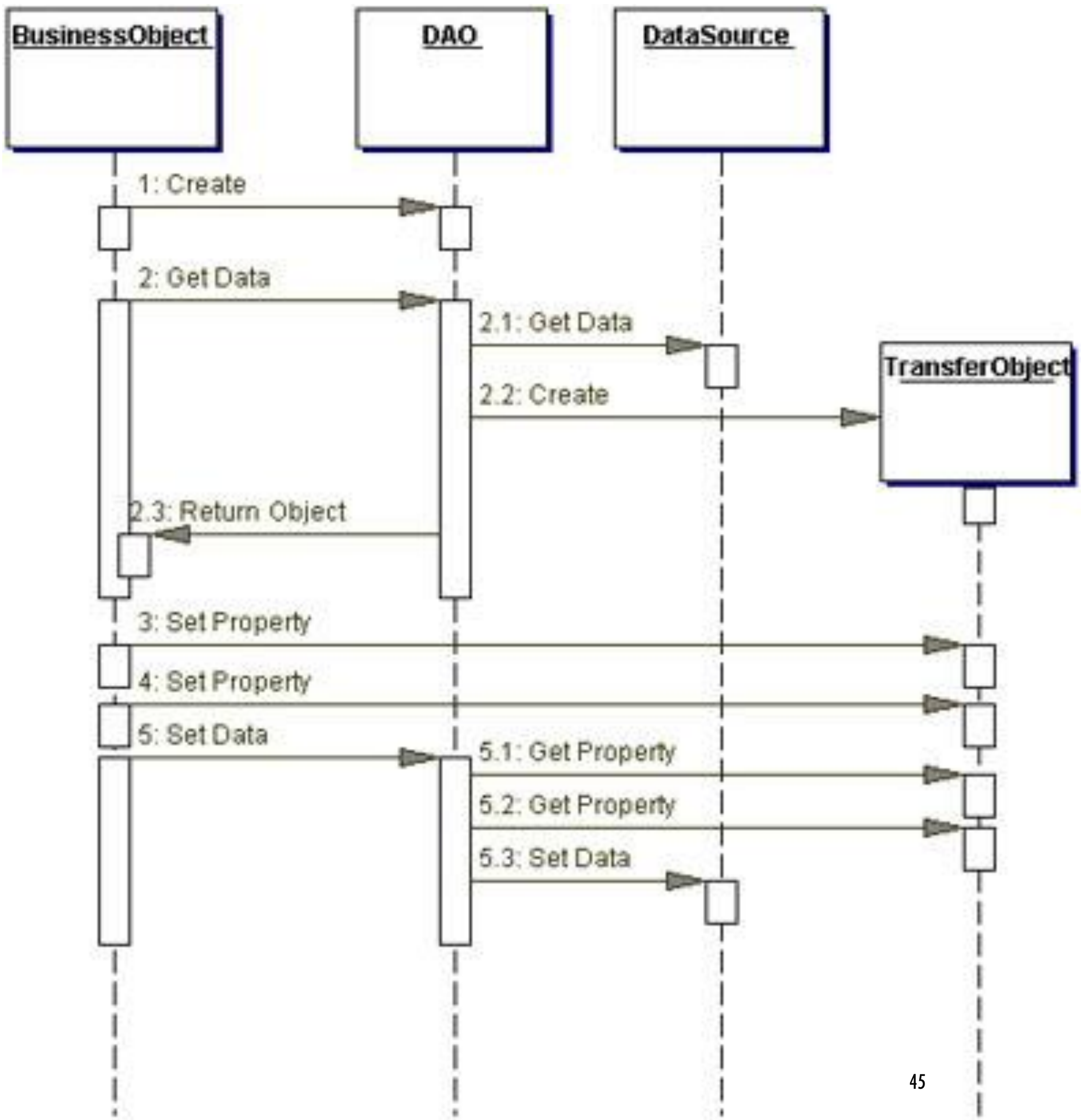
DAO PATTERN

Intent: Abstract and Encapsulate all access to the data source



Sun Developer Network - Core J2EE Patterns
http://www.oracle.com/technetwork/java/dataaccessobject-138824.html

