DEVELOPMENT OF INFORMATION SYSTEMS

Lecture 4

2021/2022

Martin Radvanský

Repetition of the last lecture

Design patterns: ready-made solutions

Design patterns are solutions to general problems that software developers are faced during software development.

General principles of software development

- 23 patterns GoF
- Creational patterns
- Structural patterns
- Behavioral patterns

Repetition of the last lecture

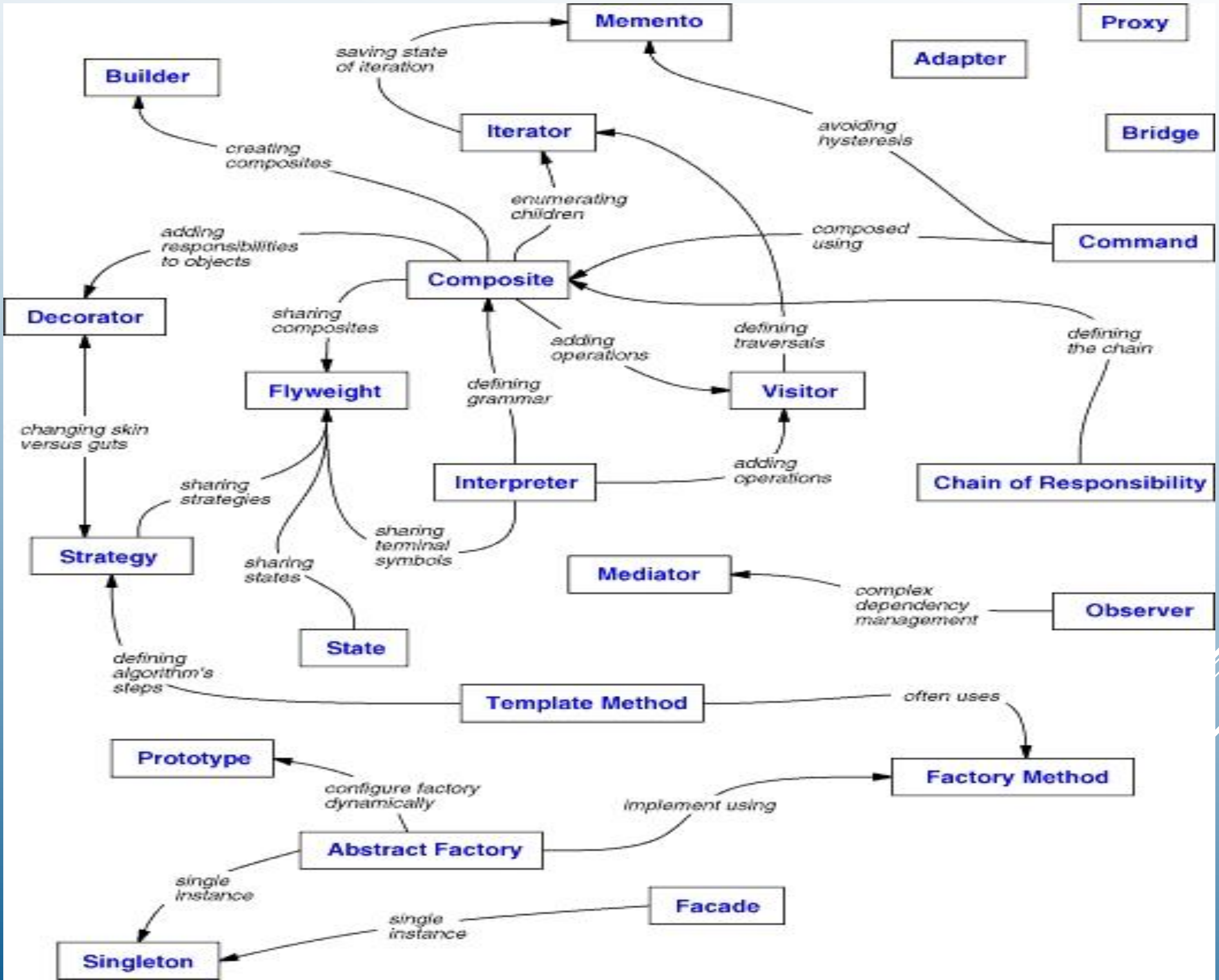
There are exist Anti-patterns too. It is not a working solution from a long term point of view.

Patterns are described in the form of a structured document containing:

- description of the problem
- context of the solution
- solution – mostly with UML diagram
- example of usage with source code

Repetition of the last lecture

GoF 23 design patterns



Repetition of the last lecture

Fowler Enterprise design patterns (2003)

Catalogue of more than 50 patters for ussage in develplopment not only Enterprise systems.

- Domain Logic Patterns
- Data Source Architectural Patterns
- Object-Relational Behavioral Patterns
- Object-Relational Structural Patterns
- Object-Relational Metadata Mapping Patterns
- Web Presentation Patterns
- Distribution Patterns
- Offline Concurrency Patterns
- Session State Patterns
- Base Patterns

Repetition of the last lecture

Layered architecture

Logical and physical separation of functionality.

- **Layer** refers to a functional division of the software.
- **Tier** refers to a functional division of the software that runs on infrastructure separate from the other divisions.

Advantages:

- Faster development
- Improved scalability
- Improved reliability
- Improved security

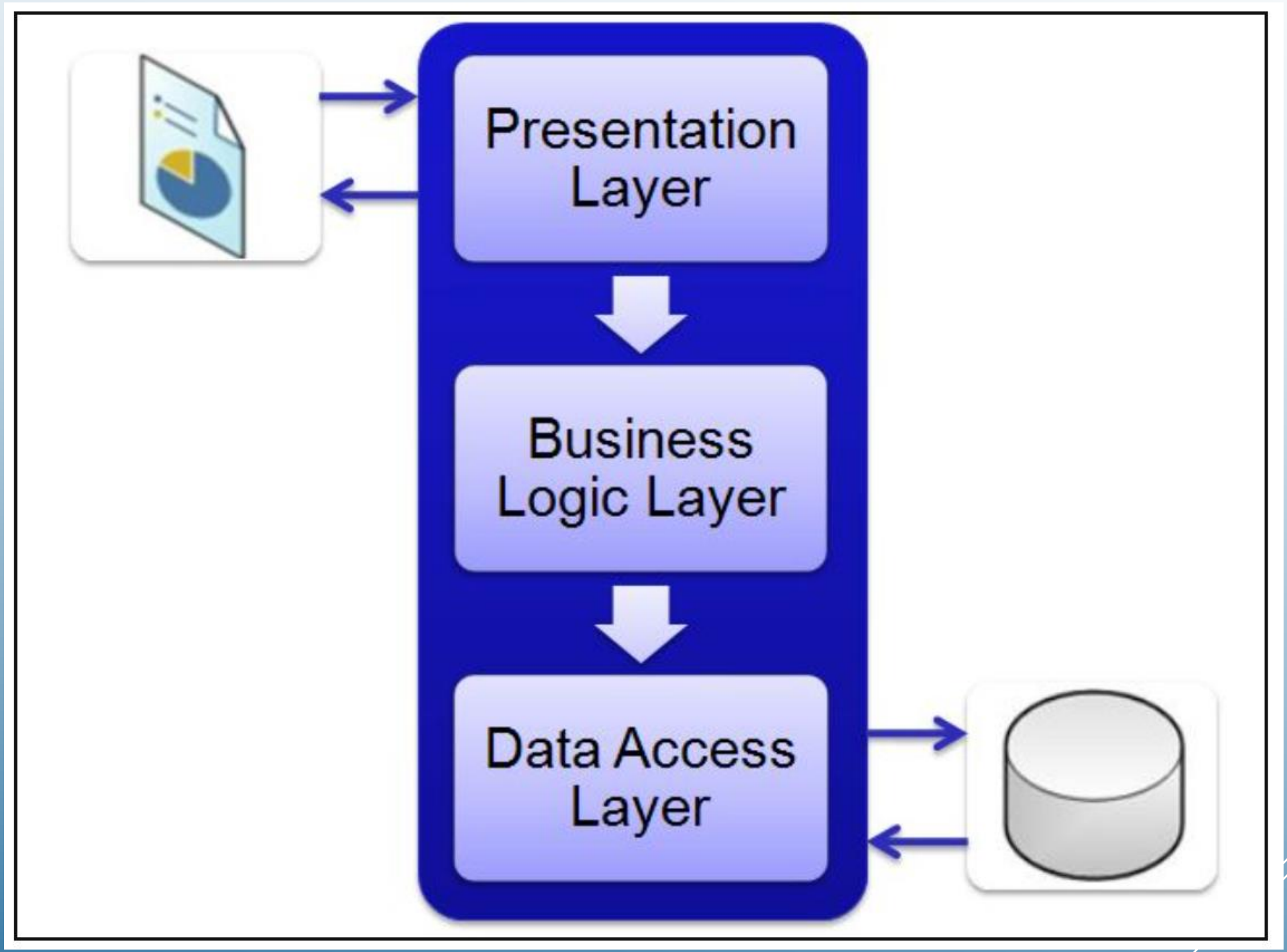
Repetition of the last lecture

Three layer/tier architecture

Application architecture that organizes applications into three logical and physical computing tiers.

- the presentation tier, or user interface
- the application tier, where data is processed
- the data tier, where the data associated with the application is stored and managed.

