


# DEVELOPMENT OF INFORMATION SYSTEMS

## Lecture 5

# Repetition of the last lecture

## 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)
- 

# Repetition of the last lecture

- **Table data gateway** - an object that acts as a gateway to a database table. One instance handles all the rows in the table.
- **Row data gateway** - an object that acts as a gateway to a single record in a data source. There is one instance per row.
- **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.
- **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.

# Repetition of the last lecture

## Table data gateway (when to use)

### Pros

- Simple domain logic
- Transaction Script
- Table module
- Easy change of SQL logic

### Cons

- Not suitable for Domain Model

```
class PersonGateway...
```

```
public IDataReader FindAll() {  
    String sql = "select * from person";  
    return new OleDbCommand(sql, DB.Connection).ExecuteReader();  
}  
public IDataReader FindWithLastName(String lastName) {  
    String sql = "SELECT * FROM person WHERE lastname = ?";  
    IDbCommand comm = new OleDbCommand(sql, DB.Connection);  
    comm.Parameters.Add(new OleDbParameter("lastname", lastName));  
    return comm.ExecuteReader();  
}  
public IDataReader FindWhere(String whereClause) {  
    String sql = String.Format("select * from person where {0}", whereClause);  
    return new OleDbCommand(sql, DB.Connection).ExecuteReader();  
}
```

```
class PersonGateway...
```

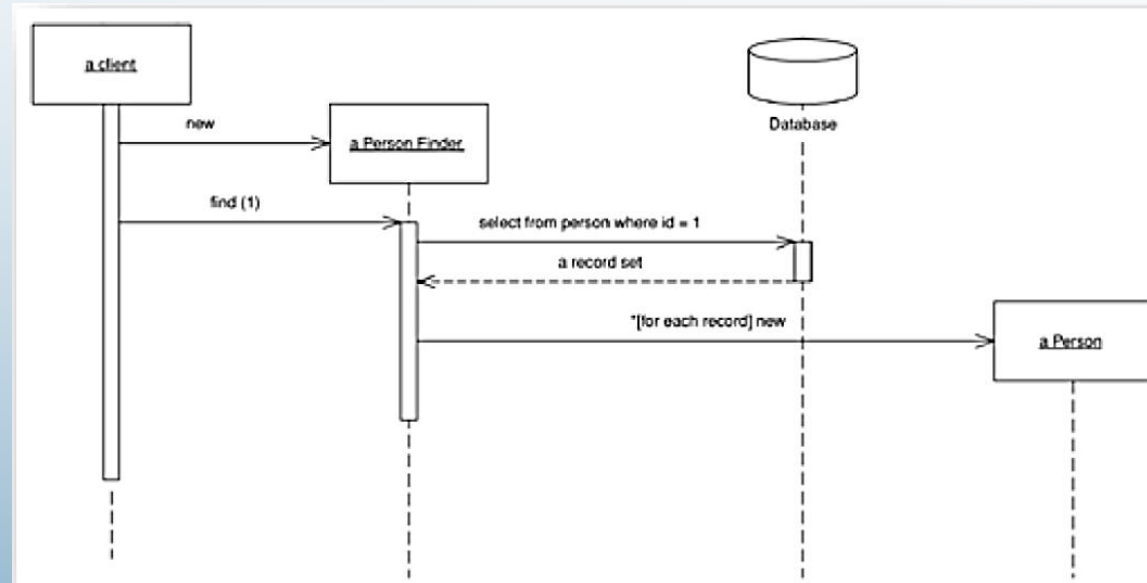
```
public Object[] FindRow (long key) {  
    String sql = "SELECT * FROM person WHERE id = ?";  
    IDbCommand comm = new OleDbCommand(sql, DB.Connection);  
    comm.Parameters.Add(new OleDbParameter("key", key));  
    IDataReader reader = comm.ExecuteReader();  
    reader.Read();  
    Object [] result = new Object[reader.FieldCount];  
    reader.GetValues(result);  
    reader.Close();  
    return result;  
}
```

# Repetition of the last lecture

## Row data gateway (when to use)

### Pros

- Simple domain logic
- Transaction Script



### Cons

- Not suitable for Domain Model

```
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);
```

# Repetition of the last lecture

## Active records (when to use) Pros

- A more complex domain, but with simple operations directly mapped to tables
- Table data gateway
- All domain patterns

## Cons

- Complicated mapping to DB

```
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);
    }
}

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;
}
```