HOW IT WORKS

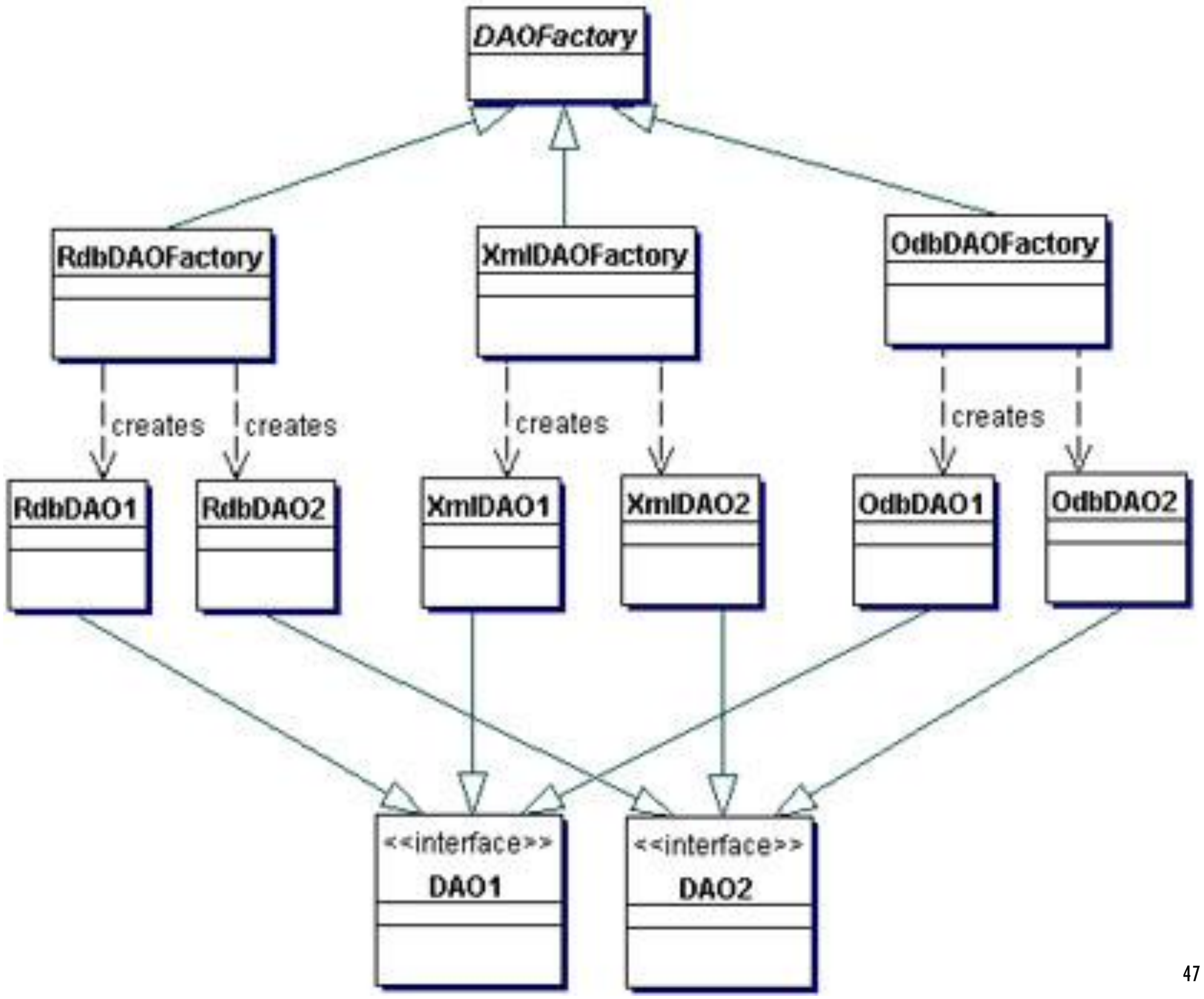

BusinessObject
DAO
DataSource
TransferObject
1: Create
2: Get Data
2.1: Get Data
2.2: Create
2.3: Return Object
3: Set Property
4: Set Property
5: Set Data
5.1: Get Property
5.2: Get Property
5.3: Set Data

HOW TO GET THE DAO

Strategies to get DAO’s:

- Automatic DAO Code Generation Strategy
  - Metadata Mapping applied
  - JPA Hibernate
- Factory for Data Access Objects Strategy
  - Use **Abstract Factory**: When the underlying storage is subject to change from one implementation to another, this strategy may be implemented using the Abstract Factory pattern.

ABSTRACT FACTORY



DATA TRANSFER OBJECT (DTO)

- An object that carries data between processes in order to reduce the number of method calls.
- Usually a set of attributes + setters and getters
- Usually aggregates data from all the server objects that the remote object is likely to want to get data from.
- DTOs are usually structured in tree structures (based on composition) containing primitive types and other DTOs
- Structured around the needs of a particular client (i.e. corresponding to web pages/GUI screens).

DESIGN DECISIONS

- Keep the DTO structures simple – they have to be serializable
- Use one DTO for a whole interaction vs. different ones for each request?
- Use one DTO for both request and response vs. different ones for each?
- DTO can be a Record Set
- DTOs and domain objects might need mapping

IMPLEMENTATION DISCUSSION

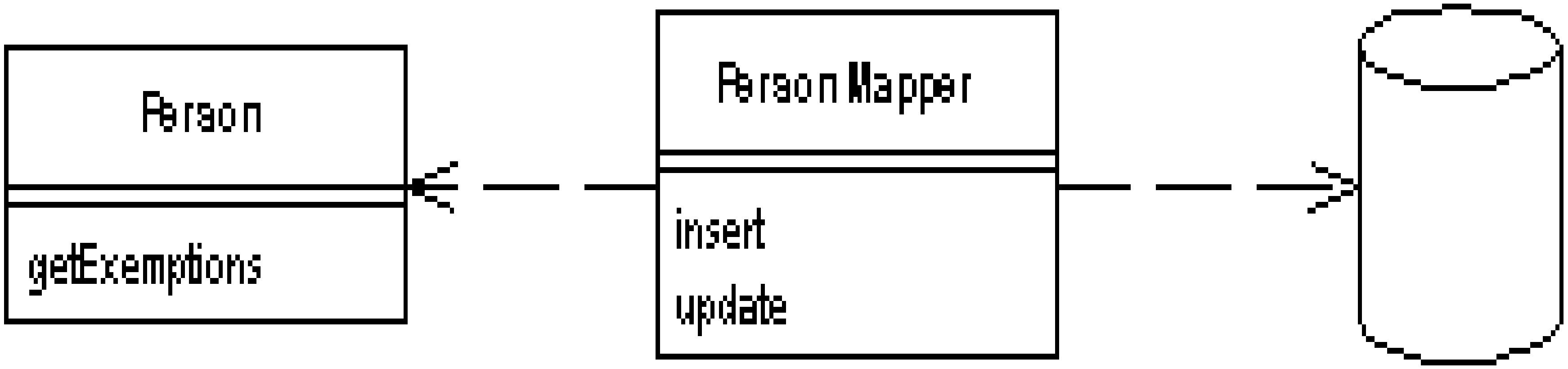
- Explicit (manual) coding
  - Needs more work and understanding
  - Flexibility related to structure
  - Better performance

- Rely on frameworks
  - Automated
  - Impact on performance
  - Constraints related to structure

DATA MAPPERS

Acts as an intermediary between Domain Models and the database.