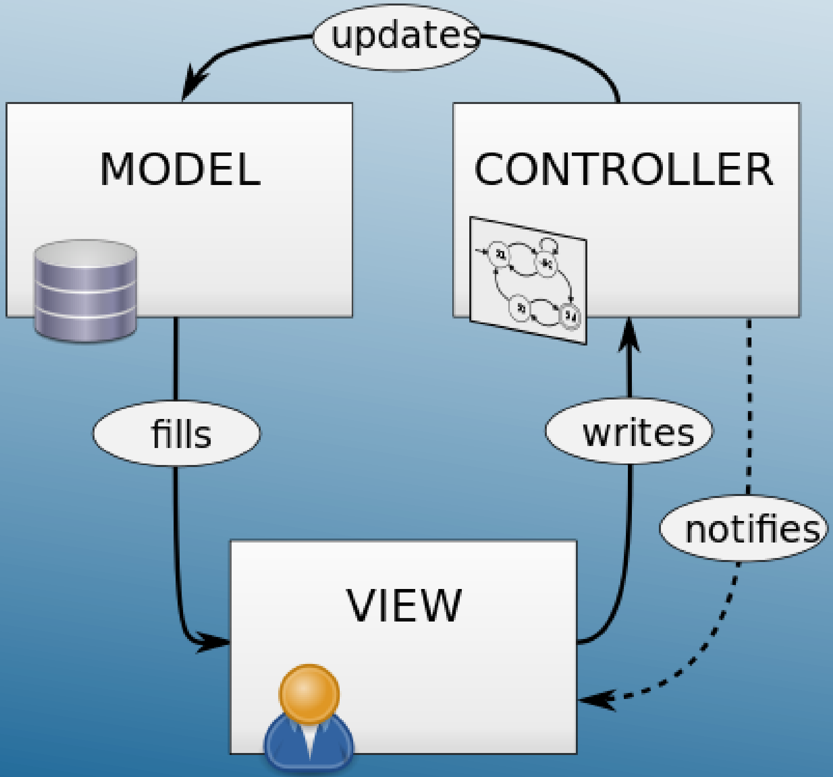
Repetition of the last lecture



Repetition of the last lecture

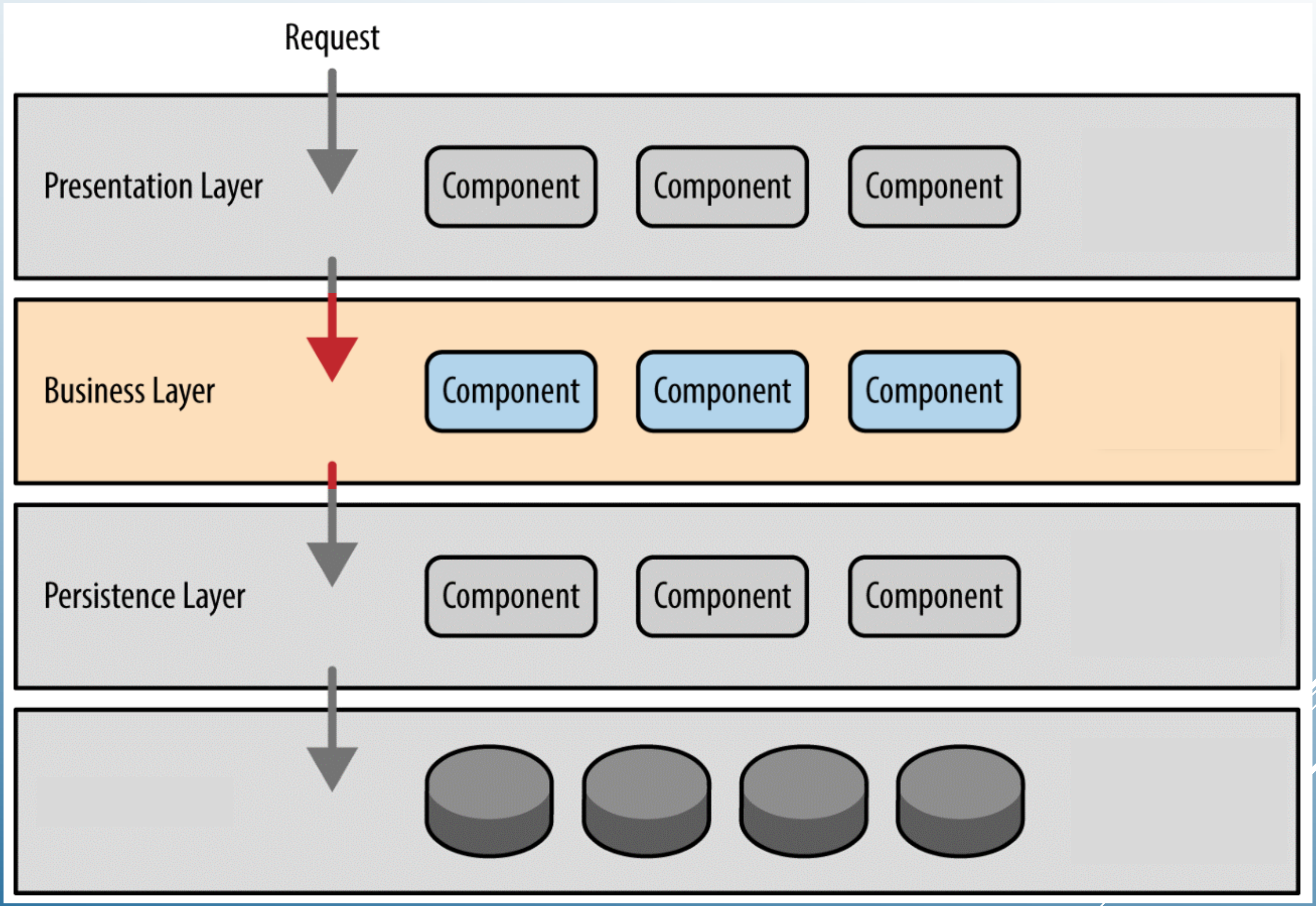
MVC Model-View-Controller

- **Model** - responsible for maintaining data
- **View** - data representation
- **Controller** - interconnection between the views and the model so it acts as an intermediary.



Repetition of the last lecture

Domain Logic Patterns



Repetition of the last lecture

Domain Logic Patterns

Domain logic or Business logic is the part of the program that encodes the real-world business rules that determine how data can be created, stored, and changed.

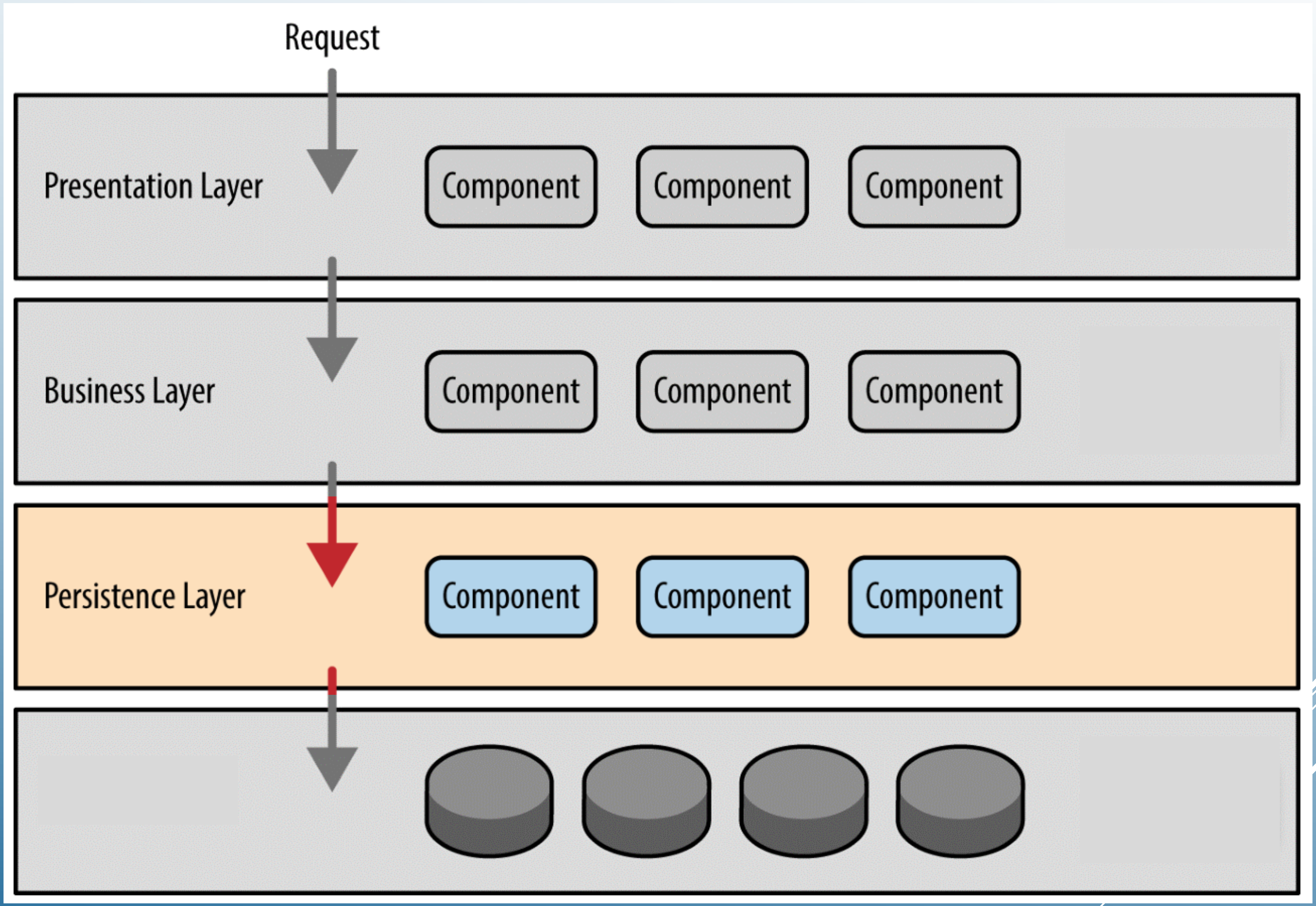
- **Transaction Script** - Organizes business logic by procedures where each procedure handles a single request from the presentation.
- **Domain Model** - An object model of the domain that incorporates both behavior and data.
- **Table Module** - A single instance that handles the business logic for all rows in a database table or view.
- **Service Layer** - application's boundary with a layer of services.

Repetition of the last lecture

Two different styles of the domain model

- **Simple** - works with simple logics and classes correspond to tables in the database
- **Full** - works with complex logics and the class model differs from database model

How to connect to relational data source?



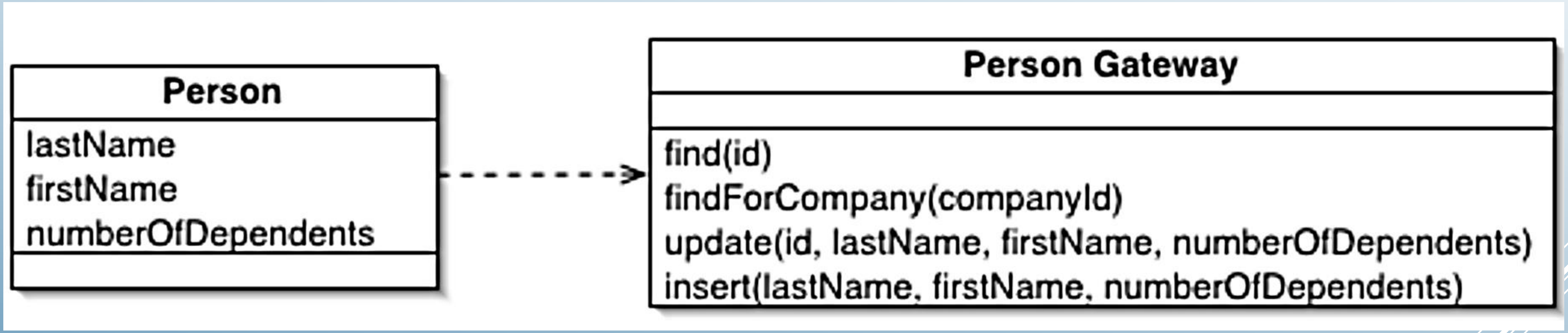
Data Source Architectural Patterns

The basic requirement is the independence of the domain logic from the logic of data access ...