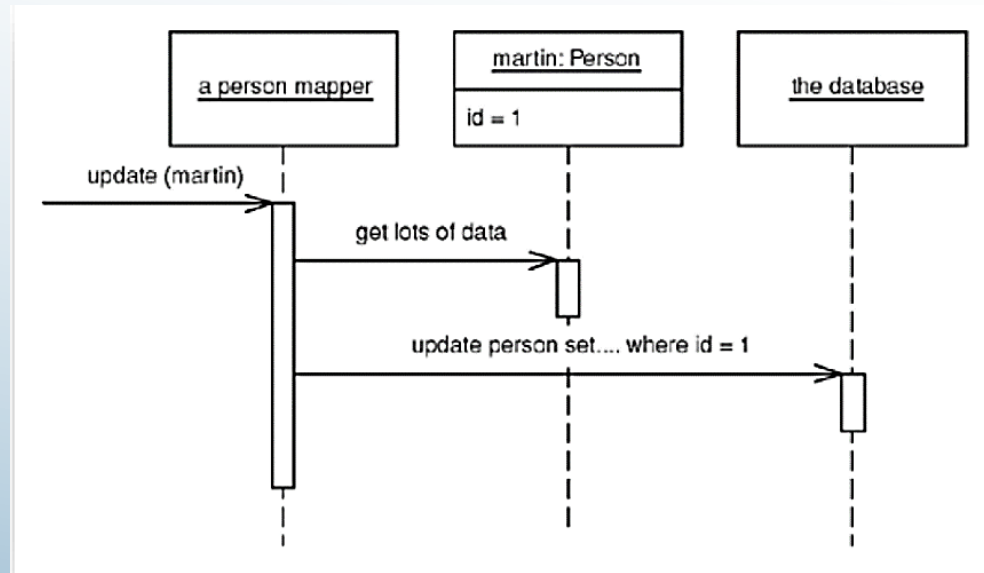
```
public Money getExemption() {
    Money baseExemption = Money.dollars(1500);
    Money dependentExemption = Money.dollars(750);
    return baseExemption.add(dependentExemption.multiply(this.getNumberOfDependents(
    }
}
```

# Repetition of the last lecture

## Data mapper (when to use)

### Pros

- Domain layer and database layer are independent
- Domain model
- Complex domain logic
- More difficult to program