Allows Domain Models and Data Source classes to be independent of each other



DATA MAPPER LAYER …

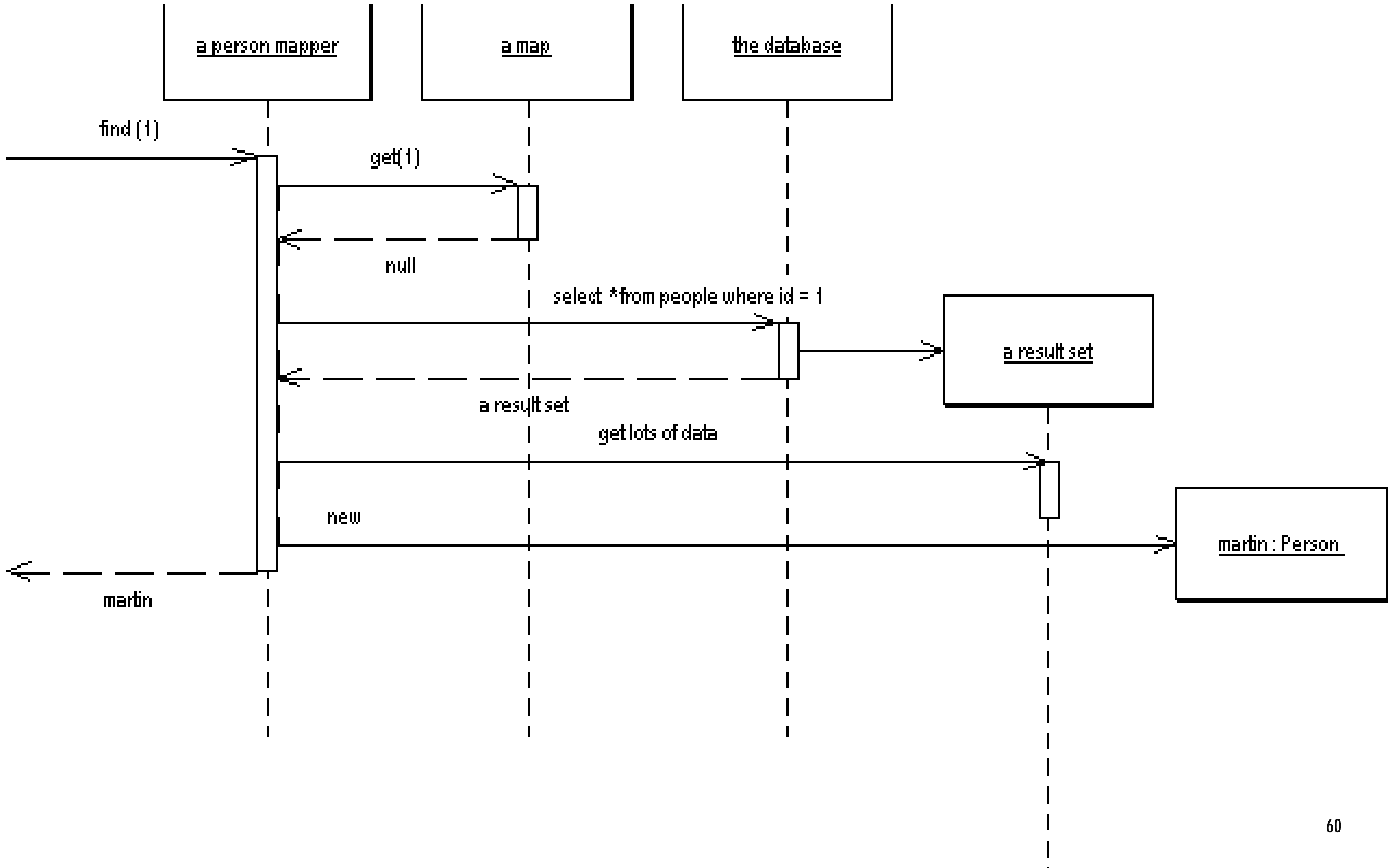
Can either

- Access the database itself, or
- Make use of a Table Data Gateway/DAO.

