Data Access Patterns

- Some of the problems with data access from OO programms:
  1. **Data source and OO program use different data modelling concepts**
  2. **Decoupling domain logic from database technology**

Problems in Data Access (2)

If business code directly accesses data sources:

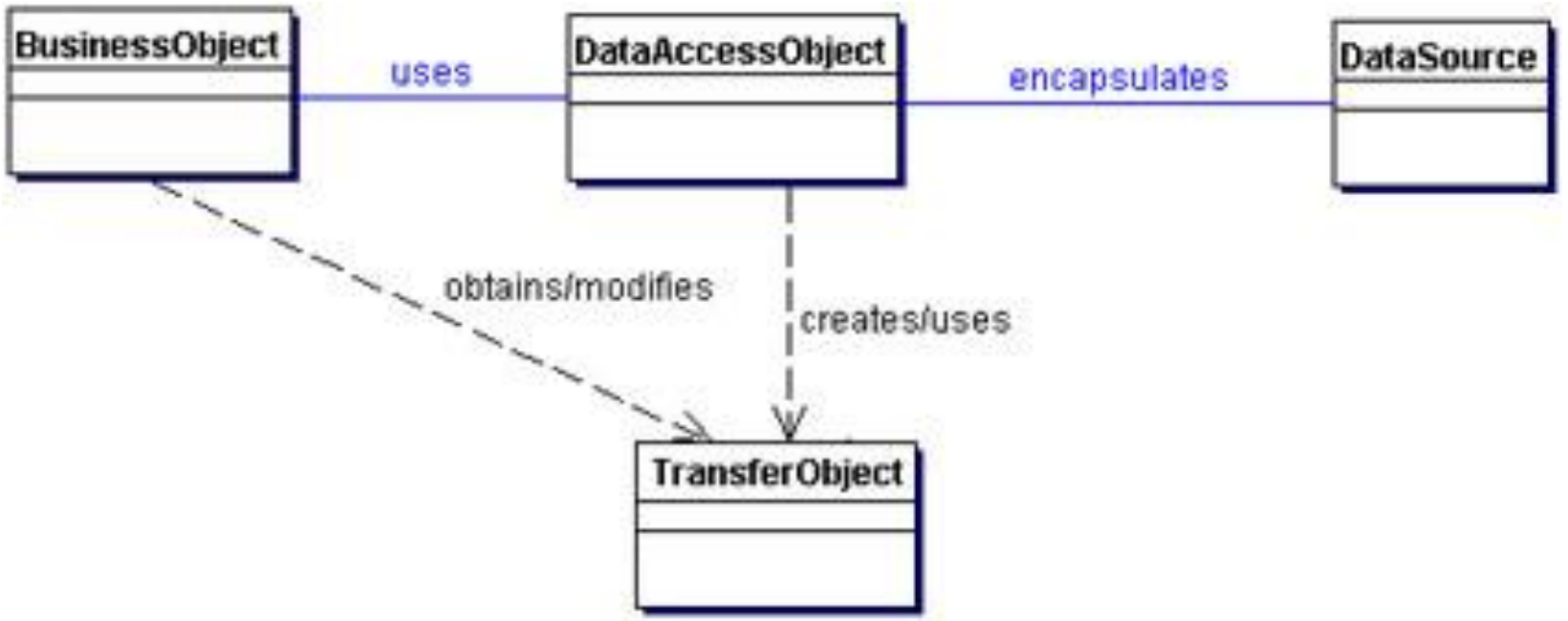
- It depends on specific APIs (e.g., JDBC, JPA, etc)
- It depends on specific databases/vendors (MySQL, Oracle, files, etc.)
- It mixes business logic + persistence logic

=> Result:

- Hard to maintain
- Hard to test
- Hard to migrate to another type of data source