- Table data gateway (TDG)
- Row data gateway (RDG)
- Active record (AR)
- Data mapper (DM)

Table data gateway

An object that acts as a Gateway (GoF) to a database table. One instance handles all the rows in the table.


Person
lastName
firstName
numberOfDependents
Person Gateway
find(id)
findForCompany(companyId)
update(id, lastName, firstName, numberOfDependents)
insert(lastName, firstName, numberOfDependents)

A TDG holds all the SQL for accessing a single table or view: selects, inserts, updates, and deletes. Other code calls its methods for all interaction with the database.

Table data gateway - How It Works

- A TDG has a simple interface, usually consisting of several find methods to get data from the database and update, insert, and delete methods.
- Each method maps the input parameters into a SQL call and executes the SQL against a database connection.
- The Table Data Gateway is usually **stateless**, as its role is to push data back and forth.

How it returns information from a query?

- Even a simple find-by-ID query will return multiple data items.
- In environments where you can return multiple items you can use that for a single row, but many languages give you only a single return value and many queries return multiple rows.
- One alternative is to return some simple data structure, such as a map. A map works, but it forces data to be copied out of the record set that comes from the database into the map.
- A better alternative is to use a **Data Transfer Object**. It's another object to create but one that may well be used elsewhere.
- You can return the **Record Set** that comes from the SQL query (.NET, JAVA)

Table data gateway and Domain patterns

Table Data Gateway is probably the simplest database interface pattern to use, as it maps so nicely onto a database table or record type. It makes a natural point to encapsulate the precise access logic of the data source.

- **Transaction script** is wery good choice for TDG. The result set representation is convenient for the Transaction Script to work with.
- **Table module** goes very well with TDG. TDG produces a record set data structure for the Table Module to work on.
- **Domain model** is not very suitable for work with TDG. Data Mapper gives a better isolation between the Domain Model and the database.

Table data gateway

- Returned data can be based on views rather than on the actual tables, which reduces the coupling between your code and the database.
- For very simple cases, however, you can have a single TDG that handles all methods for all tables.
- You can also have one for views or even for interesting queries that aren’t kept in the database as views but it won’t have update behavior. However, if you can make updates to the underlying tables, then encapsulating the updates behind update operations on the TDG is a supposed technique.

Table data gateway - implementation

Class diagram of data-set-oriented gateway and the supporting data holder

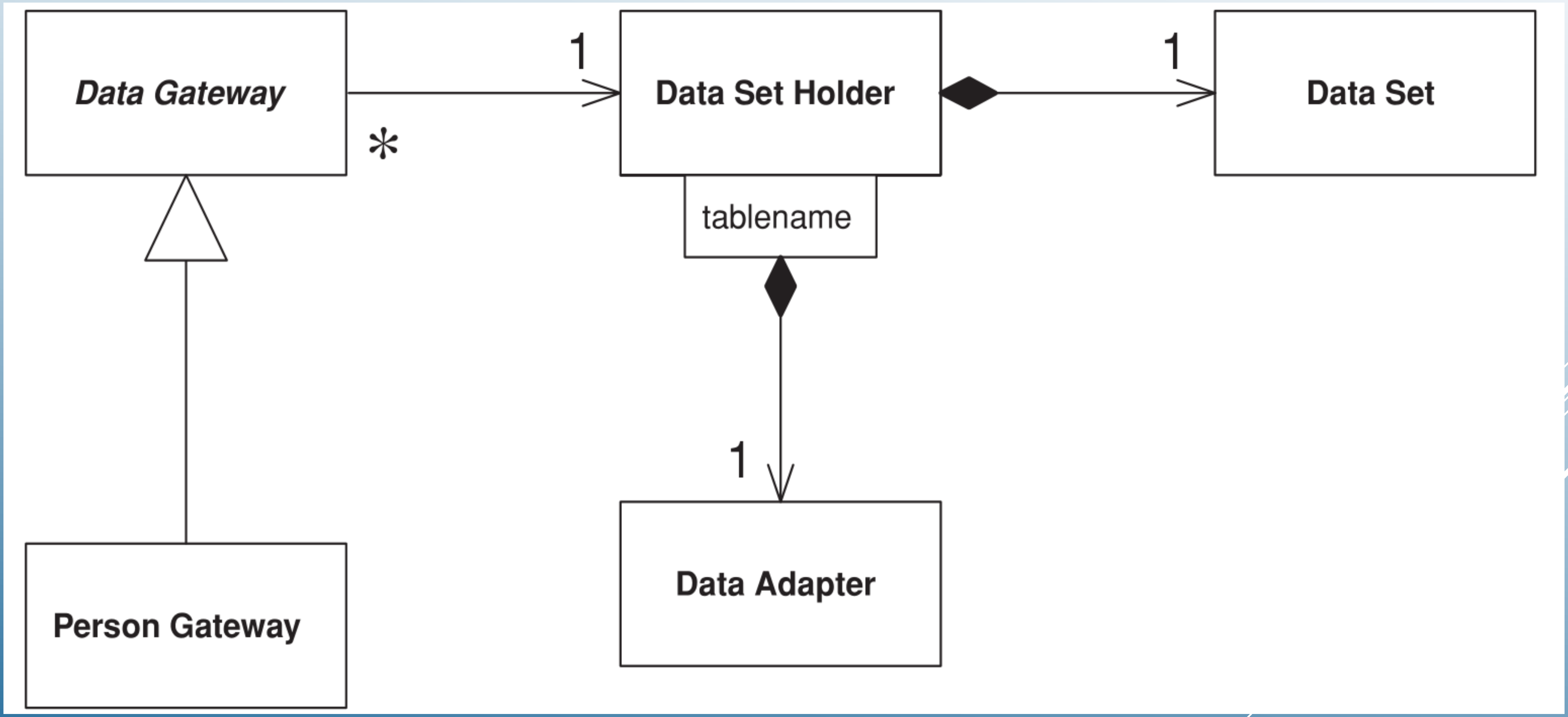


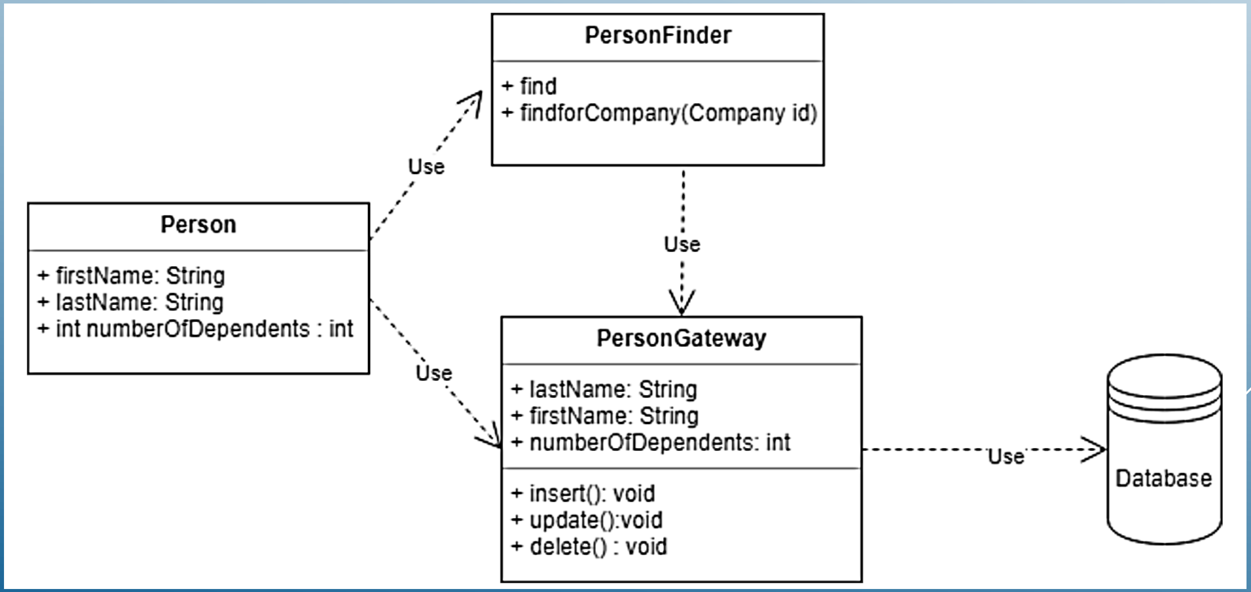
Table data gateway - implementation

- The gateway stores the holder and exposes the data set for its clients.
- The **find** behavior can work a bit differently here. A data set is a container for table-oriented data and can hold data from several tables. For that reason it’s better to load data into a data set.
- To **update** data you manipulate the data set directly in some client code and the update triggers update behavior on the holder.
- **Insertion**: Get a data set, insert a new row in the data table, and fill in each column. However, an update method can do the insertion in one call.