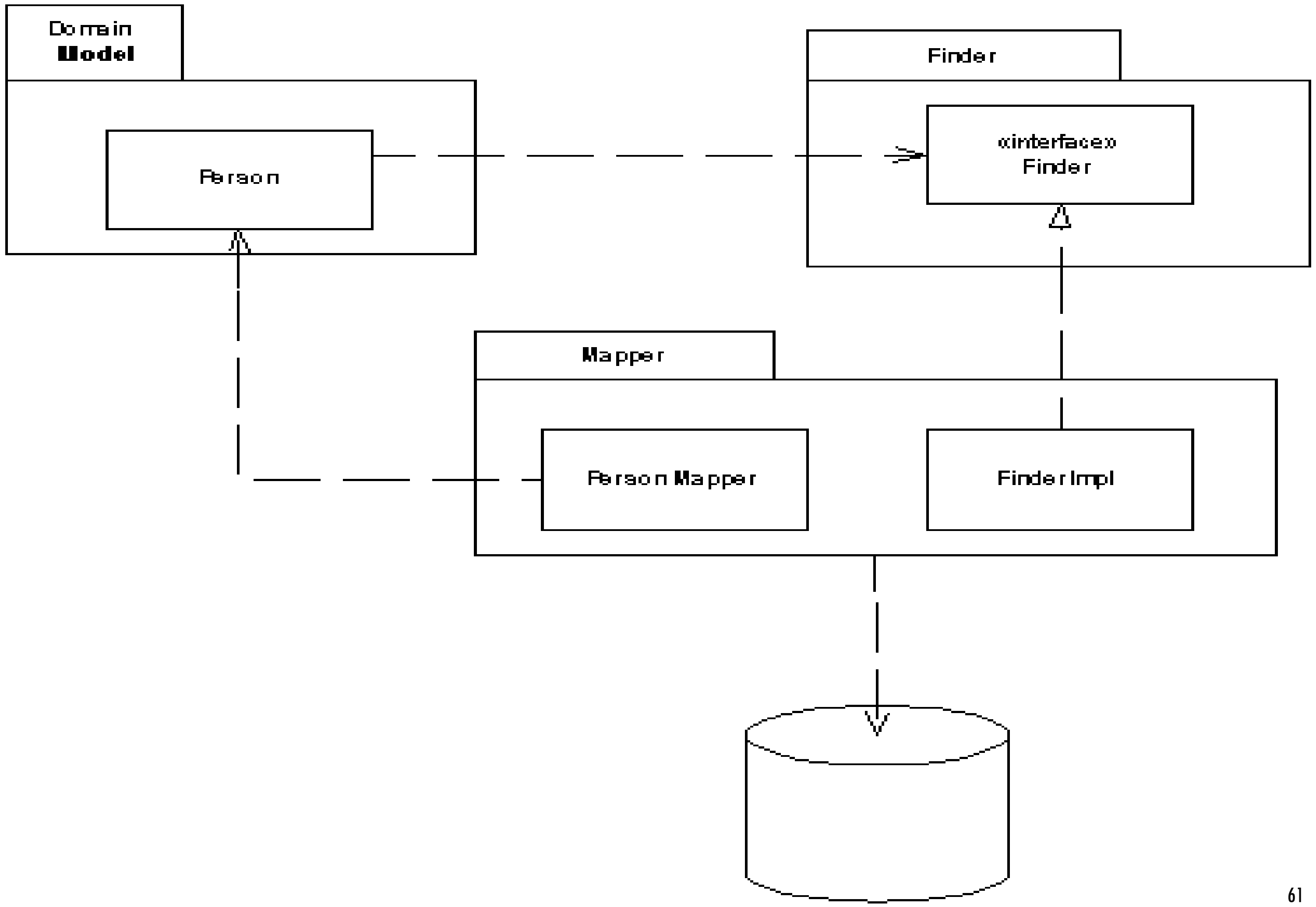
Does not contain Domain Logic.

When it uses a TDG/DAO, the Data Mapper can be placed in the (lower) Domain layer.