### Cons

- Not suitable for Transaction script and Table module

```
class PersonMapper...

protected String findStatement() {
    return "SELECT " + COLUMNS +
        " FROM people" +
        " WHERE id = ?";
}

public static final String COLUMNS = " id, lastname, firstname, number_of_dependent";
public Person find(Long id) {
    return (Person) abstractFind(id);
}

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



# Data Transfer Objects

An object that carries data between processes in to reduce the number of method calls.

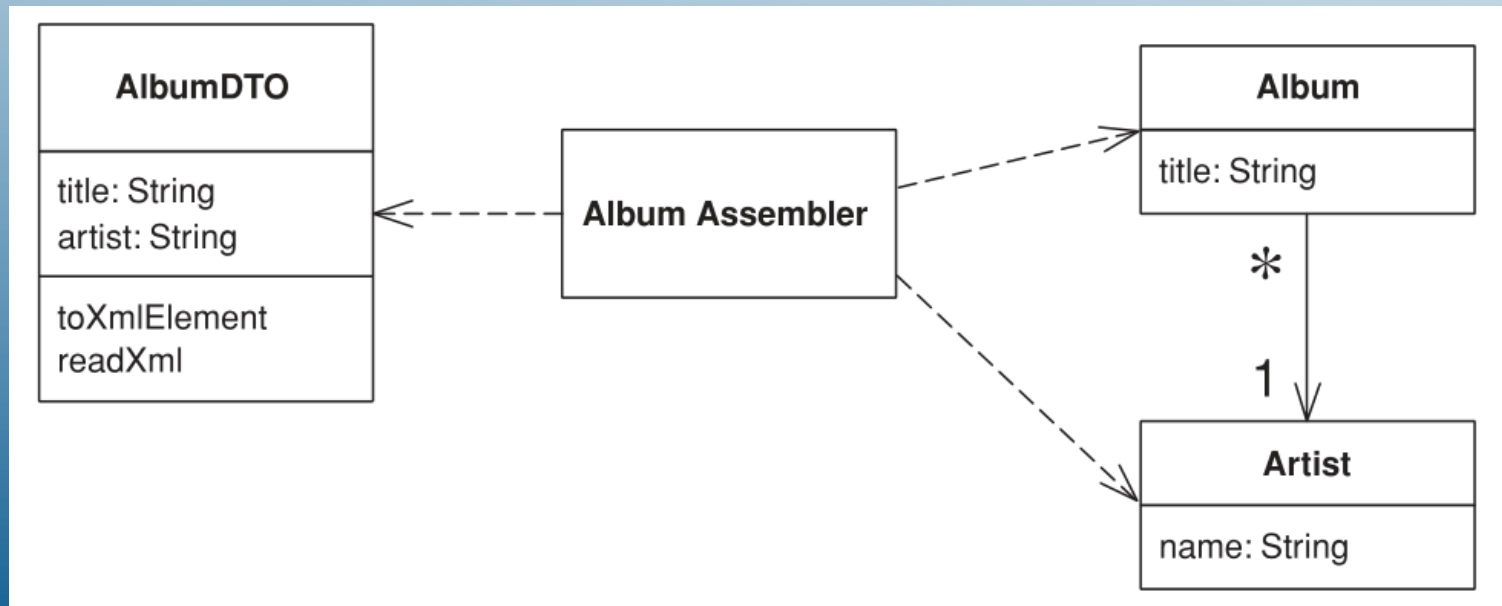
When you're working with a remote interface each call to it is expensive. As a result you need to reduce the number of calls, and that means that you need to transfer more data with each call. One way to do this is to use lots of parameters.

The solution is to create a Data Transfer Object that can hold all the data for the call. It needs to be serializable to go across the connection. Usually an assembler is used on the server side to transfer data between the DTO and any domain objects.



# Data Transfer Objects

A single Data Transfer Object usually contains more than just a single server (business) object. It aggregates data from all the server objects that the remote object is likely to want data from. Thus, if a remote object requests data about an order object, the returned Data Transfer Object will contain data from the order, the customer, the line items, the products on the line items, the delivery information, etc.



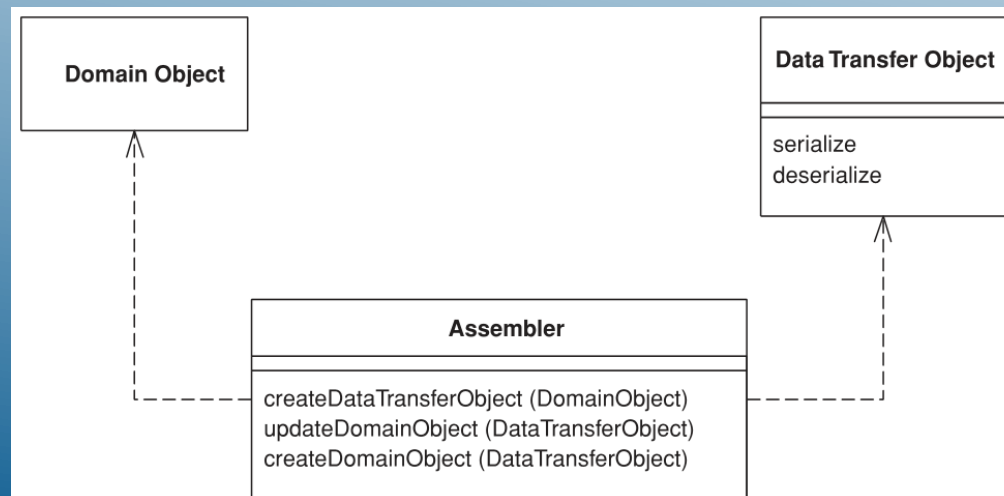
# Data Transfer Objects

- You can't usually transfer objects from a Domain Model (by its complexity).
- By DTO you transfer a simplified form of the data from the server (domain) objects.
- Structure between data transfer objects should be a simple graph structure (not as complicated as graph structure in Domain Model) – easy serialization.
- Data Transfer Object classes and any classes they reference must be present on both sides.
- DTO are often used for connecting to Web pages or GUI screens. You may also see multiple Data Transfer Objects for an order, depending on the particular screen.