The Data Access Object (DAO) Pattern

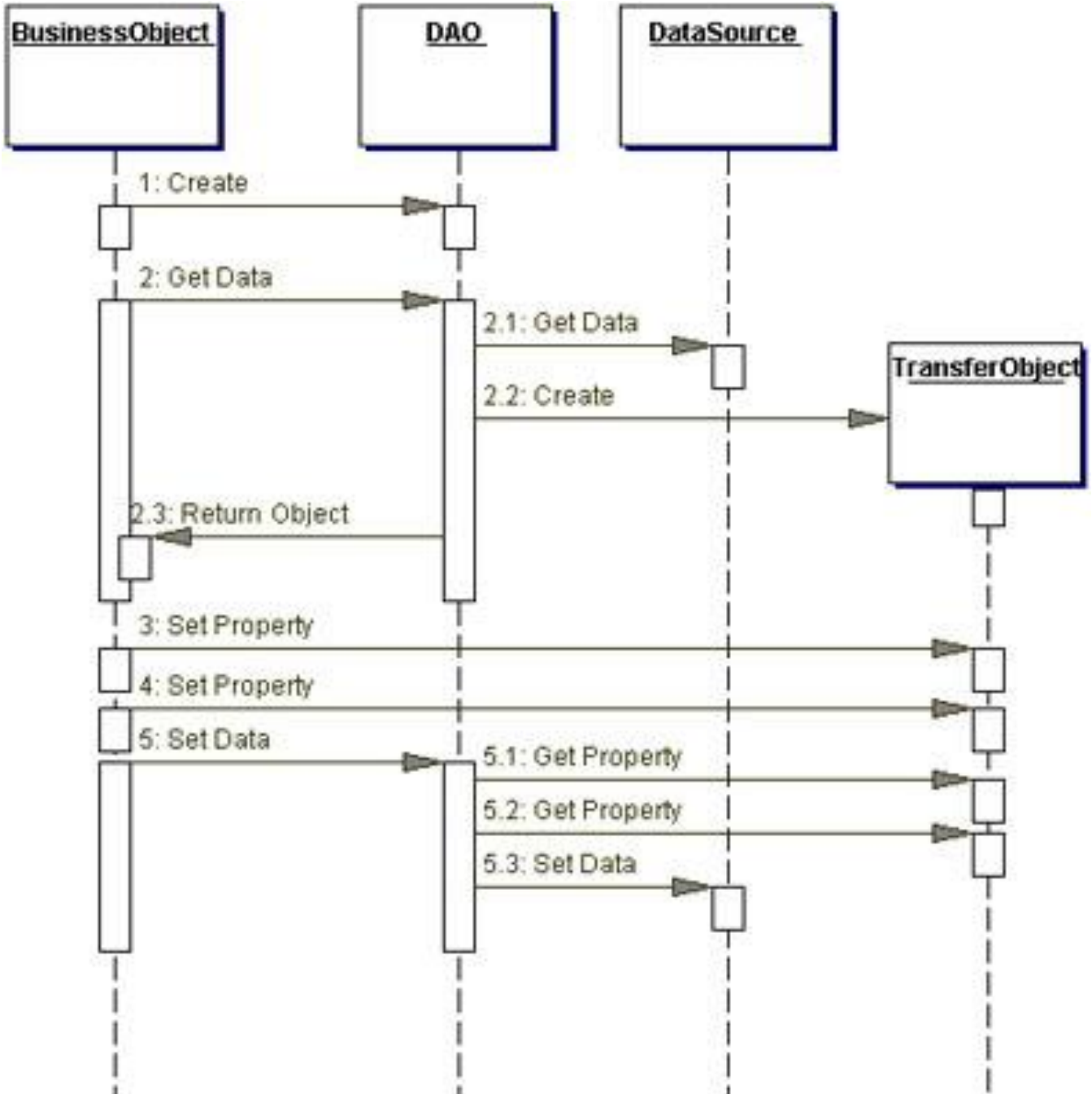
- **Intent:** **Abstract and Encapsulate all access to the data source**



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

DAO – Participants and responsibilities

- **BusinessObject**: represents the data client, the domain logic.
  - needs data, but should NOT know how data is stored
- **DataAccessObject**: Abstraction layer used by BusinessObject
  - Provides clean methods like: findById(); findAllWithAge(); save(); update()
  - Internally, the DAO implementation contains all the database queries, fileaccesses, API calls
- **DataSource:** represents the actual data storage
  - Could be a database such as an RDBMS, OODBMS, XML repository, flat file system, etc.
- **Transfer Object**: used as a data carrier.
  - Simple object used to carry data
  - Contains only fields and getters/setters
  - Must not reflect technology of Data source