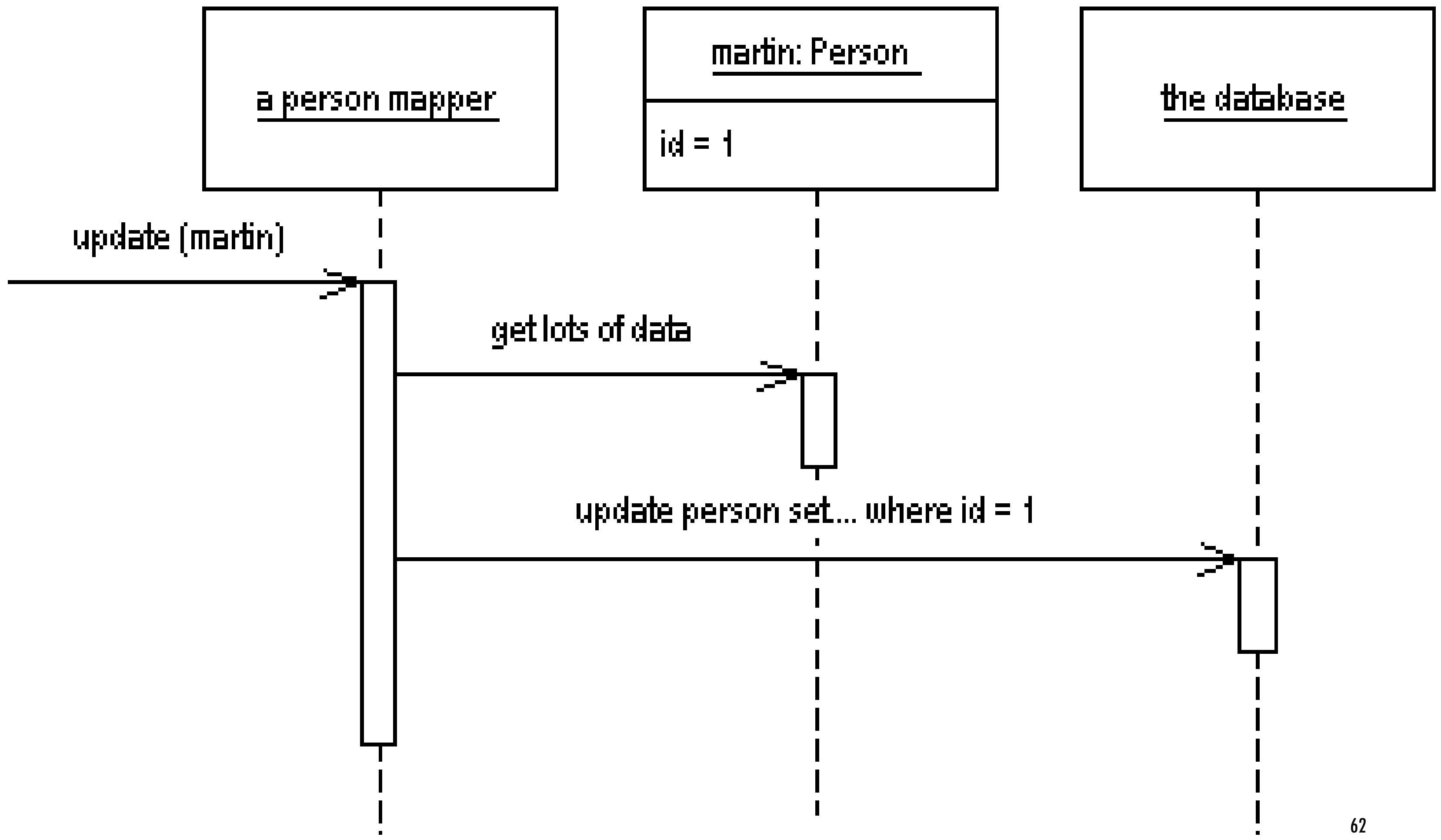
RETRIEVING DATA



FINDING OBJECTS



UPDATING DATA



FEATURES

- Independent database schema and object model
- Extra layer
- Makes sense with Domain Model

IMPLEMENTATION

```
class Person {
 private String name;
 private int numberOfDependents;
… }
```

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

ABSTRACTMAPPER

```
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 METHOD IN ABSTRACTMAPPER

```
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 SQLException;
```

MAPPER CLASS IMPLEMENTS FINDER

