Data Access Techniques

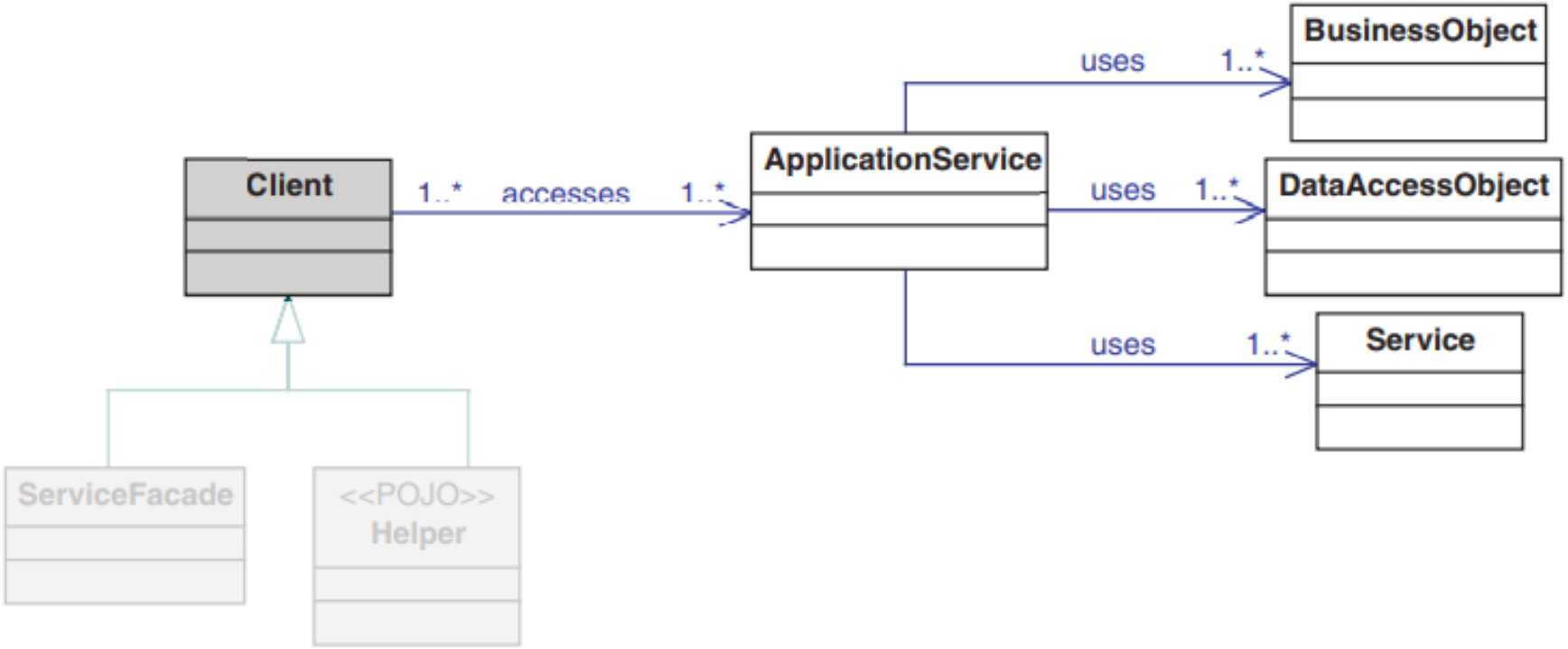
pm_jat @ daiict

Recap: Last Lecture

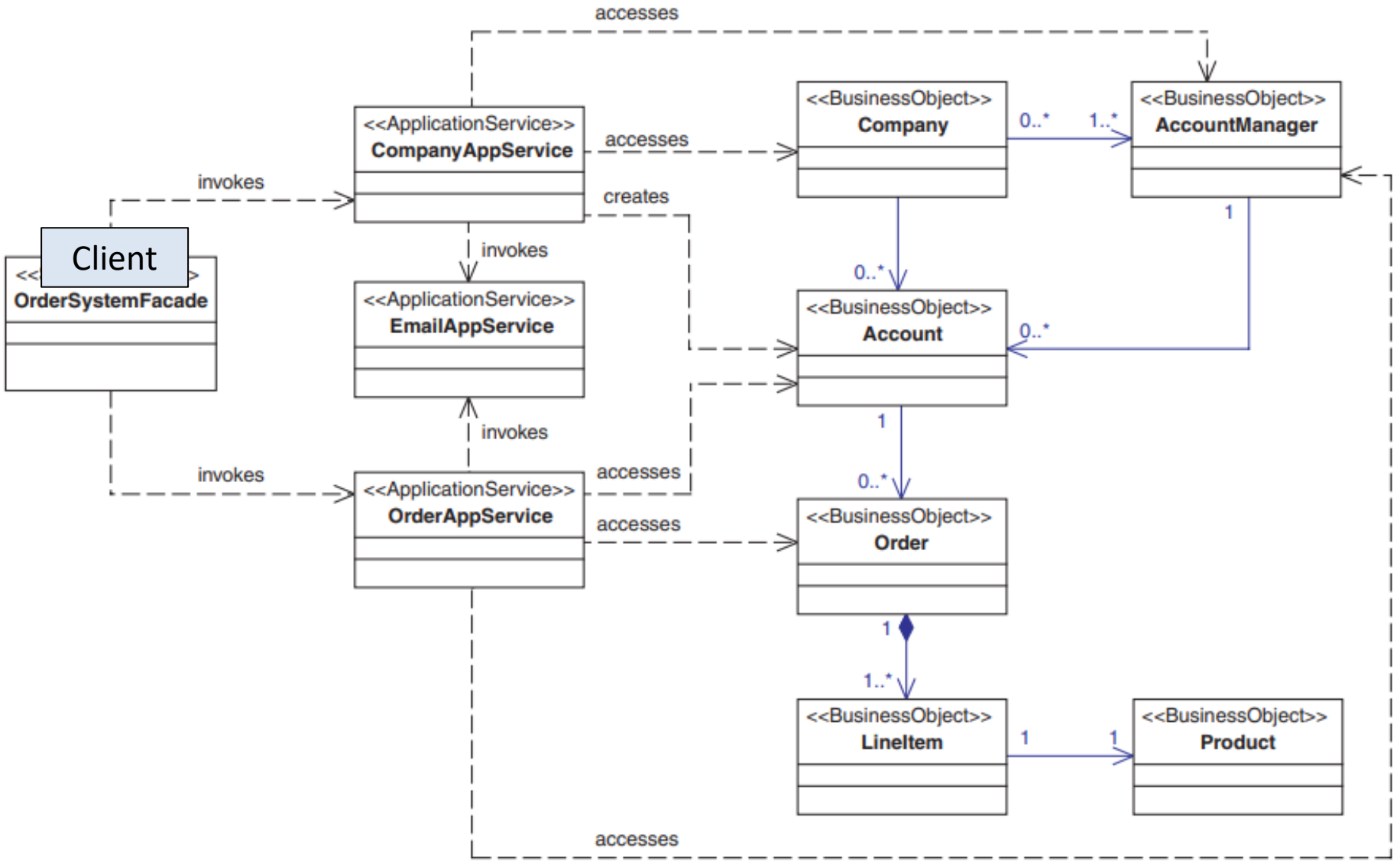
- We tried understanding
  - **Business Objects**: to represent data model
  - **Service Objects**:
    - Modules that implement “business logic” at larger granularity than Business Objects.
    - More of coordination work related to a set of use cases
    - Services can take services of another services
    - Services that are further coarser and used for performing actions for uses cases are called “application services”!

Application Service – Class Diagram[1]

- Here is a class diagram depicting a typical architecture of Application Service (source: book Core J2EE patterns [3])



Application Service Example [1]: Order System Class Diagram



Data Access Techniques

- Objective:
  - Data Access should be separate from Business Logic
  - Business Objects needs to be made persistent

- Here we discuss following three techniques:
  - Data Access Objects [1]
  - Domain Stores [1]
  - Java Persistence API (that tools like Hibernate implement)