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

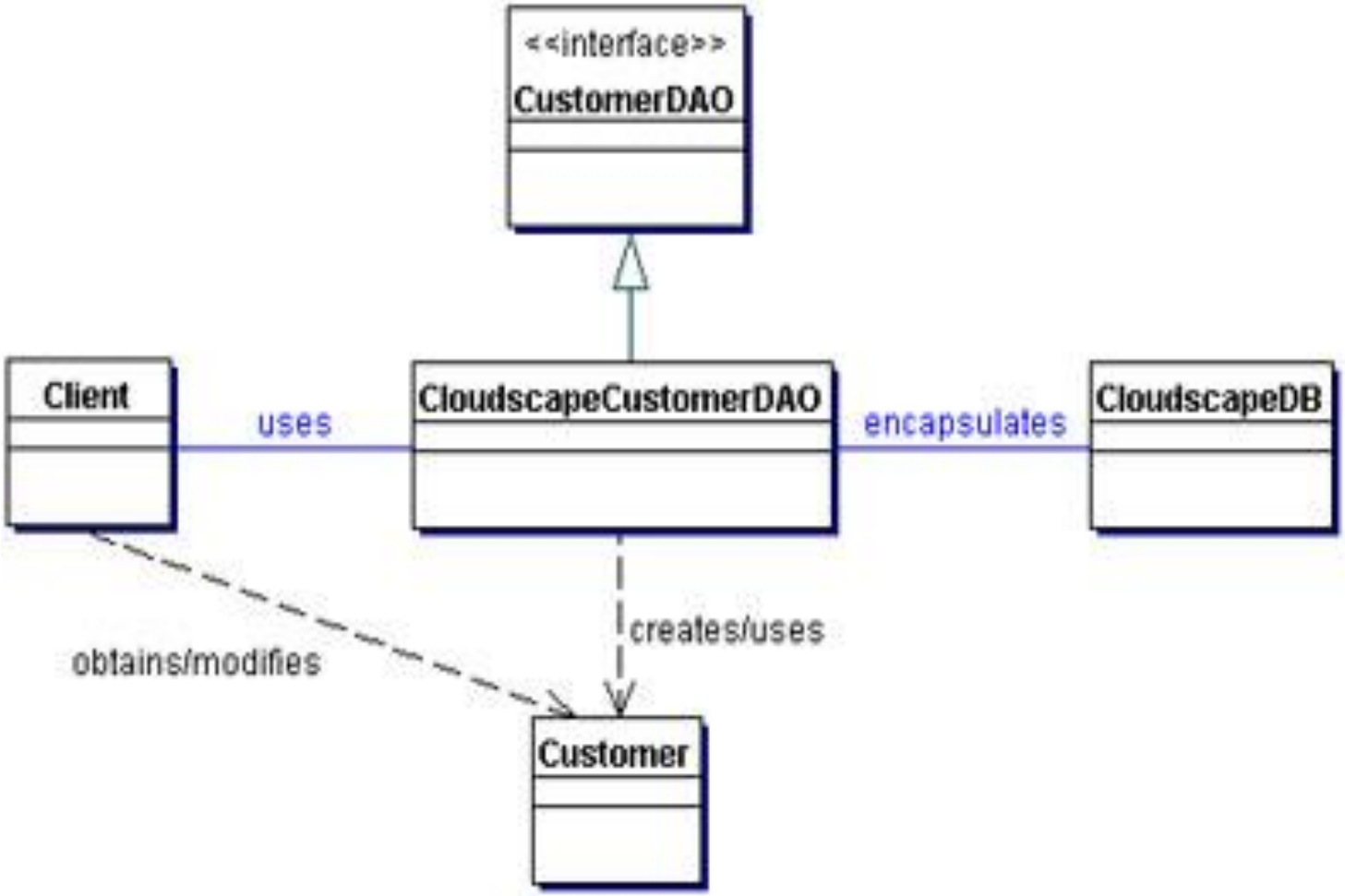
Where to get DAO's from ?

- Strategies to get DAO's:
  - **Manual implementation**:
    - Developer manually creates the DAO implementation classes
    - Often follows a **Factory pattern**
    - DAO Factory: A factory creates DAOs depending on environment
      - Example: OracleCustomerDAO, MySQLCustomerDAO
      - Business code asks factory -> gets correct DAO
    - Abstract Factory: Extends DAO Factory idea for multiple related DAOs
  - **Automatic DAO Code Generation Strategy**
    - Framework generates DAO implementations automatically (Example: Spring Data JPA)

Consequences

- DAO Pattern :
  - Advantages:
    - Enables Transparency
    - Enables Easier Migration
    - Reduces Code Complexity in Business Objects
  - Disadvantages:
    - Adds Extra Layer

Simple Example – Manual Implementation of DAO



Interface that all CustomerDAOs must support

```
// Interface that all CustomerDAOs must support
public interface CustomerDAO {
  public int insertCustomer(...);
  public boolean deleteCustomer(...);
  public Customer findCustomer(...);
  public boolean updateCustomer(...);
  public RowSet selectCustomersRS(...);
  public Collection selectCustomersTO(...);
  ...
}
```

Customer Transfer Object

```
public class Customer implements java.io.Serializable {
  // member variables
  int CustomerNumber;
  String name;
  String streetAddress;
  String city;
  ...

  // getter and setter methods...
  ...
}
```