Row Data Gateway

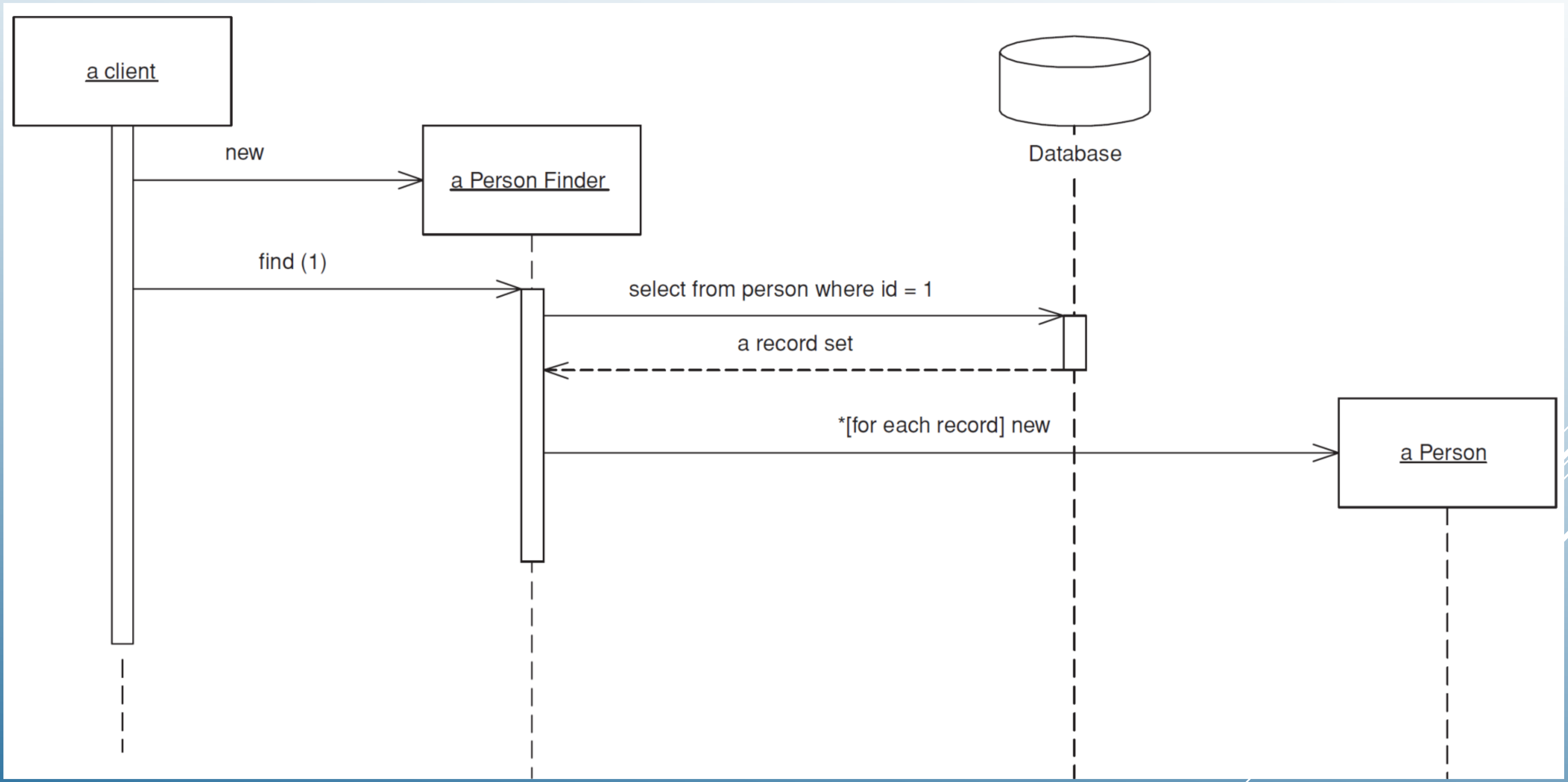
An object that acts as a Gateway to a single record in a data source. There is one instance per row.

Gateway means an object that encapsulates access to an external system or resource.
This object does not contain domain logic methods. If you introduce other methods (in particular domain logic) the object becomes an **Active Record Pattern.**



Row Data Gateway

Interactions for a find with a row-based Row Data Gateway.



Row Data Gateway – How it works?

- A Row Data Gateway gives you objects that look exactly like the record in your record structure but can be accessed with the regular mechanisms of your programming language. All details of data source access are hidden behind this interface.

- A Row Data Gateway acts as an object that exactly mimics a single record, such as one database row. In it each column in the database becomes one field. The Row Data Gateway will usually do any type conversion from the data source types to the in-memory types, but this conversion is pretty simple. This pattern holds the data about a row so that a client can then access the RowData Gateway directly. The gateway acts as a good interface for each row of data.

How it returns information from a query?

- Returns values similarly like Table Data Gateway
- Usually do any type of conversion from the data source types to the in-memory types by the simple conversions
- This pattern holds the data about a row

Row Data Gateway and Domain patterns

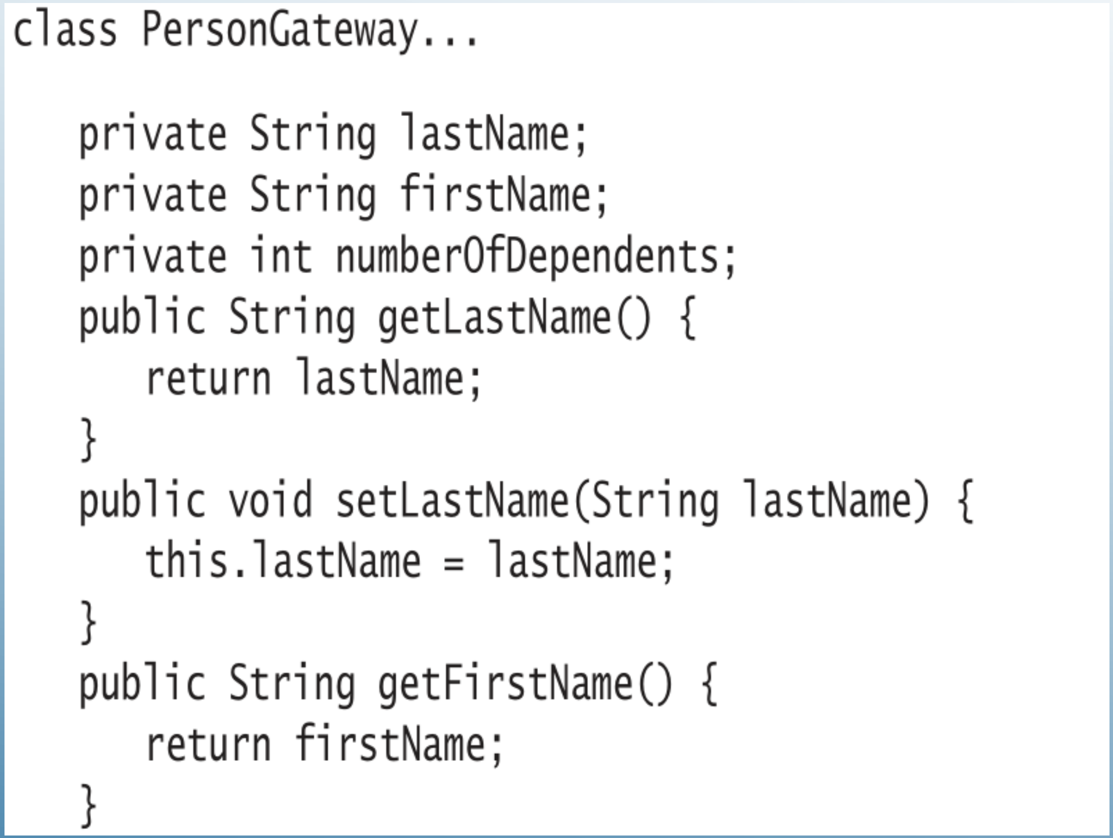
The choice of Row Data Gateway often takes two steps: first whether to use a gateway at all and second whether to use Row Data Gateway or Table Data Gateway.

- Using Row Data Gateway is most often together with a Transaction Script. In this case it nicely factors out the database access code and allows it to be reused easily by different Transaction Scripts.

- Row Data Gateway is not recommended together with a Domain Model. If the mapping is simple, Active Record does the same job without an additional layer of code. If the mapping is complex, Data Mapper works better, as it's better at decoupling the data structure from the domain objects.

Row data gateway - implementation

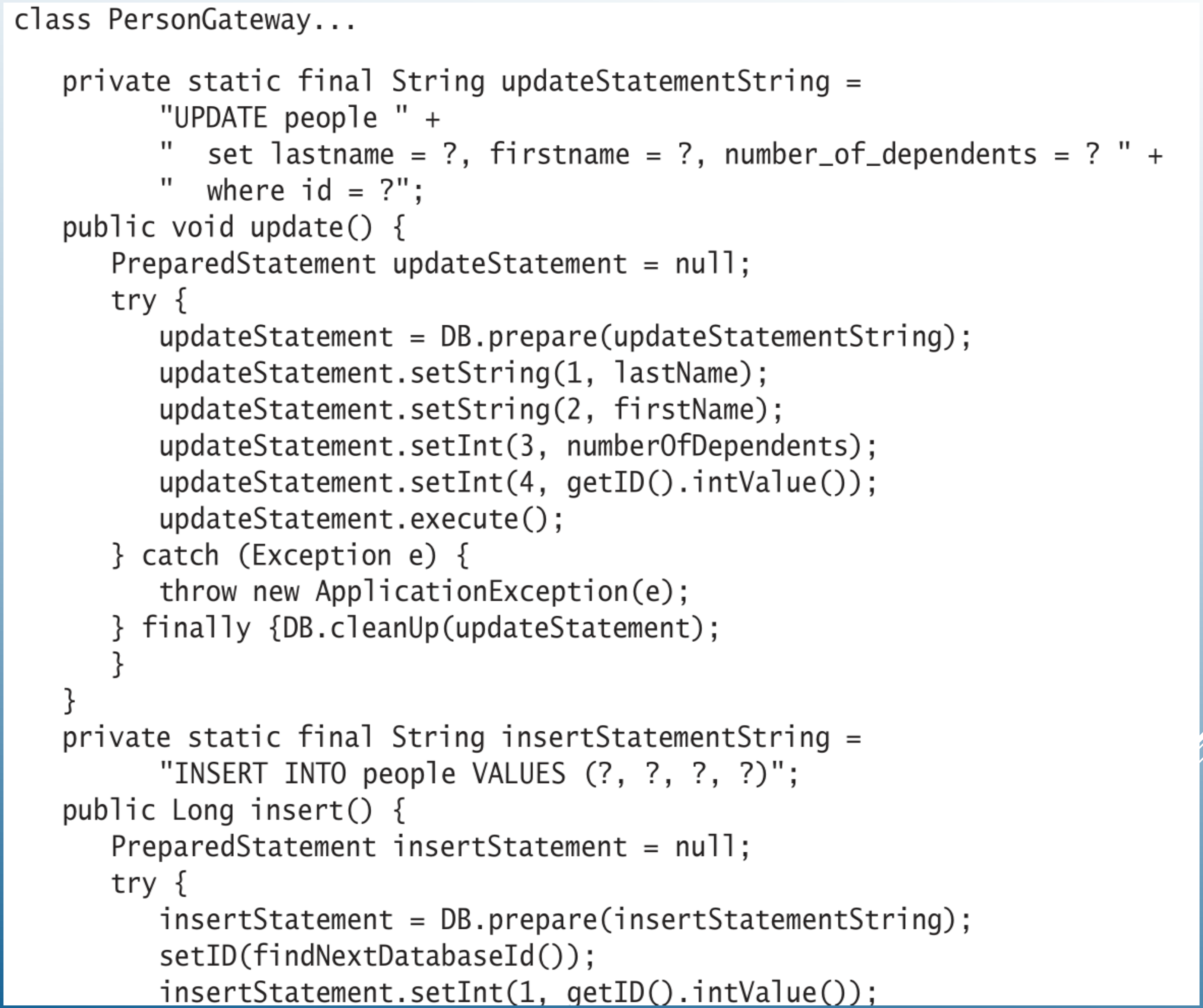
PersonGateway is a gateway for the table. It starts with data fields and accessors



```
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 String getFirstName() {
      return firstName;
   }
```