Data Access Techniques

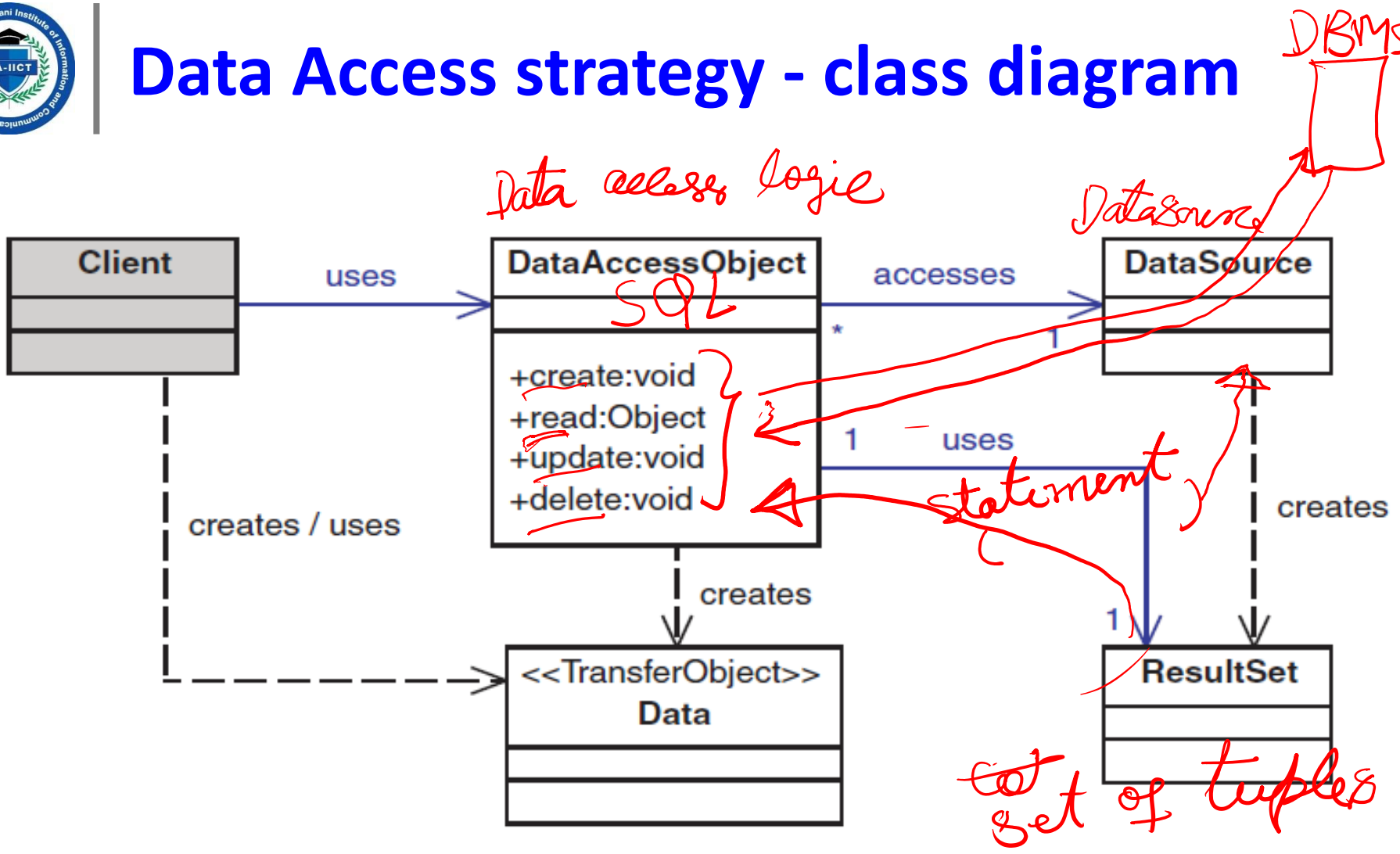
- **Objective**: Data Access should be separate from Business Logic

- Here we discuss a technique called "**Data Access Objects**" from book "**Core J2EE Patterns: Best Practices and Design Strategies**" by Deepak, Dan, and John [1]

Data Access Objects - Motivation

- We want to implement data access mechanisms to access and manipulate data in a persistent storage.
- We want to decouple the persistent storage implementation from the rest of your application.
- We want to provide a uniform data access API for a persistent mechanism to various types of data sources, such as RDBMS, LDAP, OODB, XML repositories, flat files, and so on.
- We want to organize data access logic and encapsulate proprietary features to facilitate maintainability and portability.