- If different presentations require similar data, then it makes sense to use a single Data Transfer Object to handle them all.

# Data Transfer Objects

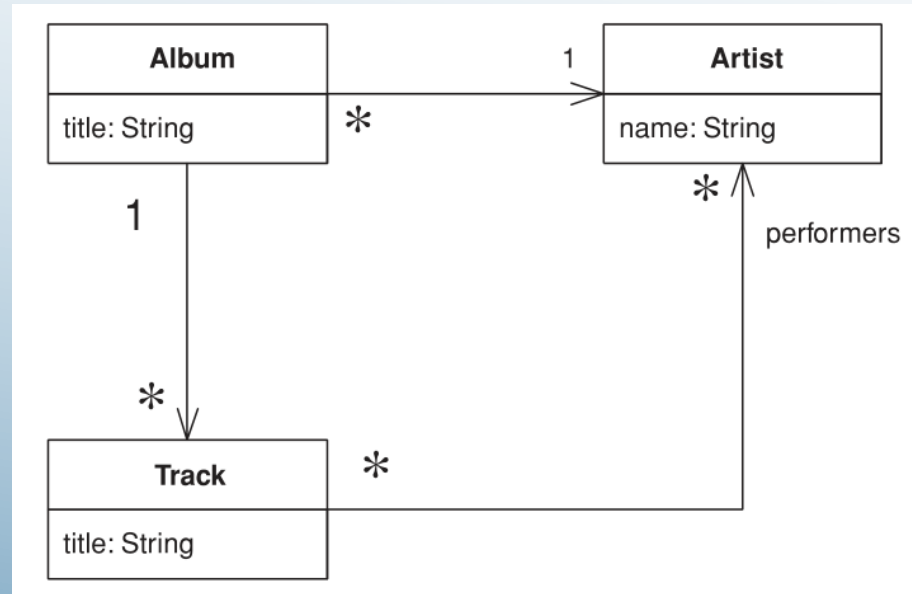
## Assembling a Data Transfer Object from Domain Objects

- A Data Transfer Object doesn't know about how to connect with domain objects.
- DTO should be deployed on both sides of the connection
- The domain objects to be dependent of the Data Transfer Object since the Data Transfer Object structure will change when I alter interface formats. As a general rule, we want to keep the domain model independent of the external interfaces.

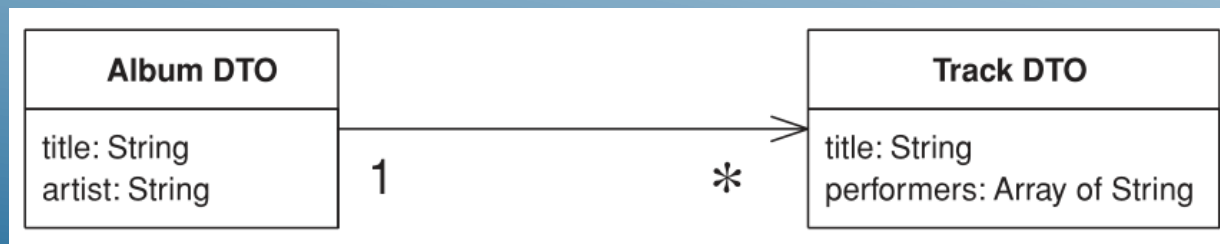


# Example: Transferring Information About Albums


The data we want to transfer is the data about these linked domain classes:



The data transfer objects simplify this structure a good bit.




# Data, Data, Data ....

- **Persistence** - programs change, data stay the same.
  - **Big data** - lots of data (memory is not enough to hold it all).
  - **Availability** - access to data (many users from different places, concurrent access).
- 
- Three parallel white lines of varying lengths are positioned in the bottom right corner of the slide, slanted diagonally upwards from left to right.

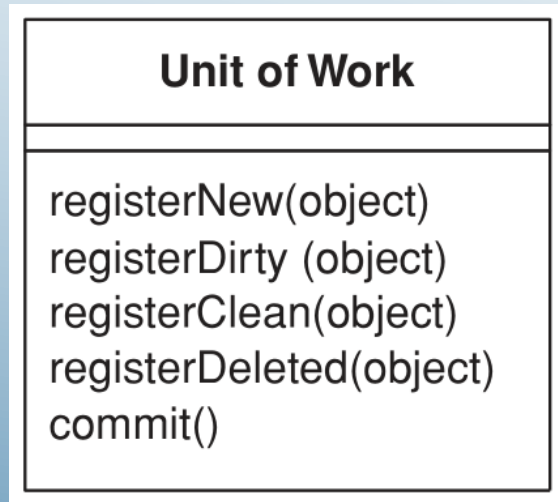
# Object-Relational Behavioral Patterns

These patterns are designed to support and reduce problems that are inherent in data source patterns. These problems include managing of transactions and managing of in-memory objects to avoid duplication and preserve data integrity.

- Unit of Work (UoW)
  - Identity Map (IM)
  - Lazy Load (LL)
- 
- A series of three parallel white diagonal lines in the bottom right corner of the slide, extending from the middle of the right edge towards the bottom left.

# 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.



A Unit of Work keeps track of everything you do during a business transaction that can affect the database. When you're done, it figures out everything that needs to be done to alter the database as a result of your work.




# Unit of Work – How it works

The obvious things that cause you to deal with the database are changes: new object created and existing ones updated or deleted.