Row data gateway - implementation

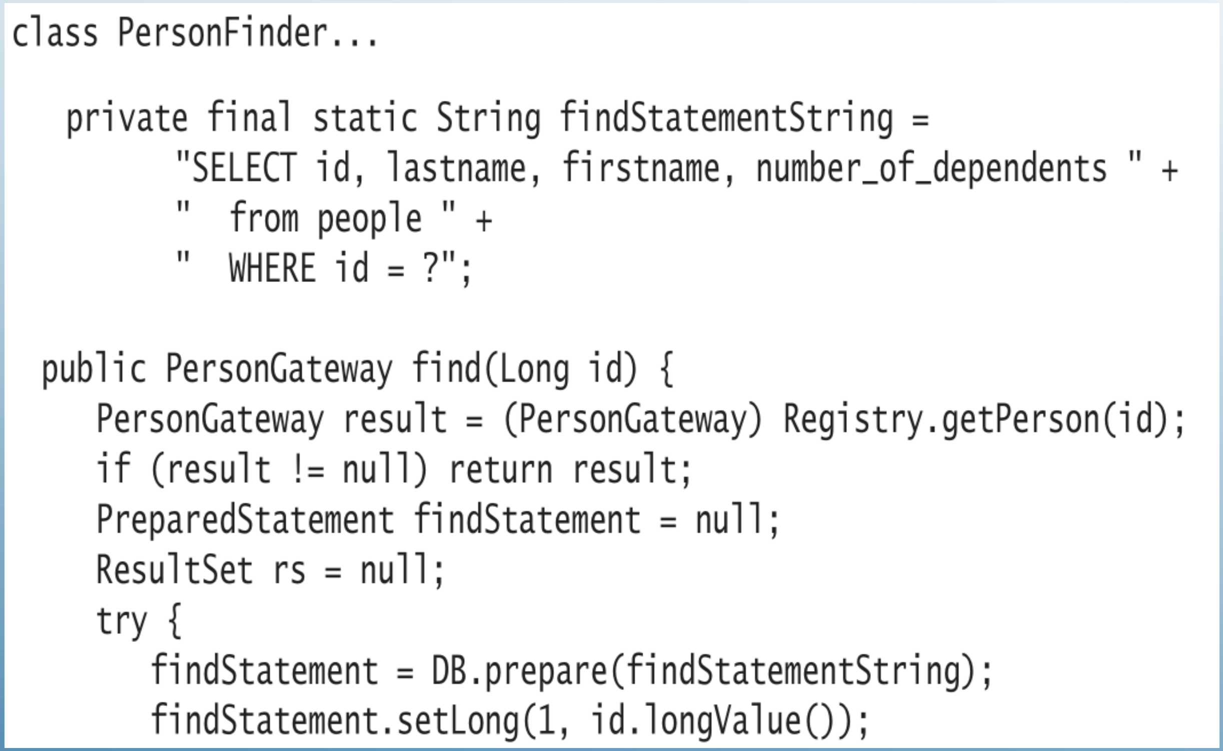
The gateway class itself can handle updates and inserts.



```
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);
      }
   }
   private static final String insertStatementString =
         "INSERT INTO people VALUES (?, ?, ?, ?)";
   public Long insert() {
      PreparedStatement insertStatement = null;
      try {
         insertStatement = DB.prepare(insertStatementString);
         setID(findNextDatabaseId());
         insertStatement.setInt(1, getID().intValue());
```

Row data gateway - implementation

To pull people out of the database, we have a separate PersonFinder. This works with the gateway to create new gateway objects.



```
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;
     PreparedStatement findStatement = null;
     ResultSet rs = null;
     try {
        findStatement = DB.prepare(findStatementString);
        findStatement.setLong(1, id.longValue());
```

Row data gateway - implementation

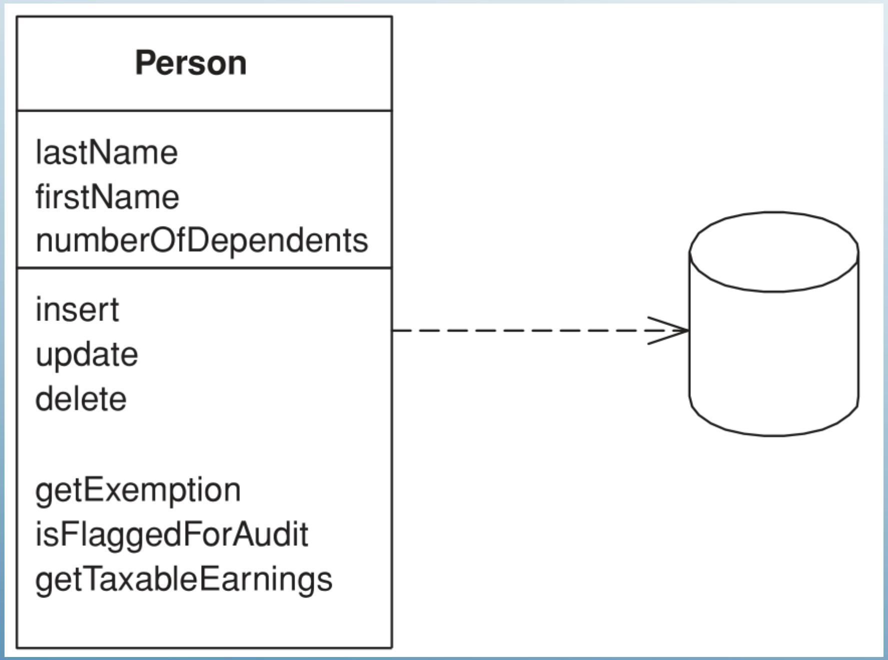
```
class PersonGateway...

   public static PersonGateway load(ResultSet rs) throws SQLException {
         Long id = new Long(rs.getLong(1));
         PersonGateway result = (PersonGateway) 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 PersonGateway(id, lastNameArg, firstNameArg, numDependentsArg);
         Registry.addPerson(result);
         return result;
   }
```

To find more than one person according to some criteria we can provide a suitable finder method.

```
public List findResponsibles() {
   List result = new ArrayList();
   PreparedStatement stmt = null;
   ResultSet rs = null;
   try {
      stmt = DB.prepare(findResponsibleStatement);
      rs = stmt.executeQuery();

      while (rs.next()) {
         result.add(PersonGateway.load(rs));
      }
      return result;
```

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.



An object carries both data and behavior. Much of this data is persistent and needs to be stored in a database.

Active record – How it works?

- Active Record uses the most obvious approach, putting data access logic in the domain object. This way all people know how to read and write their data to and from the database.
- Each Active Record is responsible for saving and loading to the database and also for any domain logic that acts on the data.
- The data structure of the Active Record should exactly match that of the database: one field in the class for each column in the table. Type the fields the way the SQL interface gives you the data - don’t do any conversion at this stage.

Active record – How it works?

The Active Record class typically has methods that do the following:

- 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
- Update the database and insert into it the data in the Active Record
- Get and set the fields
- Implement some pieces of business logic

How it returns information from a query?

- Active record add domain logic
- Returns values similarly like Row Data Gateway
- The getting and setting methods can do some other intelligent things, such as convert from SQL-oriented types to better in-memory types. Also, if you ask for a related table, the getting method can return the appropriate Active Record, even if you aren't using Identity Field on the data structure (by doing a lookup).
- In this pattern the classes are convenient, but they don't hide the fact that a relational database is present. As a result you usually see fewer of the other object-relational mapping patterns present when you're using Active Record.

Active record and Domain patterns

Active Record is a good choice for domain logic that isn't too complex, such as creates, reads, updates, and deletes. Derivations and validations based on a single record work well in this structure.

- **Domain model:** Not for complex business logic (problems with object's direct relationships, collections, inheritance).

- **Transaction script:** Good choice for it simplicity, use Active record in case you are beginning to feel the pain of code duplication and the difficulty in updating scripts and tables

Active record - Implemetation

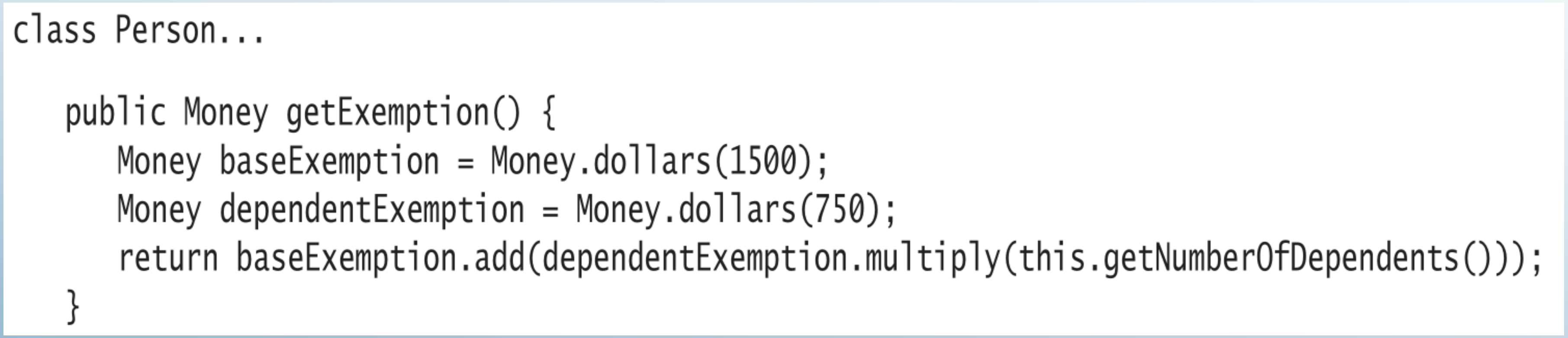
```
class Person...

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

To load an object, the person class acts as the finder and also performs the load. It uses static methods on the person class.