Data Access strategy - class diagram



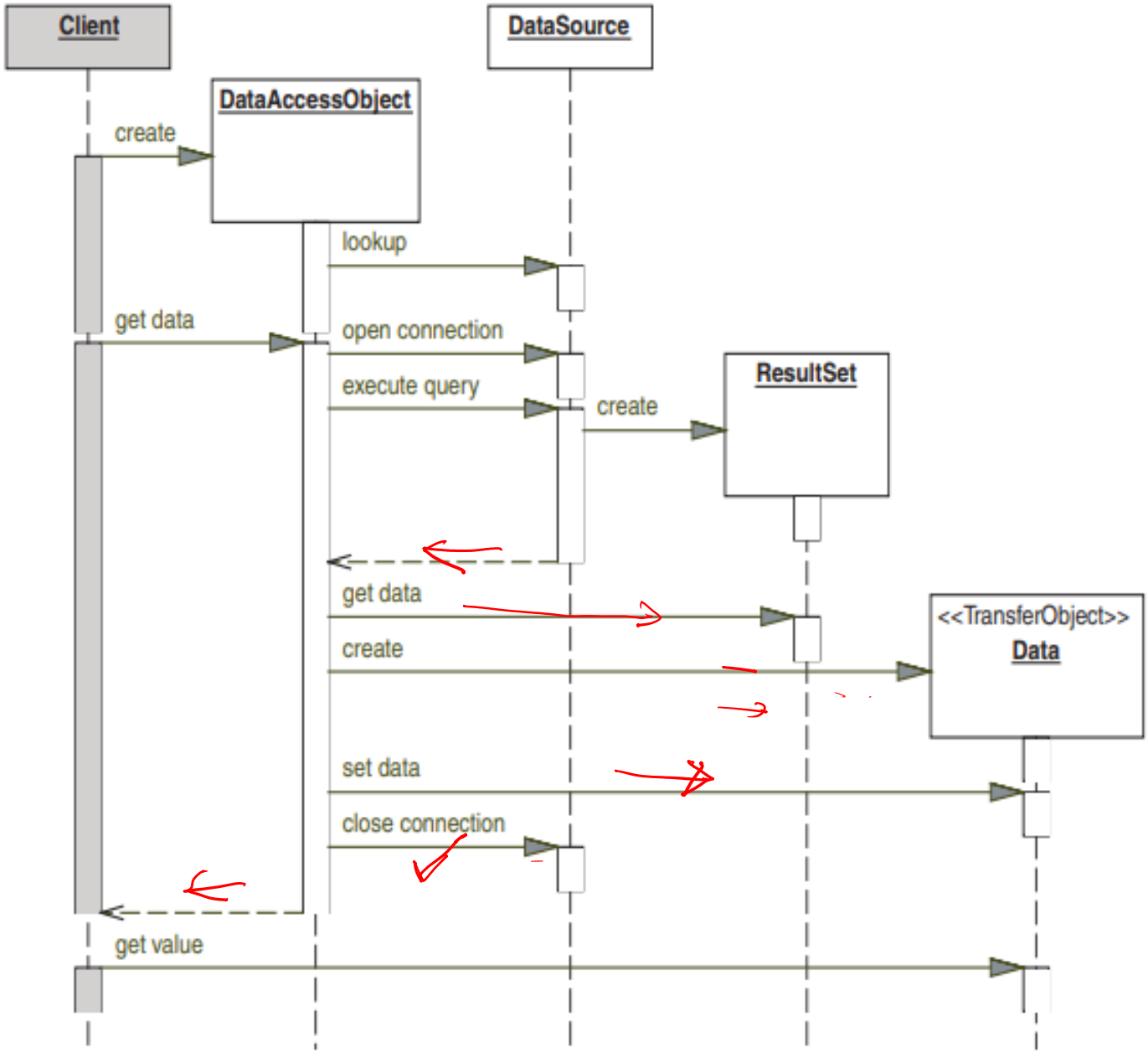
Objects participating in DAO strategy

- **Client**:
  - typically here is a business logic component
- **Data Access Object (DAO)**:
  - This is the main object of the strategy; “primary role object” of this pattern
  - Client makes all request to this object
  - Typically it provides operations for
    - Saving (Add and Update) a data object into database.
    - Reading data objects from database
    - Deleting objects from the database