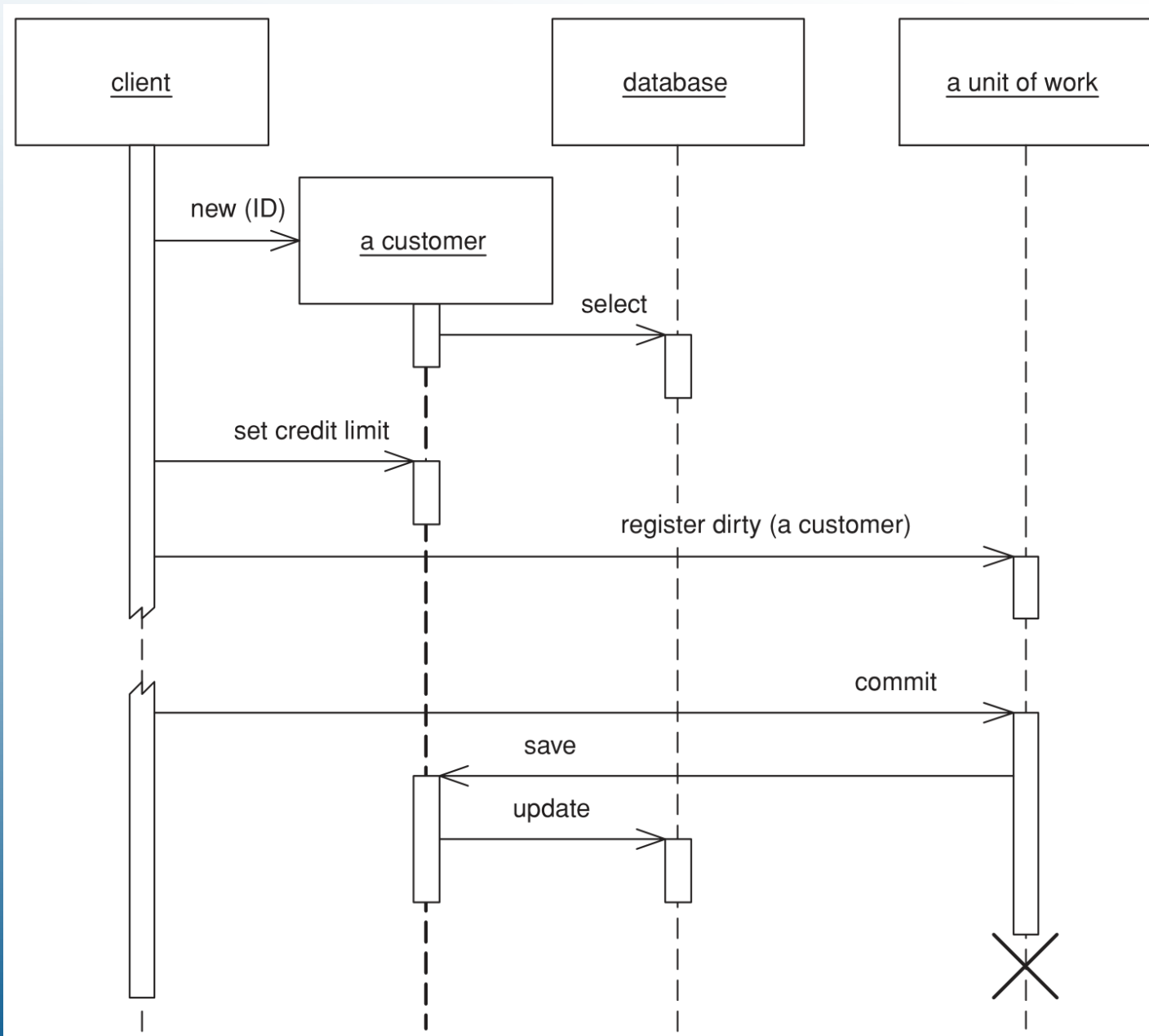
1. Unit of Work is an object that keeps track of these things. As soon as you start doing something that may affect a database, you create a Unit of Work to keep track of the changes.
2. Every time you create, change, or delete an object you tell the UoW. You can also let it know about objects you've read so that it can check for inconsistent reads by verifying that none of the objects changed on the database during the business transaction.

# Unit of Work – How it works

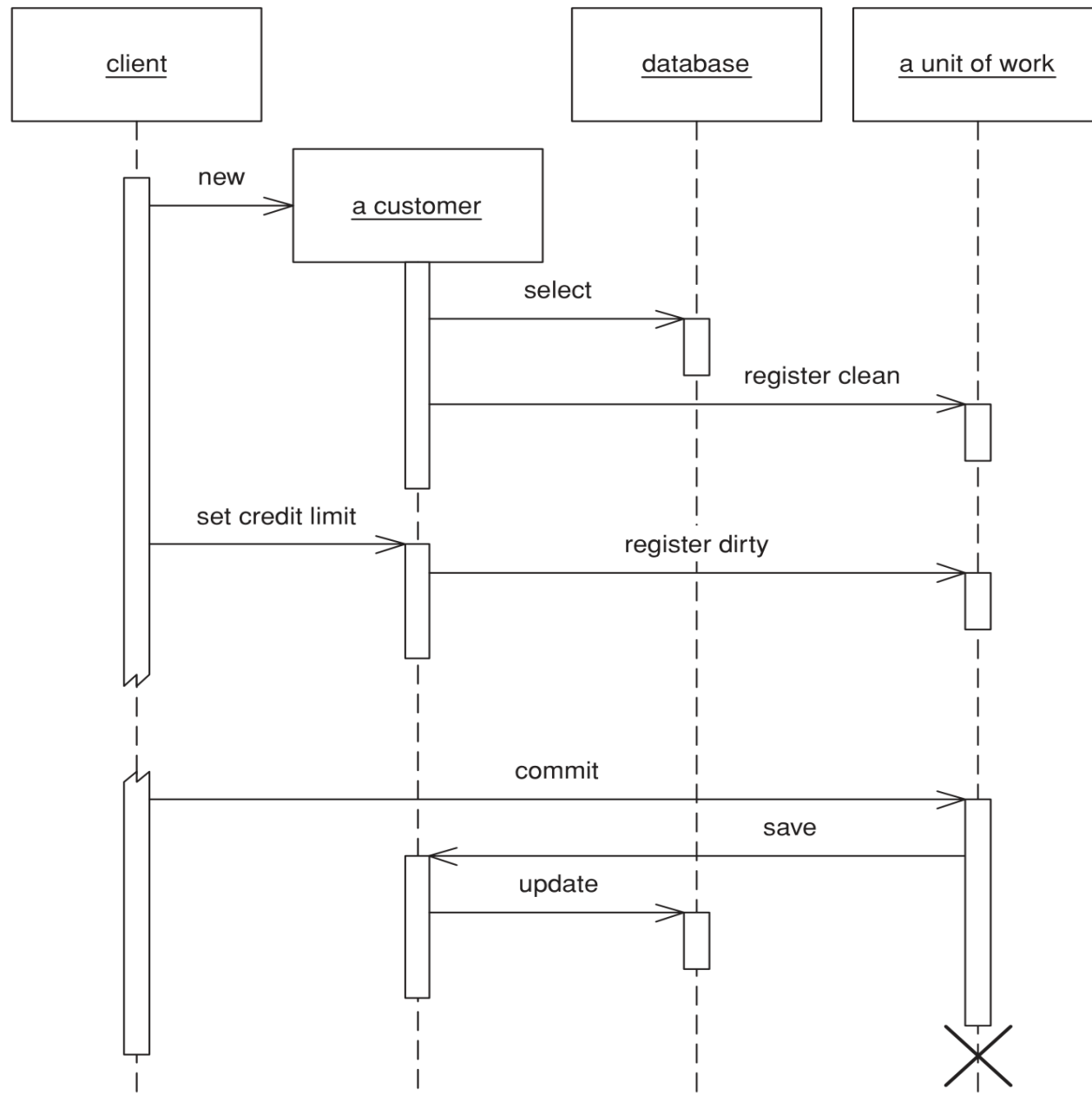
3. The key thing about Unit of Work is that, when it comes time to commit, the Unit of Work decides what to do. It opens a transaction, does any concurrency checking, and writes changes out to the database.
  4. Application programmers never explicitly call methods for database updates. This way they don't have to keep track of what's changed or worry about how referential integrity affects the order in which they need to do things
- 

# Unit of Work

Caller register a changed object



# Unit of Work Getting the receiver object to register itself



# Unit of Work advanced usage

- UoW can be helpful in update order when a database uses referential integrity. It is the natural place to sort out the update order.
- UoW can be used to minimize deadlocks. If every transaction uses the same sequence of tables to edit, you greatly reduce the risk of deadlocks. The Unit of Work is an ideal place to hold a fixed sequence of table writes so that you always touch the tables in the same order.
- UoW makes an obvious point of handling batch updates. (The idea behind a batch update is to send multiple SQL commands as a single unit so that they can be processed in a single remote call.)

# Unit of Work implementation

- In **.NET** the Unit of Work is often done by the disconnected data set. This makes it a slightly different pattern from the classical variety. Most Units of Work come across register and track changes to objects. .NET reads data from the database into a data set, which is a series of objects arranged like database tables, rows, and columns.
- **JAVA** - To store the change set we use three lists: new, dirty, and removed domain objects.