```
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;
}
```

Active record - Implemetation

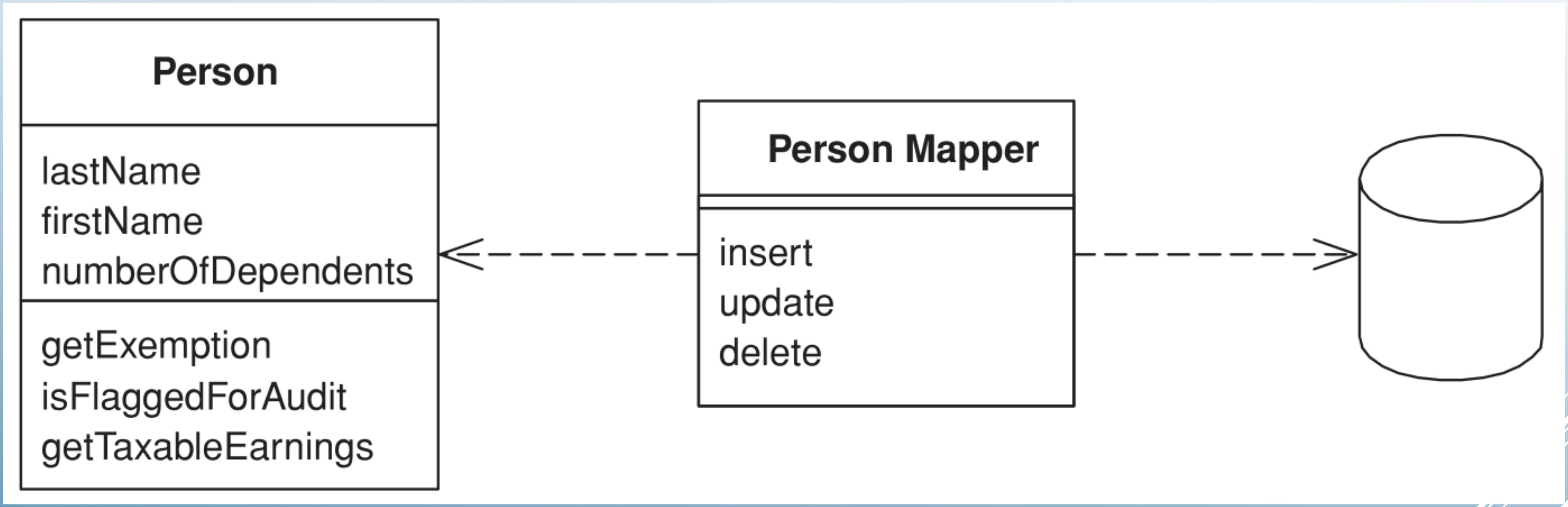
Any business logic, such as calculating the exemption, sits directly in the Person class.



```
class Person...

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

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.



The Data Mapper is a layer of software that separates the in-memory objects from the database. Its responsibility is to transfer data between the two and also to isolate them from each other.

Data Mapper – How it works?

- The separation between domain and data source is the main function of a Data Mapper, but there are plenty of details that have to be addressed to make this happen. There's also a lot of variety in how mapping layers are built.

- Objects and relational databases have different mechanisms for structuring data. Many parts of an object, such as collections and inheritance, aren't present in relational databases. When you build an object model with a lot of business logic it's valuable to use mechanisms to better organize the data and the behavior that goes with it.

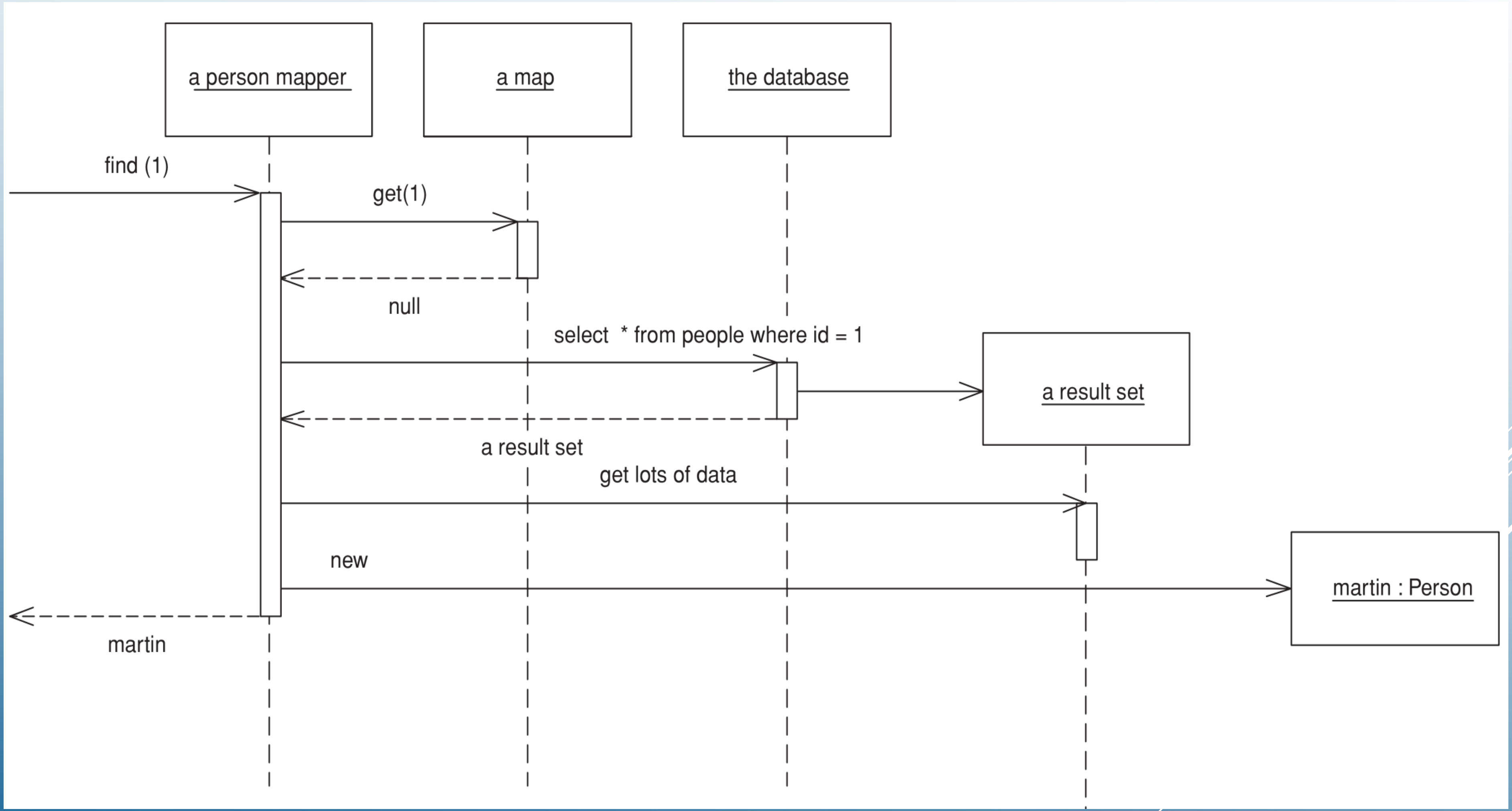
Data Mapper – How it works?

- The separation between domain and data source is the main function of a Data Mapper, but there are plenty of details that have to be addressed to make this happen. There's also a lot of variety in how mapping layers are built.

- Objects and relational databases have different mechanisms for structuring data. Many parts of an object, such as collections and inheritance, aren't present in relational databases. When you build an object model with a lot of business logic it's valuable to use mechanisms to better organize the data and the behavior that goes with it.