```
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);
 }
```

DOLOAD IN PERSONMAPPER

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

STRUCTURAL PROBLEMS

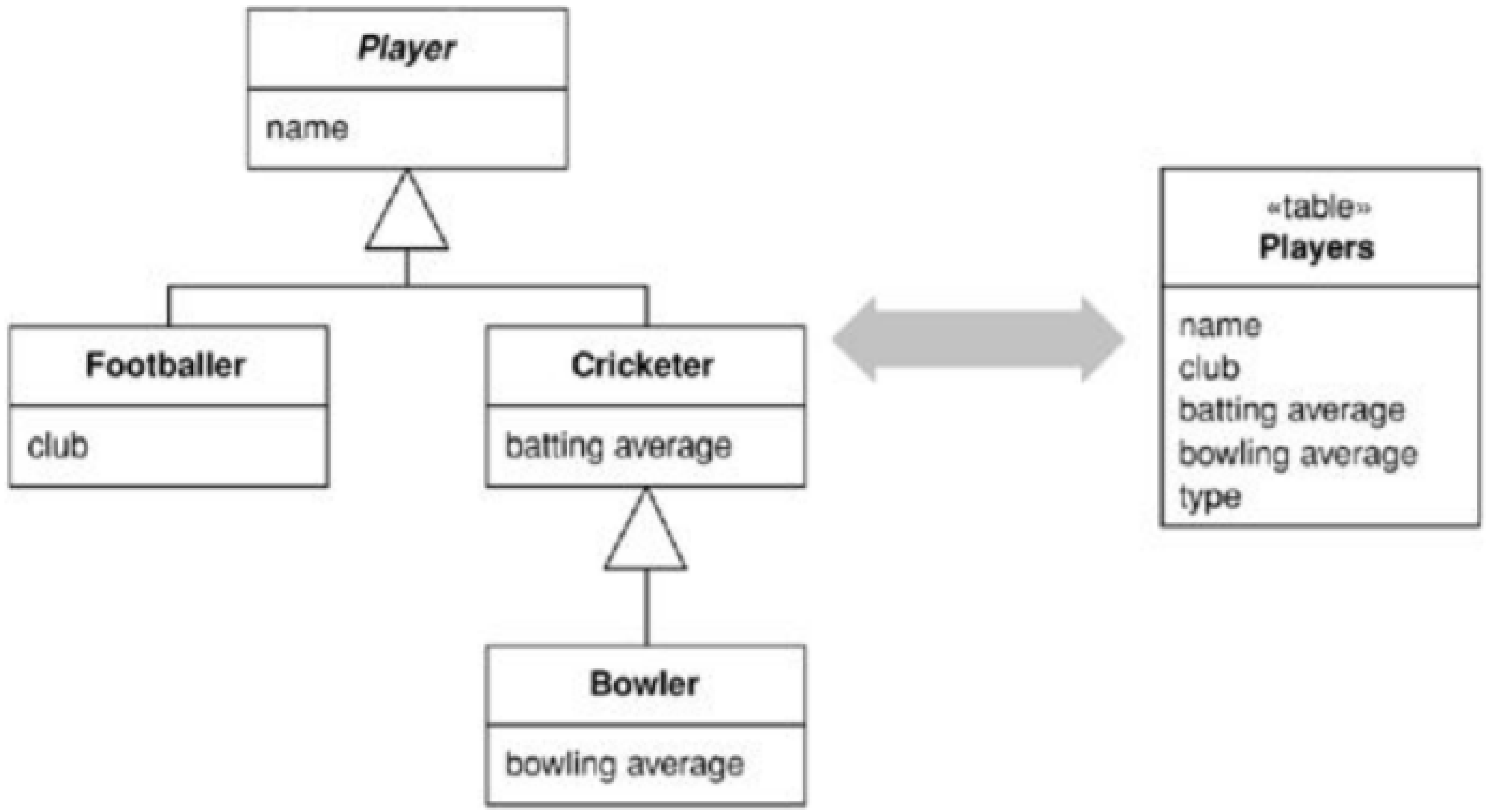
Object Relationships structural mapping problems

- Association/Composition
- **Inheritance**

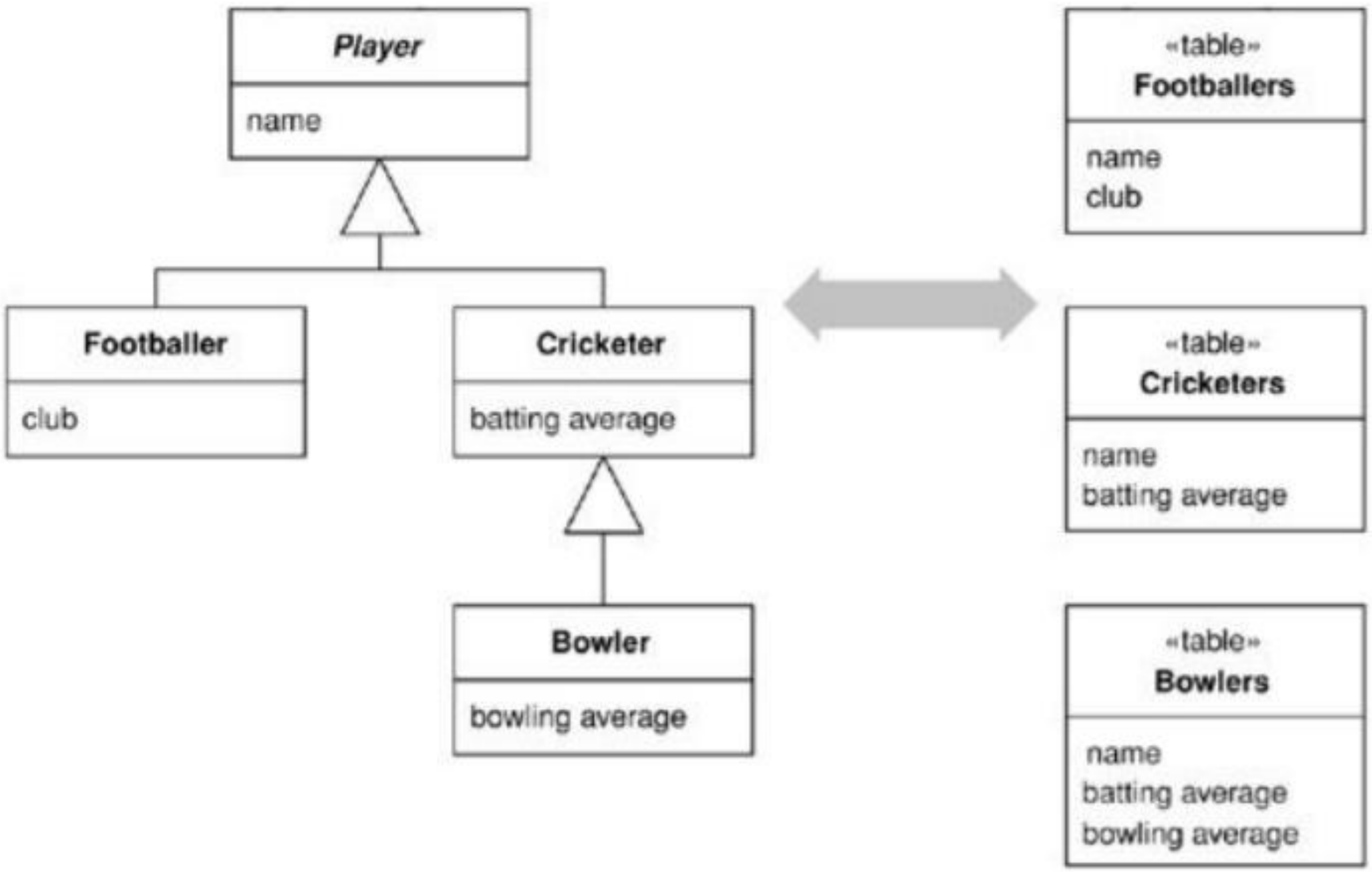
Object-Relational Structural Patterns

- **Single Table Inheritance**
- **Class Table Inheritance**
- **Concrete Table Inheritance**

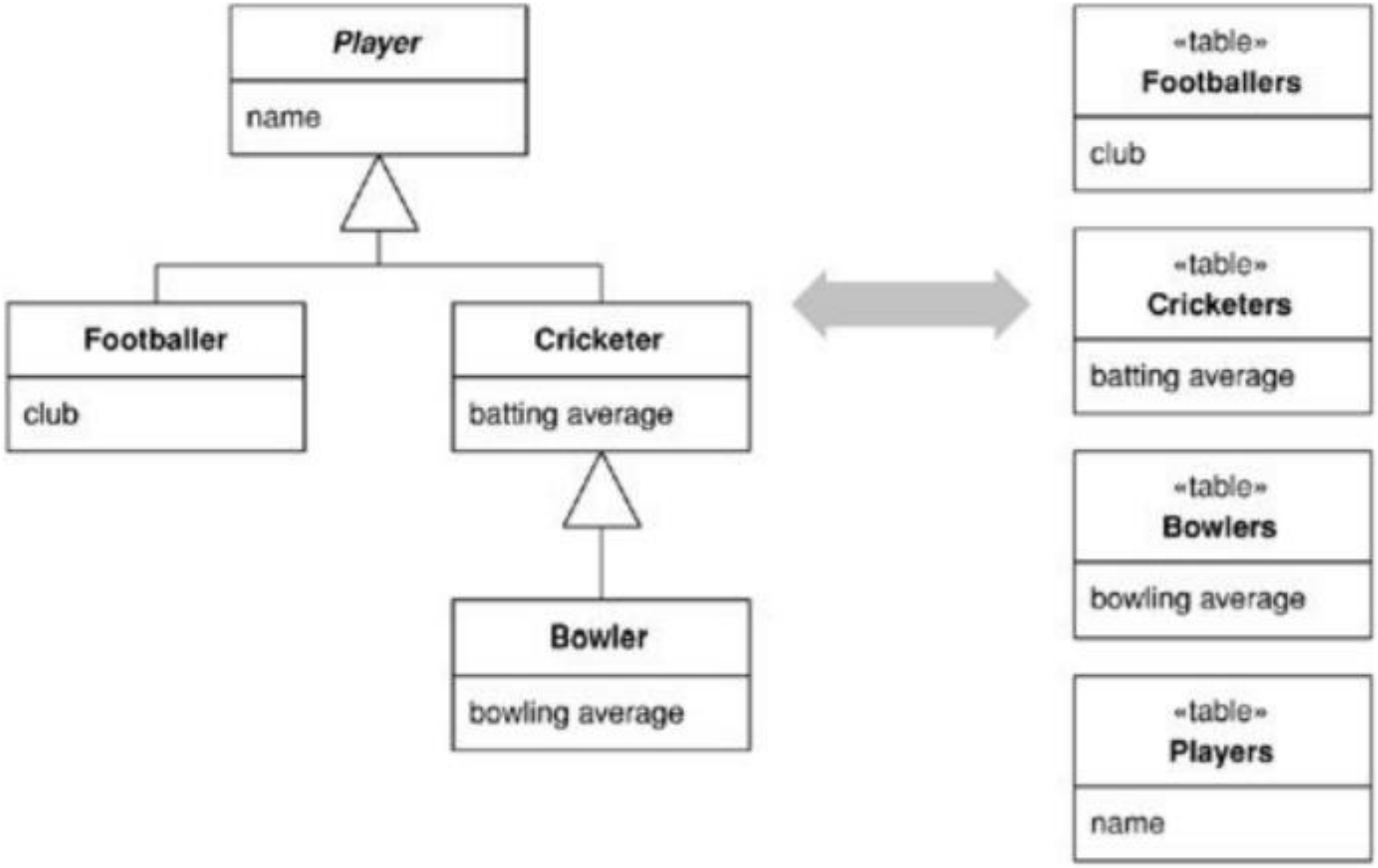
SINGLE TABLE INHERITANCE



CONCRETE TABLE INHERITANCE



CLASS TABLE INHERITANCE



BEHAVIORAL PROBLEMS