```
class UnitOfWork...
```

```
    private List newObjects = new ArrayList();  
    private List dirtyObjects = new ArrayList();  
    private List removedObjects = new ArrayList();
```

# 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.

- An Identity Map keeps a record of all objects that have been read from the database in a single business transaction. Whenever you want an object, you check the Identity Map first to see if you already have it.
- Related to this is an obvious performance problem. If you load the same data more than once you're incurring an expensive cost in remote calls.



# Identity Map – How it Woks

1. The idea behind the Identity Map is to have a series of maps containing objects that have been pulled from the database.
2. When you load an object from the database, you first check the map. If there's an object in it that corresponds to the one you're loading, you return it.
3. If not, you go to the database, putting the objects into the map for future reference as you load them.

# Identity Map – How it Woks

**Choice of Keys:** The obvious choice is the primary key of the corresponding database table. This works well if the key is a single column and immutable. The key will usually be a simple data type so the comparison behavior will work nicely.

**Explicit or Generic:** An explicit IM is accessed with distinct methods for each kind of object you need (`findPerson(1)`). A generic map uses a single method for all kinds of objects, with a parameter to indicate which kind of object you need (`find("Person", 1)`)

# Identity Map – How it Woks

**How Many:** A single map for the session works only if you have database-unique keys. Using one map per class or per table works well if your database schema and object models are the same, otherwise base the maps on your objects.

**Where to Put Them:** You need to put the Identity Map on a session-specific object. If you're using Unit of Work that's by far the best place for the Identity Maps since the Unit of Work is the main place for keeping track of data coming in or out of the database.

# Identity Map - implementation

For each Identity Map we have a map field and accessors.