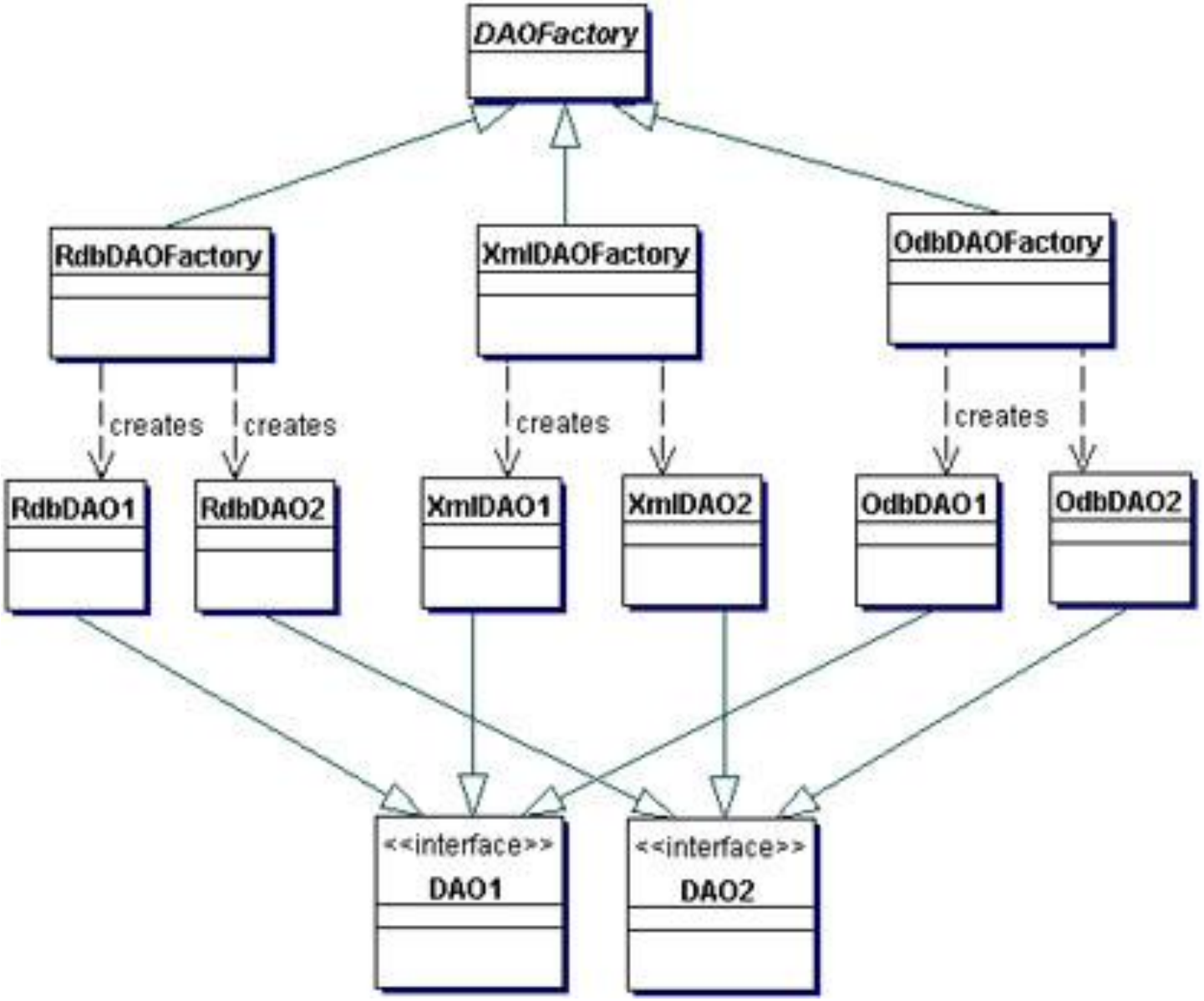
CloudscapeCustomerDAO implementation

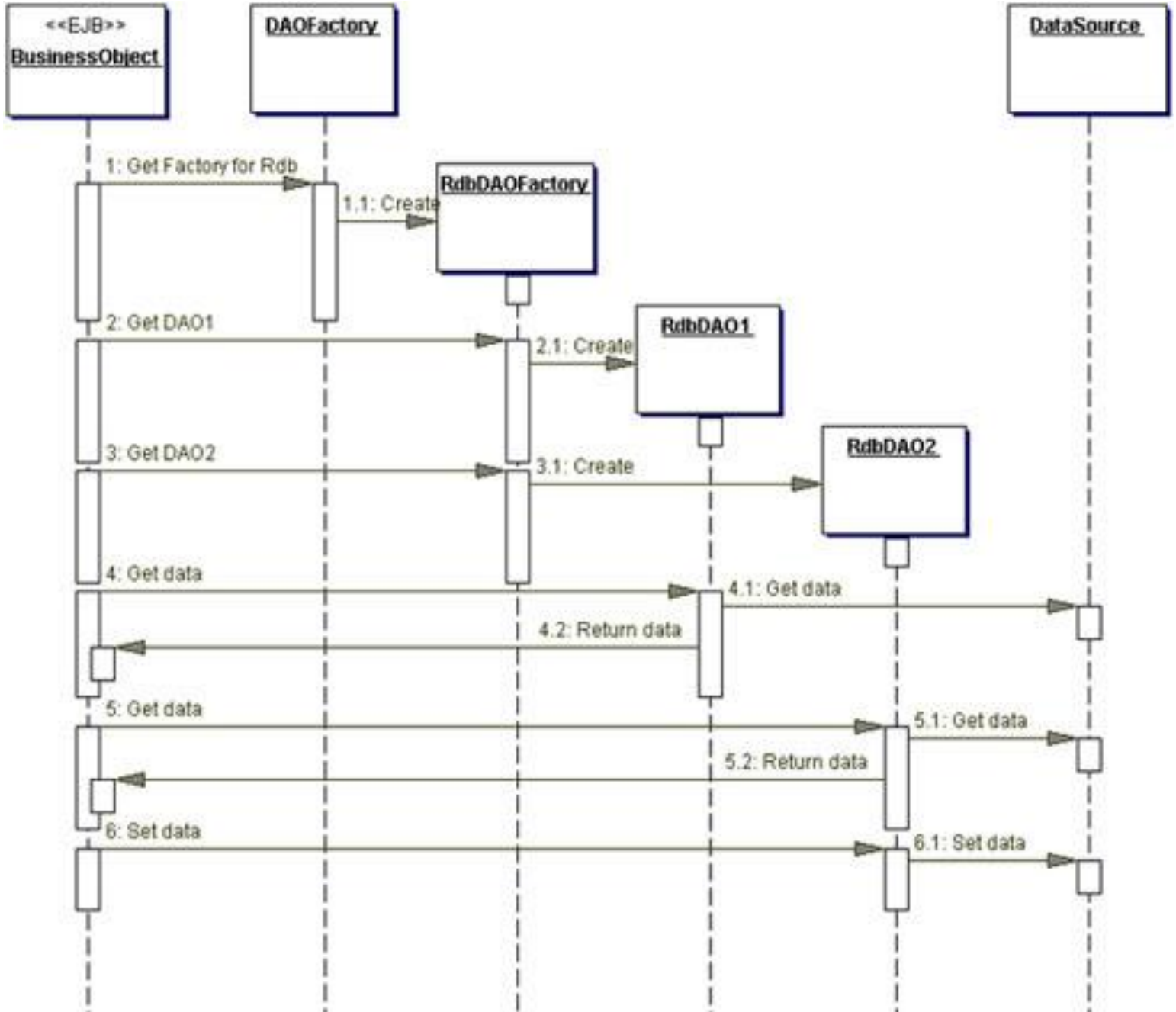
```
// CloudscapeCustomerDAO implementation of the CustomerDAO interface.
// This class can contain all  Cloudscape specific code and SQL statements.

import java.sql.*;

public class CloudscapeCustomerDAO implements CustomerDAO {
  public CloudscapeCustomerDAO() {
    // initialization
  }

  // The following methods can use  CloudscapeDAOFactory.createConnection()
  // to get a connection as required

  public int insertCustomer(...) {
    // Implement insert customer here
    // Return newly created customer number  or a -1 on error
  }
  public boolean deleteCustomer(...) {
    // Implement delete customer here
    // Return true on success, false on failure
  }
  public Customer findCustomer(...) {
    // Implement find a customer
    // Return a Transfer Object if found,  return null on error or if not found
  }
```