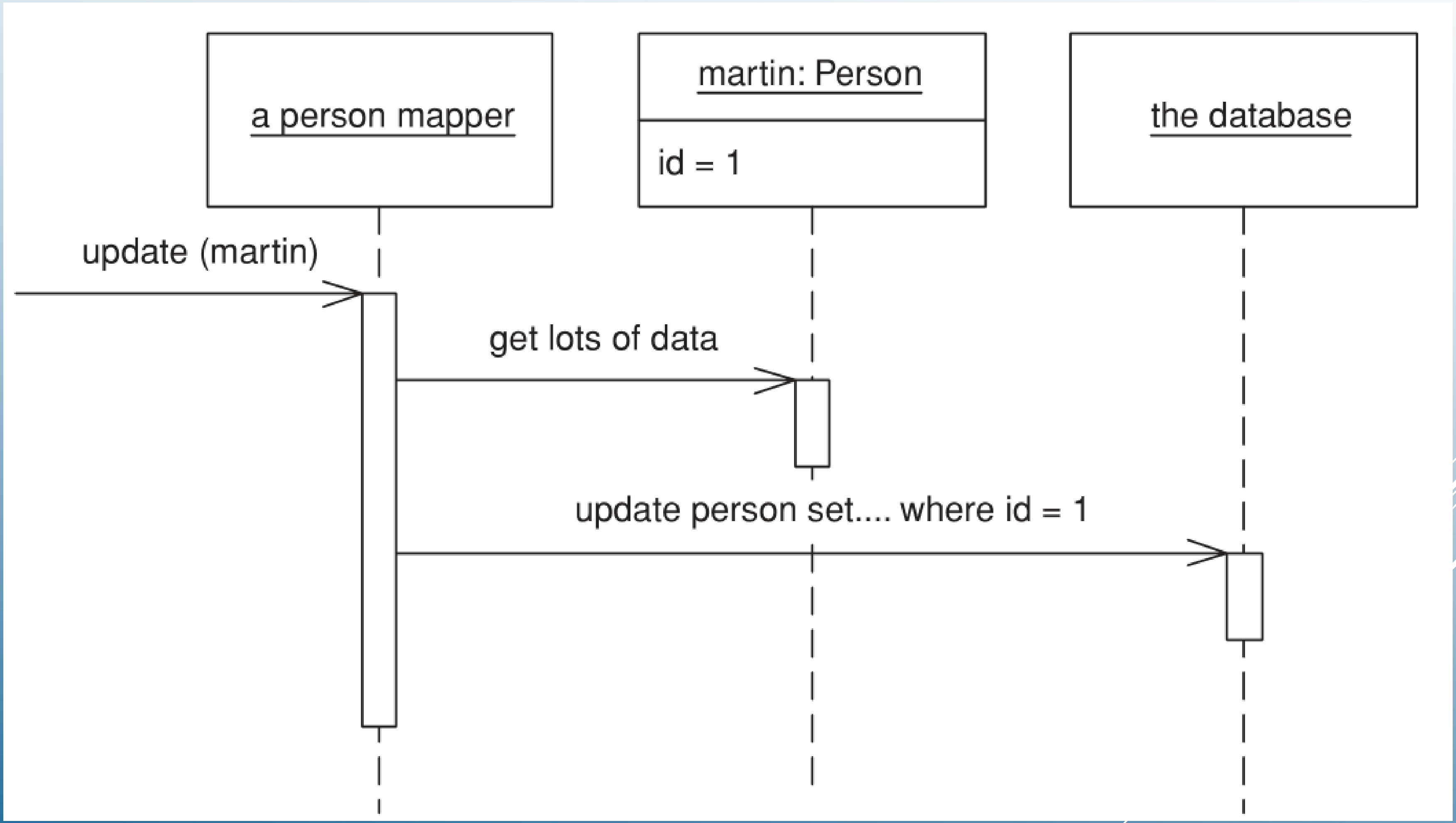
Data Mapper

Retrieving data from a database



Data Mapper

Updating data



Data Mapper – How it works?

- A simple case would have a Person and Person Mapper class. To load a person from the database, a client would call a find method on the mapper. The mapper uses an Identity Map to see if the person is already loaded; if not, it loads it.

- Updates - client asks the mapper to save a domain object. The mapper pulls the data out of the domain object and shuttles it to the database.

- A simple Data Mapper would just map a database table to an equivalent in-memory class on a field-to-field basis.

- An application can have one Data Mapper or several. If you’re hardcoding your mappers, it’s best to use one for each domain class or root of a domain hierarchy.

How it returns information from a query?

Mappers need access to the fields in the domain objects. Often this can be a problem because you need public methods to support the mappers you don't want for domain logic.

- You could use a lower level of visibility by packaging the mappers closer to the domain objects.
- You can use reflection, which can often bypass the visibility rules of the language.

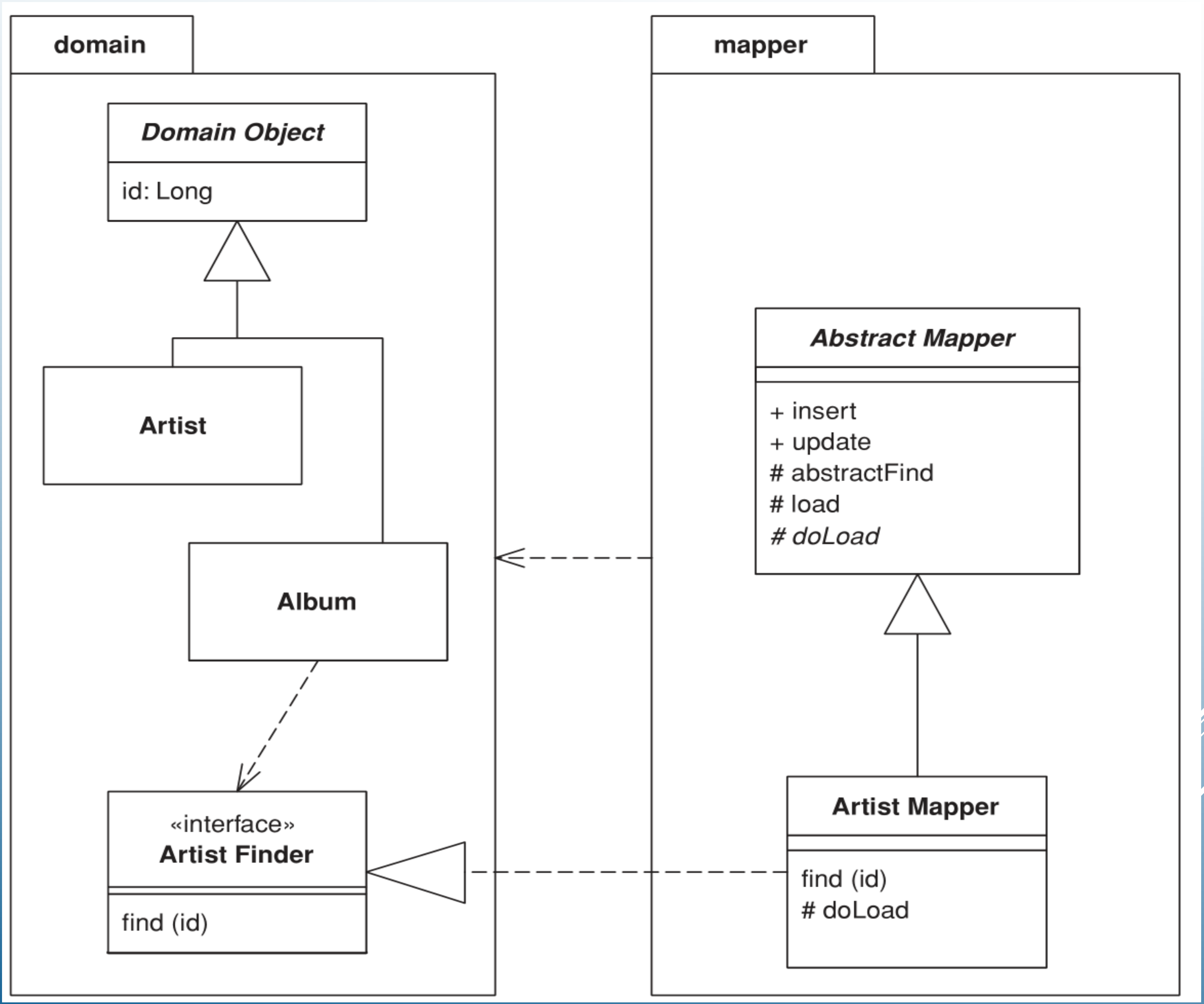
- Use public methods, but guard them with a status field so that they throw an exception if they're used outside the context of a database load.

Data Mapper and Domain patterns

Domain Model - most common caseis using Data Mapper. Main benefit is that when working on the domain model you can ignore the database.

Active Records, Table Module – not used with Data Mapper due its high complexity. (AR, TM is mostly used with simple business logic)

Data Mapper - Implementation



Data Mapper - Implementation

- We’ll use the simple case here, where the Person Mapper class also implements the finder and Identity Map. However, there is added an abstract mapper Layer Supertype to indicate where can be pull out some common behavior.

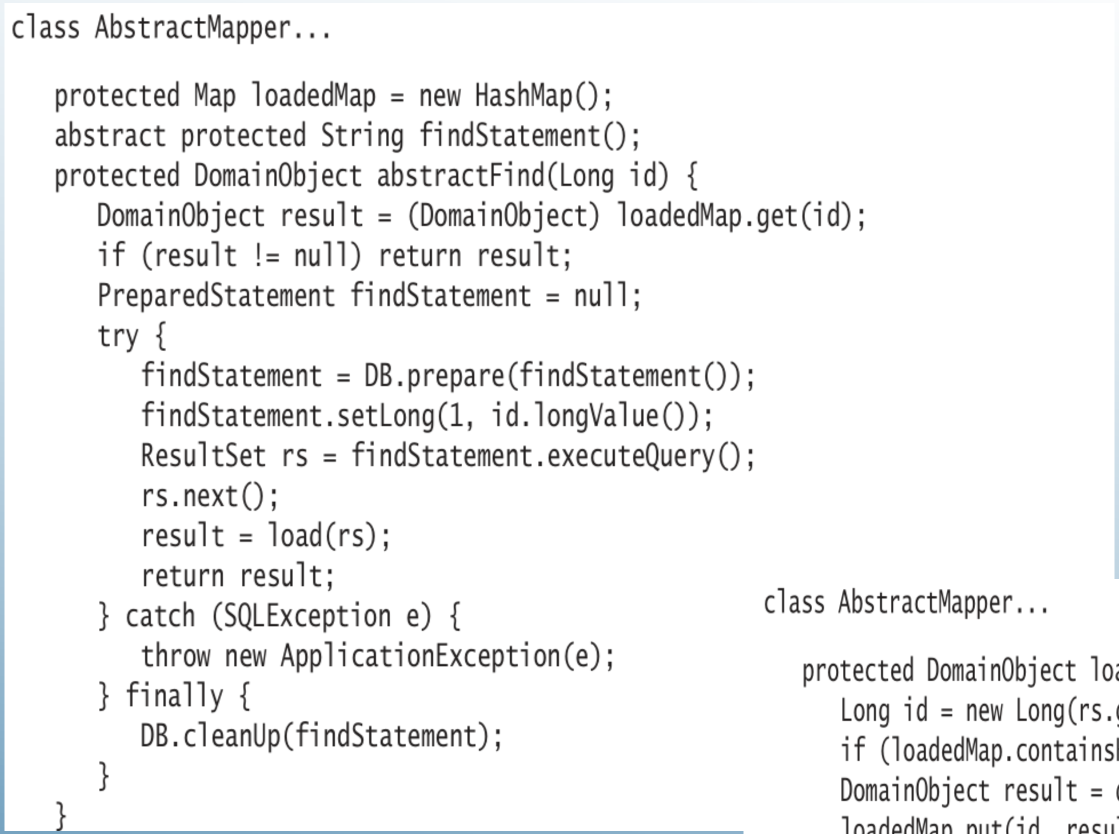
- Loading involves checking that the object isn’t already in the Identity Map and then pulling the data from the database.

- The find behavior starts in the Person Mapper, which wraps calls to an abstract find method to find by ID.