Networks of objects

- E.g. Invoice heading relates to invoice details
- Invoice details refers to Products
- Products refers to Suppliers
- …

What to do?

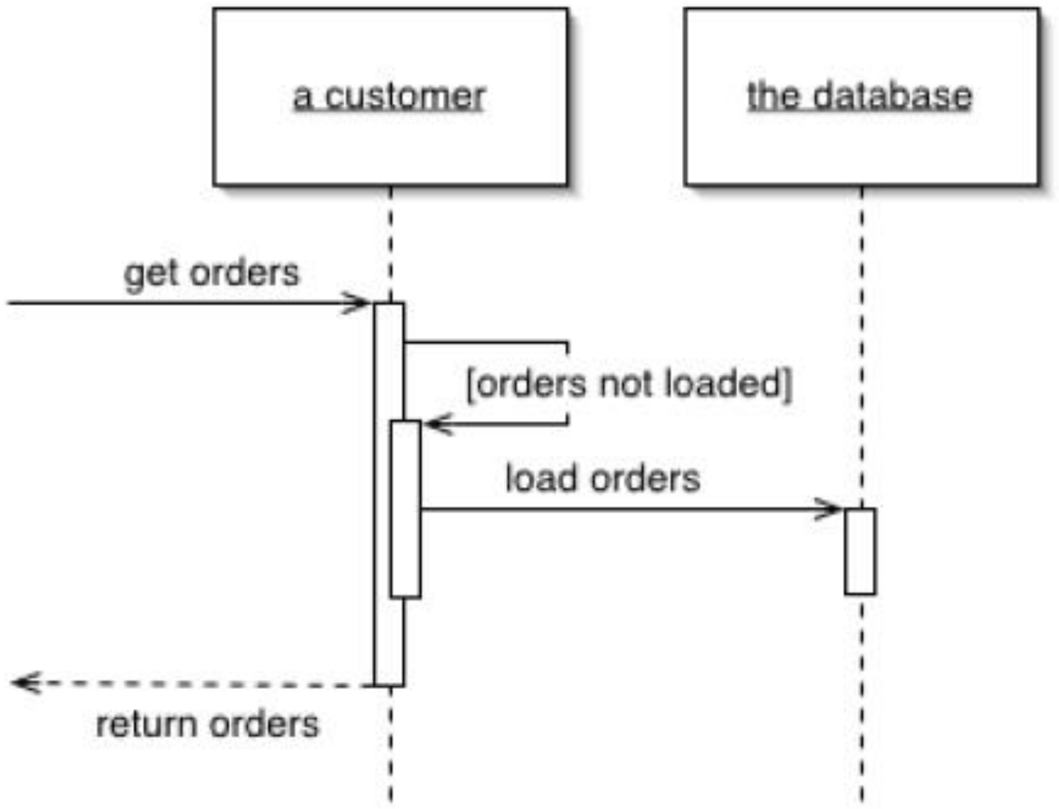
- Load them all into memory?
- How to disallow multiple in-memory copies

Object-Relational Behavioral Patterns

- **Lazy Load**
- **Identity Map**

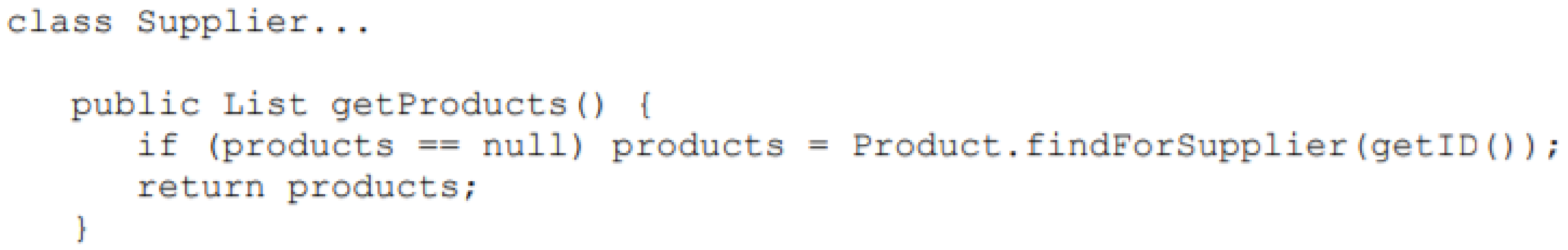
LAZY LOAD

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



IMPLEMENTATION OPTIONS

Lazy initialization: every access to the field checks first to see if it's null. If so, it calculates the value of the field before returning the field.