Extended Example – Manual Implementation with Abstract Factory

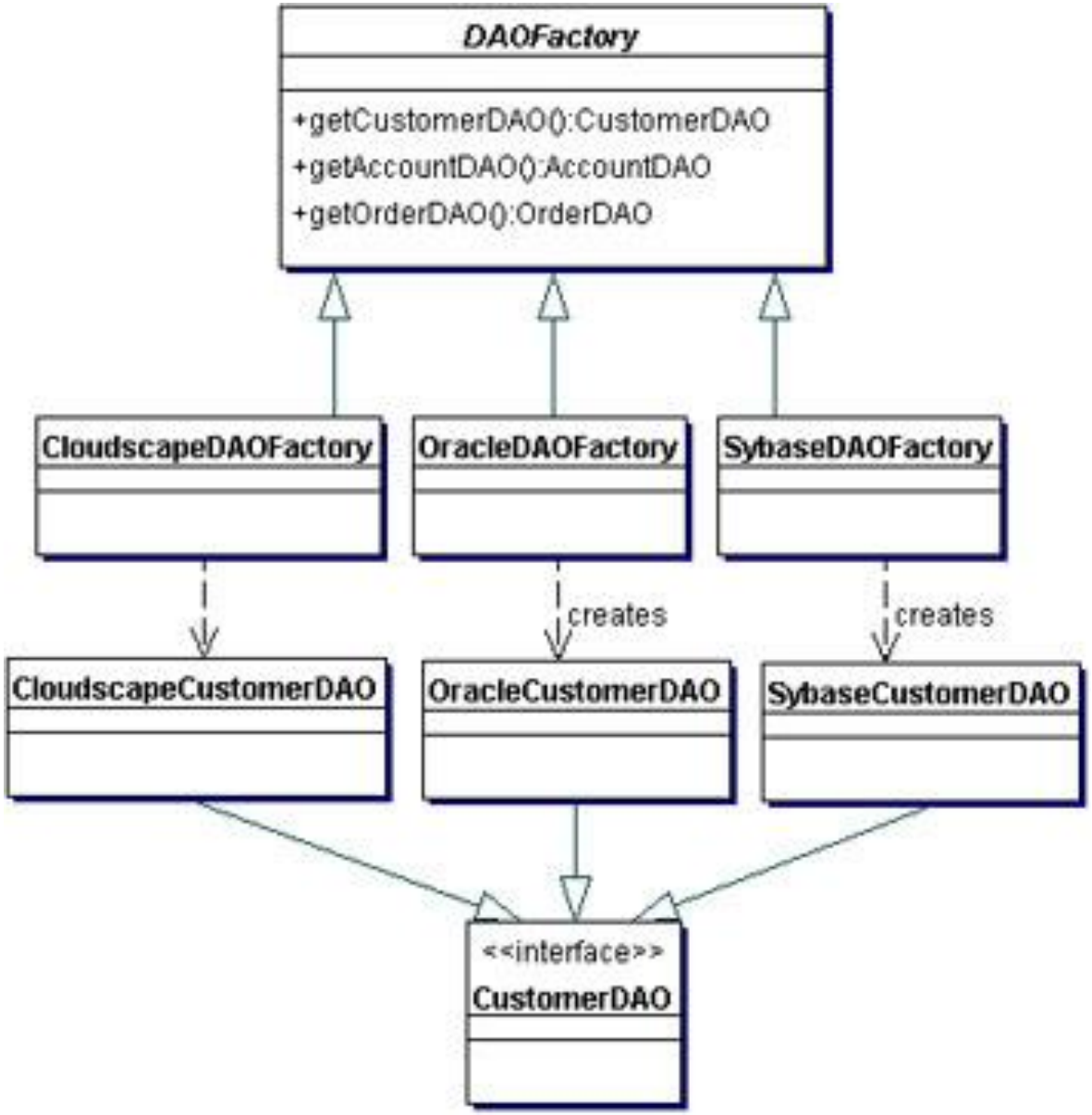
- Application uses more than one database:
  - Customer
  - Account
  - Order
  - => each database is abstracted in a DAO interface
- Database can be migrated on different technologies:
  - Cloudscape
  - Oracle
  - Sybase
  - => use **Abstract Factory Pattern:**
    - For each technology there is a concrete factory that can instantiate all DAOs for that technology

DAO with Abstract Factory




«EJB»
BusinessObject
DAOFactory
DataSource
1: Get Factory for Rdb
1.1: Create
RdbDAOFactory
2: Get DAO1
2.1: Create
RdbDAO1
3: Get DAO2
3.1: Create
RdbDAO2
4: Get data
4.1: Get data
4.2: Return data
5: Get data
5.1: Get data
5.2: Return data
6: Set data
6.1: Set data

Example- Using Abstract Factory



Abstract class DAO Factory

```
// Abstract class DAO Factory
public abstract class DAOFactory {

  // List of DAO types supported by the factory
  public static final int CLOUDSCAPE = 1;
  public static final int ORACLE = 2;
  public static final int SYBASE = 3;
  ...
  // There will be a method for each DAO that can be  created.
  //The concrete factories will have to  implement these methods.
  public  abstract CustomerDAO getCustomerDAO();
  public  abstract AccountDAO getAccountDAO();
  public  abstract OrderDAO getOrderDAO();
  ...
  public static DAOFactory getDAOFactory( int whichFactory) {
    switch (whichFactory) {
      case CLOUDSCAPE:
          return new CloudscapeDAOFactory();
      case ORACLE    :
          return new OracleDAOFactory();
      ...
      default           :
          return null;
    }
  }
}
```

Cloudscape concrete DAO Factory implementation

```
import java.sql.*;

public class CloudscapeDAOFactory extends DAOFactory {
  public static final String DRIVER=
    "COM.cloudscape.core.RmiJdbcDriver";
  public static final String DBURL=
    "jdbc:cloudscape:rmi://localhost:1099/CoreJ2EEDB";

  // method to create Cloudscape connections
  public static Connection createConnection() {
    // Use DRIVER and DBURL to create a connection
    // Recommend connection pool implementation/usage
  }
  public CustomerDAO getCustomerDAO() {
    // CloudscapeCustomerDAO implements CustomerDAO
    return new CloudscapeCustomerDAO();
  }
  public AccountDAO getAccountDAO() {
    // CloudscapeAccountDAO implements AccountDAO
    return new CloudscapeAccountDAO();
  }
  public OrderDAO getOrderDAO() {
    // CloudscapeOrderDAO implements OrderDAO
    return new CloudscapeOrderDAO();
  }
  ...
```