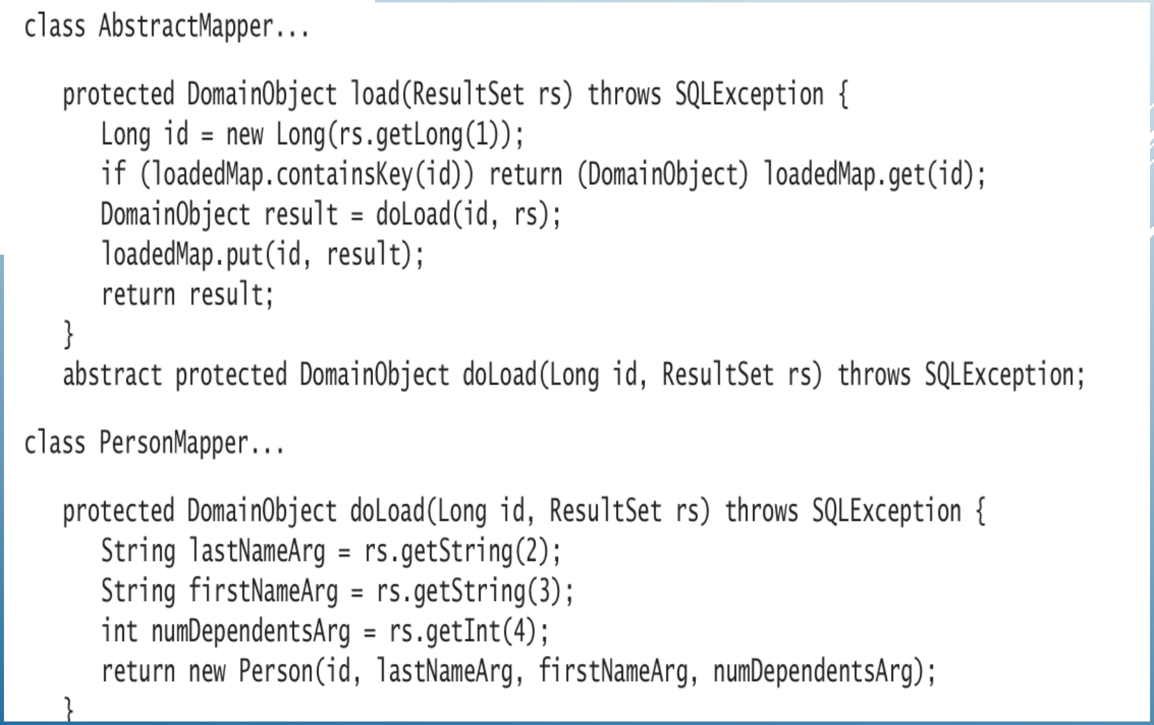
Data Mapper - Implementation

```
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));
   }
```

Data Mapper - Implementation

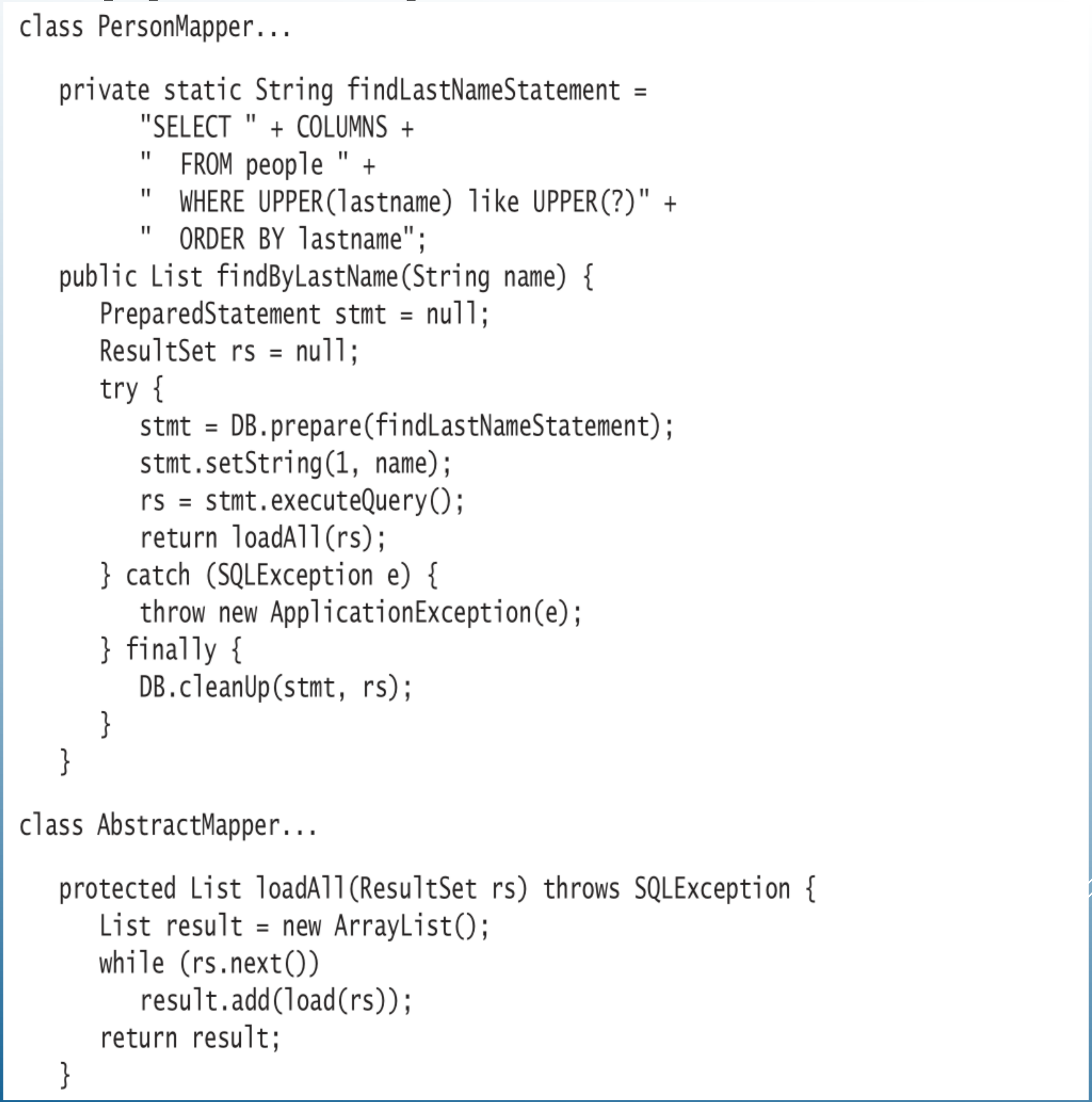


```
class AbstractMapper...

   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);
      }
   }
```



```
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;

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);
   }
```

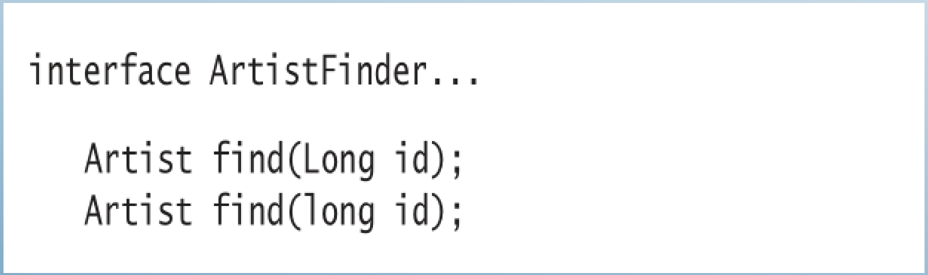
Data Mapper - Implementation



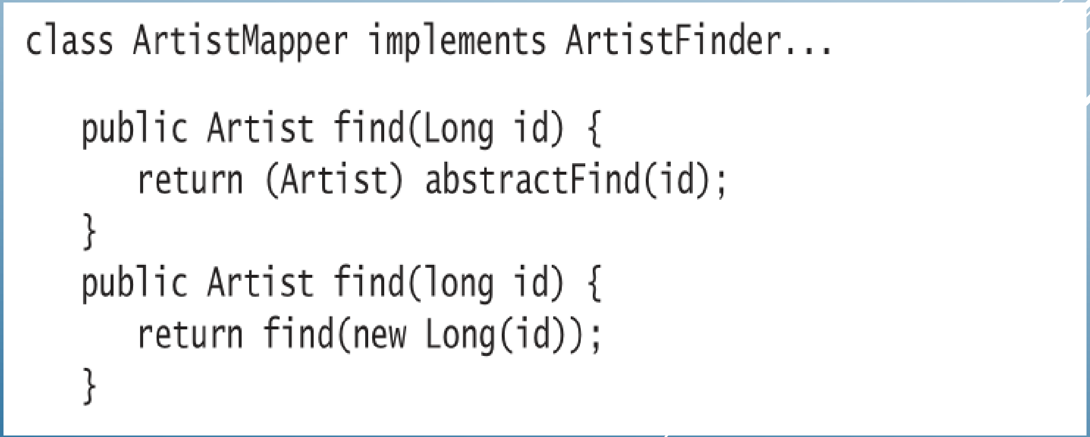
```
class PersonMapper...

   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);
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {
         DB.cleanUp(stmt, rs);
      }
   }

class AbstractMapper...

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

Data Mapper - Implementation

- To allow domain objects to invoke finder behavior. We can use Separated Interface to separate the finder interfaces from the mappers.
- We can put these finder interfaces in a separate package that's visible to the domain layer, or, as in this case, I can put them in the domain layer itself.



```
interface ArtistFinder...

   Artist find(Long id);
   Artist find(long id);
```



```
class ArtistMapper implements ArtistFinder...

   public Artist find(Long id) {
      return (Artist) abstractFind(id);
   }
   public Artist find(long id) {
      return find(new Long(id));
   }
```

Semestral project -Artifact 3

Date submission of Artifacts 2-4 is 10. 11. 2021

Analysis:
Non-functional (technical) requirements

- Conceptual domain model as input.
- Estimation of entity sizes and quantities.
- Estimation of the number of users simultaneously working with the system.
- Types of user interactions with the system and estimation of their complexity.
- First idea of the system layout and choice of platforms.

Exercise tasks

Continuing of the implementation task from the previous exercise using the patterns from the lecture

- Transaction script
- Domain model
- Module for table

Presentation of prepared use-case models

- Discussion of technical requirements
- Implementation of all patterns for accessing data sources for selected class from the chosen assignment, separation of domain class and data class into separate components.

Lecture checking questions

- What does the group of data patterns address?
- Describe the nature of each pattern for working with data sources. (table data gateway, row data gateway, active record, data mapper).
- What are the differences between the different data source patterns? When, where and why to use/not to use them?
- What patterns for domain logic can be worked with some of the patterns for working with data sources and why? Please provide a list of examples.