```
private Map people = new HashMap();  
public static void addPerson(Person arg) {  
    soleInstance.people.put(arg.getID(), arg);  
}  
public static Person getPerson(Long key) {  
    return (Person) soleInstance.people.get(key);  
}  
public static Person getPerson(long key) {  
    return getPerson(new Long(key));  
}
```

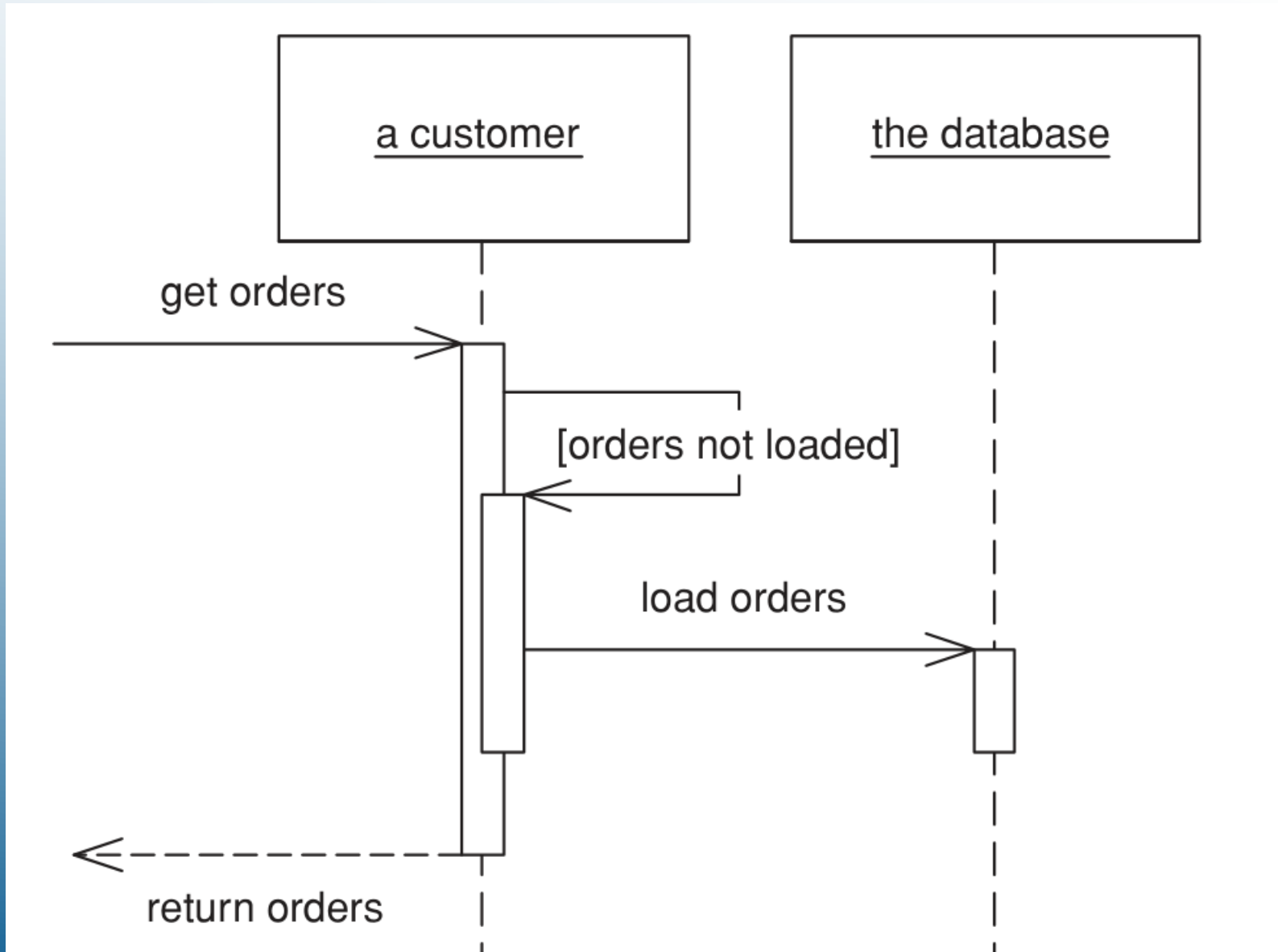
# Lazy Load

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

For loading data from a database into memory it's handy to design things so that as you load an object of interest you also load the objects that are related to it.

A Lazy Load interrupts loading process for the moment, leaving a marker in the object structure so that if the data is needed it can be loaded only when it is used.

# Lazy Load – How it Works



# Lazy Load – How it Works

There are four main ways you can implement Lazy Load:

- Lazy initialization
- Virtual proxy
- Value holder
- Ghost





# Lazy Load – How it Works

## Lazy initialization

- The basic idea is that 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.
- To make this work you have to ensure that the field is self-encapsulated, meaning that all access to the field, even from within the class, is done through a getting method.
- Active Records, Table Data Gateway, Row Data Gateway

# Lazy Load – How it Works

## Virtual proxy

- A virtual proxy is an object that looks like the object that should be in the field, but it is empty. Only when one of its methods is called does it load the correct object from the database.
- A virtual proxy is that it looks exactly like the object that's supposed to be there.
- The bad thing is that it isn't that object, so you can easily run into a nasty identity problem.
- Used with Data Mapper to hide dependency

# Lazy Load – How it Works

## Value Holder:

- A Value Holder is an object that wraps some other object.
- To get the underlying object you ask the Value Holder for its value, but only on the first access does it pull the data from the database.
- To avoid identity problems ensure that the value holder is never passed out beyond its owning class.
- Data Mapper

# Lazy Load – How it Works

## Ghost:

- A Ghost is the real object in a partial state. 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.
- Every field is lazy-initialized in one fell swoop, or as a virtual proxy, where the object is its own virtual proxy.
- There's no need to load all the data in one go; you may group it in groups that are commonly used together.
- Well used with Data Mapper

# Lazy Load – Lazy initialization implementation

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

The first access of the products field causes the data to be loaded from the database.

Three parallel white diagonal lines are located in the bottom right corner of the slide, extending from the middle of the right edge towards the bottom left.

# Lazy Load – Virtual proxy implementation

```
class SupplierMapper...
```

```
public static class ProductLoader implements VirtualListLoader {  
    private Long id;  
    public ProductLoader(Long id) {  
        this.id = id;  
    }  
    public List load() {  
        return ProductMapper.create().findForSupplier(id);  
    }  
}
```

```
class VirtualList...
```

```
private List source;  
private VirtualListLoader loader;  
public VirtualList(VirtualListLoader loader) {  
    this.loader = loader;  
}  
private List getSource() {  
    if (source == null) source = loader.load();  
    return source;  
}
```

```
class SupplierMapper...
```

```
protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException {  
    String nameArg = rs.getString(2);  
    SupplierVL result = new SupplierVL(id, nameArg);  
    result.setProducts(new VirtualList(new ProductLoader(id)));  
    return result;  
}
```

# Lazy Load – Value Holder implementation

```
class SupplierVH...  
  
    private ValueHolder products;  
    public List getProducts() {  
        return (List) products.getValue();  
    }
```

```
class ValueHolder...  
  
    private Object value;  
    private ValueLoader loader;  
    public ValueHolder(ValueLoader loader) {  
        this.loader = loader;  
    }  
    public Object getValue() {  
        if (value == null) value = loader.load();  
        return value;  
    }  
    public interface ValueLoader {  
        Object load();  
    }
```



# Lazy Load – Ghost implementation

```
class Domain Object...