Client Code

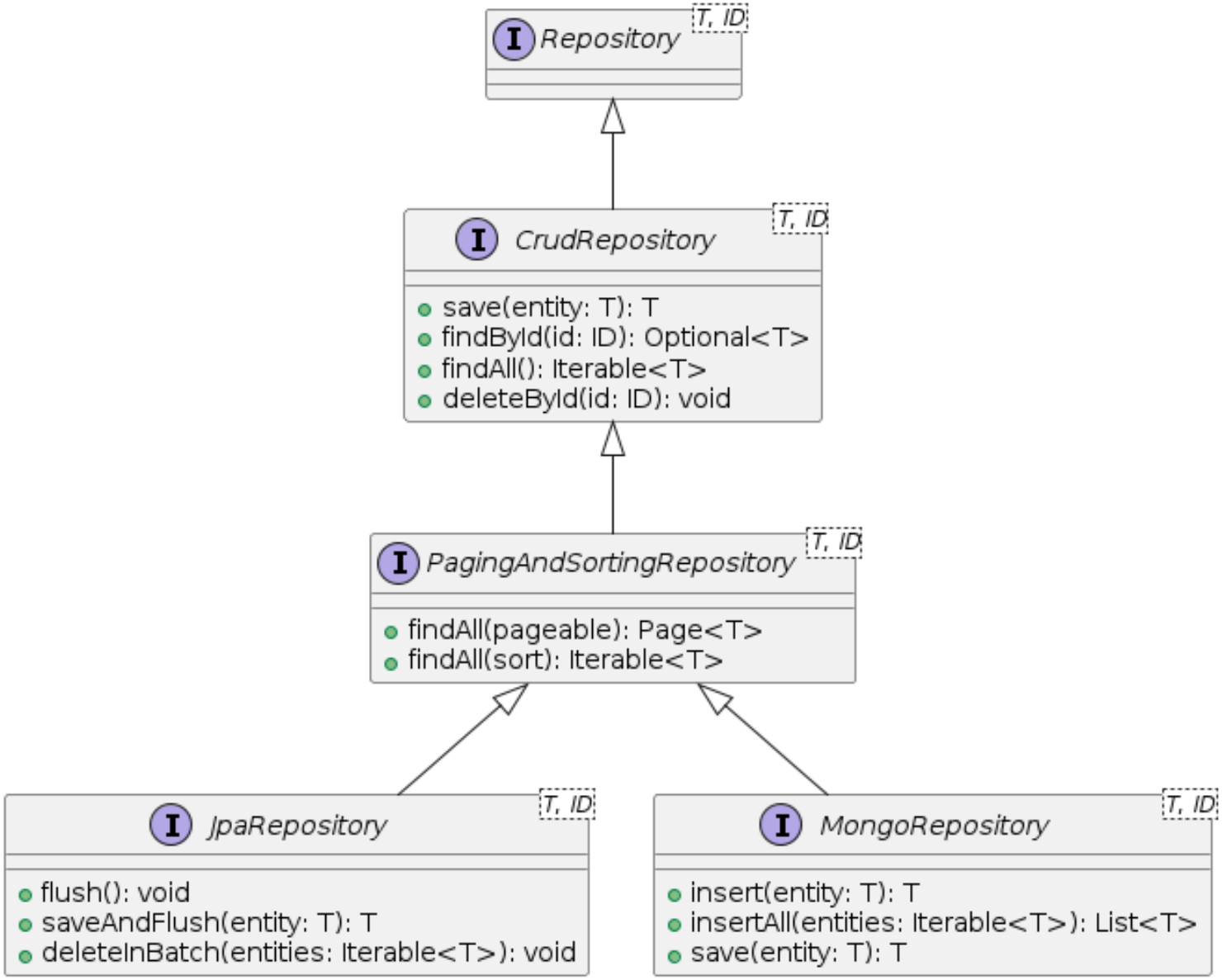
```
// Create a DAO
CustomerDAO custDAO = cloudscapeFactory.getCustomerDAO();

// create a new customer
int newCustNo = custDAO.insertCustomer(...);

// Find a customer object. Get the Transfer Object.
Customer cust = custDAO.findCustomer(...);

// modify the values in the Transfer Object.
cust.setAddress(...);
cust.setEmail(...);
// update the customer object using the DAO
custDAO.updateCustomer(cust);

// delete a customer object
custDAO.deleteCustomer(...);
// select all customers in the same city
Customer criteria=new Customer();
criteria.setCity("New York");
Collection customersList = custDAO.selectCustomersTO(criteria);
// returns customersList - collection of Customer
// Transfer Objects. iterate through this collection to
// get values.

...
```