```
class Supplier...

   public List getProducts() {
      if (products == null) products = Product.findForSupplier(getID());
      return products;
   }
```

Virtual proxy: looks like the object but doesn’t contain anything. Only when one of its methods is called does it load the object from the database.

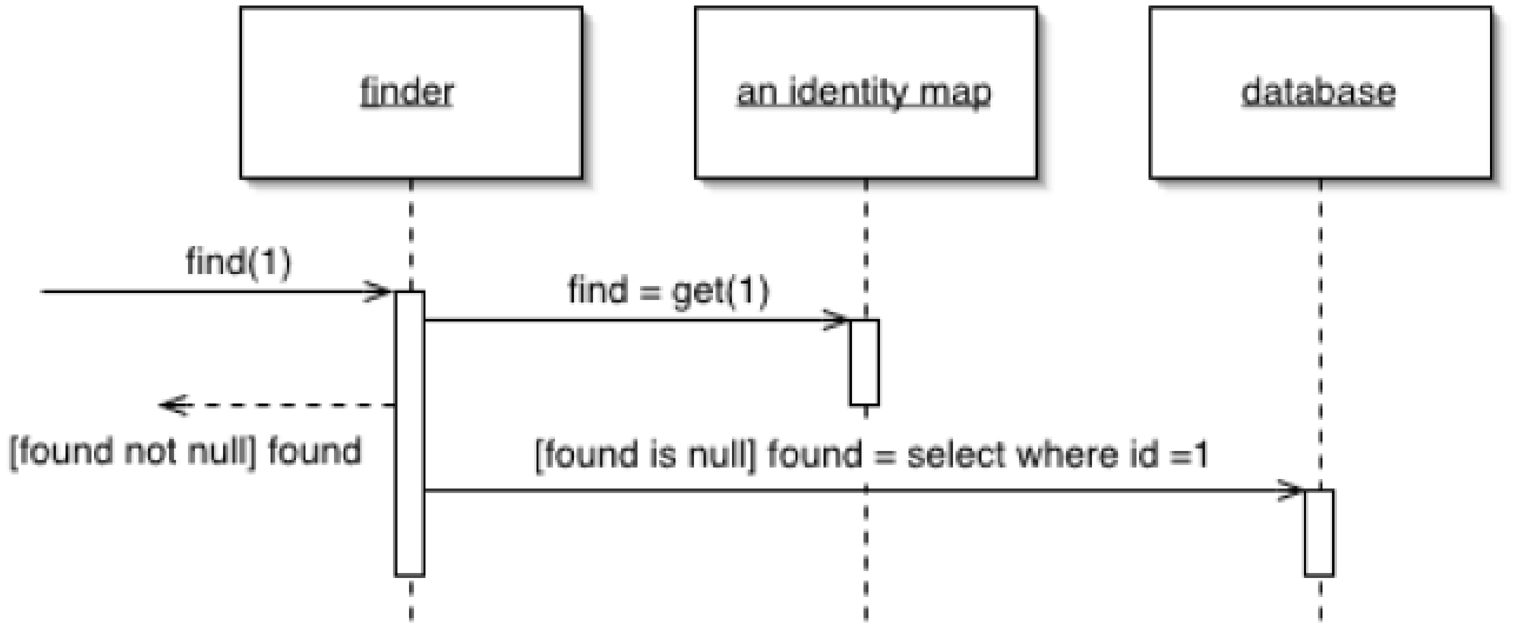
Ghost: the real object in a partial state. Ex. when you load the object from the database it contains just its ID. Whenever you try to access a field it loads its full state.

DISCUSSION

- Inheritance might be a problem with Lazy Load
- Can easily cause more database accesses than you need (ex. fill a collection with Lazy Loads)
- Deciding when to use Lazy Load is all about deciding how much you want to pull back from the database as you load an object, and how many database calls that will require.

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



HOW IT WORKS

Map key?

- Primary key in the table (if it is a single column and immutable)

Explicit vs. generic

- findPerson(1)
- find ("Person", 1)

How many?

- One map/session (if you have database-unique keys)
- One map/table
- One map/class
- One map/inheritance tree

WHEN TO USE IT

- Use an Identity Map to manage any Entity object brought from a database and modified.
- Acts as a cache for database reads
- May not need an Identity Map for immutable objects
- Helps avoid update conflicts within a single session, but it doesn't do anything to handle conflicts that cross sessions(see the Concurrency topic)

NEXT TIME

More patterns

- Concurrency
- Presentation