    LoadStatus Status;
    public DomainObject (long key) {
        this.Key = key;
    }
    public Boolean IsGhost {
        get {return Status == LoadStatus.GHOST;}
    }
    public Boolean IsLoaded {
        get {return Status == LoadStatus.LOADED;}
    }
    public void MarkLoading() {
        Debug.Assert(IsGhost);
        Status = LoadStatus.LOADING;
    }
    public void MarkLoaded() {
        Debug.Assert(Status == LoadStatus.LOADING);
        Status = LoadStatus.LOADED;
    }
    enum LoadStatus {GHOST, LOADING, LOADED};
```

# Lazy Load – Ghost implementation

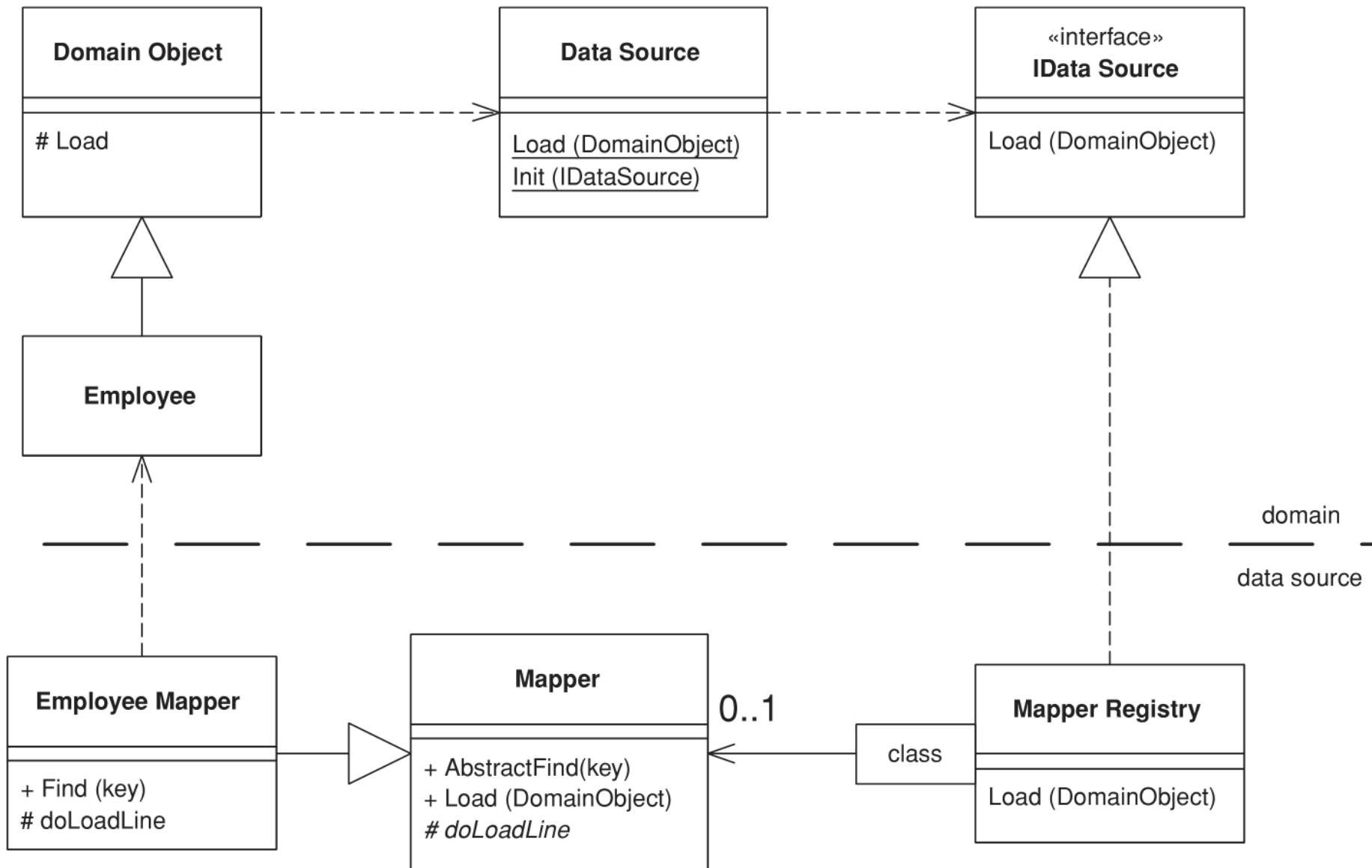
```
class Employee...
```

```
    public String Name {  
        get {  
            Load();  
            return _name;  
        }  
        set {  
            Load();  
            _name = value;  
        }  
    }  
    String _name;
```

```
class Domain Object...
```

```
    protected void Load() {  
        if (IsGhost)  
            DataSource.Load(this);  
    }
```

# Lazy Load – Ghost implementation



# Semestral project -Artifact 4

## UI: Sketch (wireframe, prototype)

- A design defining the function and content of forms.
- The layout of the functional elements on the page.
- Design of navigation (from where - to where - how).
- Simply (without final design).
- Use drawing tools or simply paper and pencil.

# Exercise tasks

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

- Discussion of use-case models
- Discussion of technical requirements
- Implementation of 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

- For each of the four design patterns for working with data sources, think about it and write a piece of source code that tells you which pattern it is.
- What do the object-relational behavioral patterns address together?
- When should we consider using the Unit of Work pattern and why? Explain, with an example, how to use it.
- When should we consider using the Identity Map pattern and why? Explain, with an example, how to use it.
- When should we consider using the Lazy Load pattern and why? Using an example, explain how to use it.
- Think about which patterns for working with data sources you could use together with some of the behavioral patterns and why? Try to find examples.