Automatic generation of DAO Implementations

- Example: **Spring Data JPA**
- https://docs.spring.io/spring-data/jpa/docs/current/reference/html/
- **Spring Data JPA can automatically generate DAO implementations**, meaning developers only need to define interfaces
- Spring Data JPA is ***one*** of the possible technology stacks with Spring Data
- Spring Data is basically a **wrapper layer on top of different data-access technologies:** it doesn't replace them but it **delegates to them.** Different Spring Data modules:
  - Spring Data JPA: ->JPA -> Hibernate -> JDBC -> Database
  - Spring Data JDBC: ->JDBC Template -> JDBC -> Database
  - Spring Data MongoDB: -> MongoDB driver -> MongoDB Server

Spring Data – Repository Concept

- Spring Data introduces the core concept of **Repository**: A Spring Data Repository is similar to a **DAO Interface**
- A Repository is a generic interface that takes the entity type and the ID type as type arguments.
- A Repository interface will contain basic CRUD operations plus custom queries
- Different types of Repositories:
  - Repository -> CrudRepository -> JpaRepository
    -> MongoRepository


Repository
T, ID
CrudRepository
T, ID
save(entity: T): T
findById(id: ID): Optional<T>
findAll(): Iterable<T>
deleteById(id: ID): void
PagingAndSortingRepository
T, ID
findAll(pageable): Page<T>
findAll(sort): Iterable<T>
JpaRepository
T, ID
flush(): void
saveAndFlush(entity: T): T
deleteInBatch(entities: Iterable<T>): void
MongoRepository
T, ID
insert(entity: T): T
insertAll(entities: Iterable<T>): List<T>
save(entity: T): T

Generating Repositories

- Spring Data reduces boilerplate code for database access. Developers only define the **DAO interface** - implementation is generated automatically
  - Limitation: **method names must follow some naming conventions (**countBy, deleteBy, findBy - method name contains the query) -> only in this way the framework can generate the class implementing them
- Any Spring Data module that has a repository implementation + query derivation engine supports method generation
- Modules that support method generation:
  - Spring Data JPA: uses JPA and Hibernate: strongest query generation support
  - Spring Data MongoDB
  - Spring Data JDBC: simple queries with no joins

Spring Data JpaRepository

- DAO interface extends the **JpaRepository** interface:
  - Some CRUD operations do not need to be defined any more: save(), findById(), findAll(), deleteById(), existsById(), count()
  - Application developer still needs to declare custom queries as methods in interface
    - Examples: `findByLastname`, `findByLastnameAndFirstname`, `countByLastname`, `deleteByLastname`
    - Queries may involve attributes of entity class and also attributes of related (joined) entities
  - Application developer does not need to provide implementation class !

Example – Using JpaRepository

```
@Entity
class Person {
    @Id
    @GeneratedValue(strategy = Generatic
    private Long id;
    private String firstname;
    private String lastname;
   // …
}
```

Interface PersonRepository is the DAO interface. This interface is written by the application developer. It extends JpaRepository, this means the implementation of this interface is generated

```
interface PersonRepository extends JpaRepository<Person, Long> {

    List<Person> findByLastname(String lastname);

    long countByLastname(String lastname);

    long deleteByLastname(String lastname);

    List<Person> findByFirstnameAndLastname(String firstname, String
lastname);

     List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname,
String firstname);
…
}
```

Example – Using JpaRepository

```
@Entity
class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstname;
    private String lastname;
   // …
}


interface PersonRepository ext

    List<Person> findByLastname(                    lastname);

    long countByLastname(String lastname);

    long deleteByLastname(String lastname);

    List<Person> findByFirstnameAndLastname(String firstname, String
lastname);

     List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname,
String firstname);
…
}
```

Method names must follow certain naming conventions (countBy, deleteBy, findBy - method name contains the query) such that DAO implementations can be automatically generated by the framework

Method names ->SQL Queries

```
List<Person> findByLastname(String lastname);
```

SELECT * FROM person WHERE lastname = ?

```
long countByLastname(String lastname);
```

SELECT COUNT(*) FROM person WHERE lastname = ?

```
List<Person> findByFirstnameAndLastname(String firstname,
                                        String lastname);
```

SELECT * FROM person WHERE firstname = ? AND lastname = ?;

DAO Pattern (Repository Pattern)

- Advantages
  - Domain-focused abstraction
    - Works with aggregates/entities, not tables
    - Uses business language (e.g.: findByNameAndCarsModel)
  - Hides persistence completely
    - Domain layer does not know SQL or storage type
  - Improves maintainability
    - Easier to change database or ORM without affecting domain logic
  - Encourages clean architecture / DDD
- Disadvantages
  - Less low-level control: harder to optimize specific SQL queries directly
  - Can hide performance issues: abstraction may lead to inefficient generated queries
  - Overhead for simple apps : overengineering for CRUD-only applications