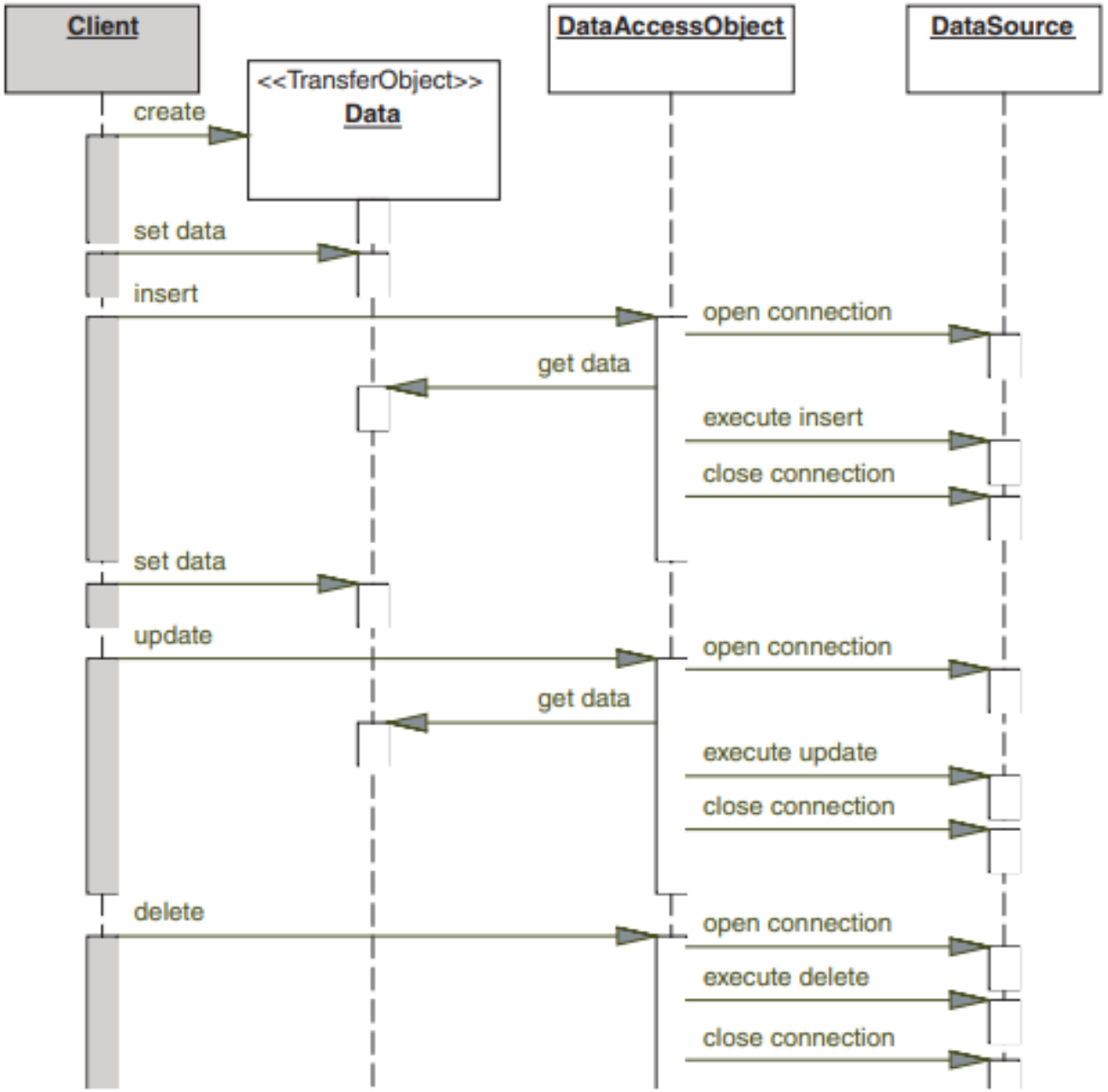
Objects participating in DAO strategy

- Transfer Object: represent another strategy (pattern – let us bit later)
- Data Source:
  - Encapsulated within the DAO and not seen by business clients
  - Basically a delegate for a DBMS or any other data repository (No SQL system, XML, Flat Files)
  - It can also represent another system (legacy/mainframe), service (B2B service or credit card bureau), or some kind of repository (LDAP)
- Result Set:
  - Again private to DAO; often a entity-set returned as result of a select query

DAO – Sequence Diagram



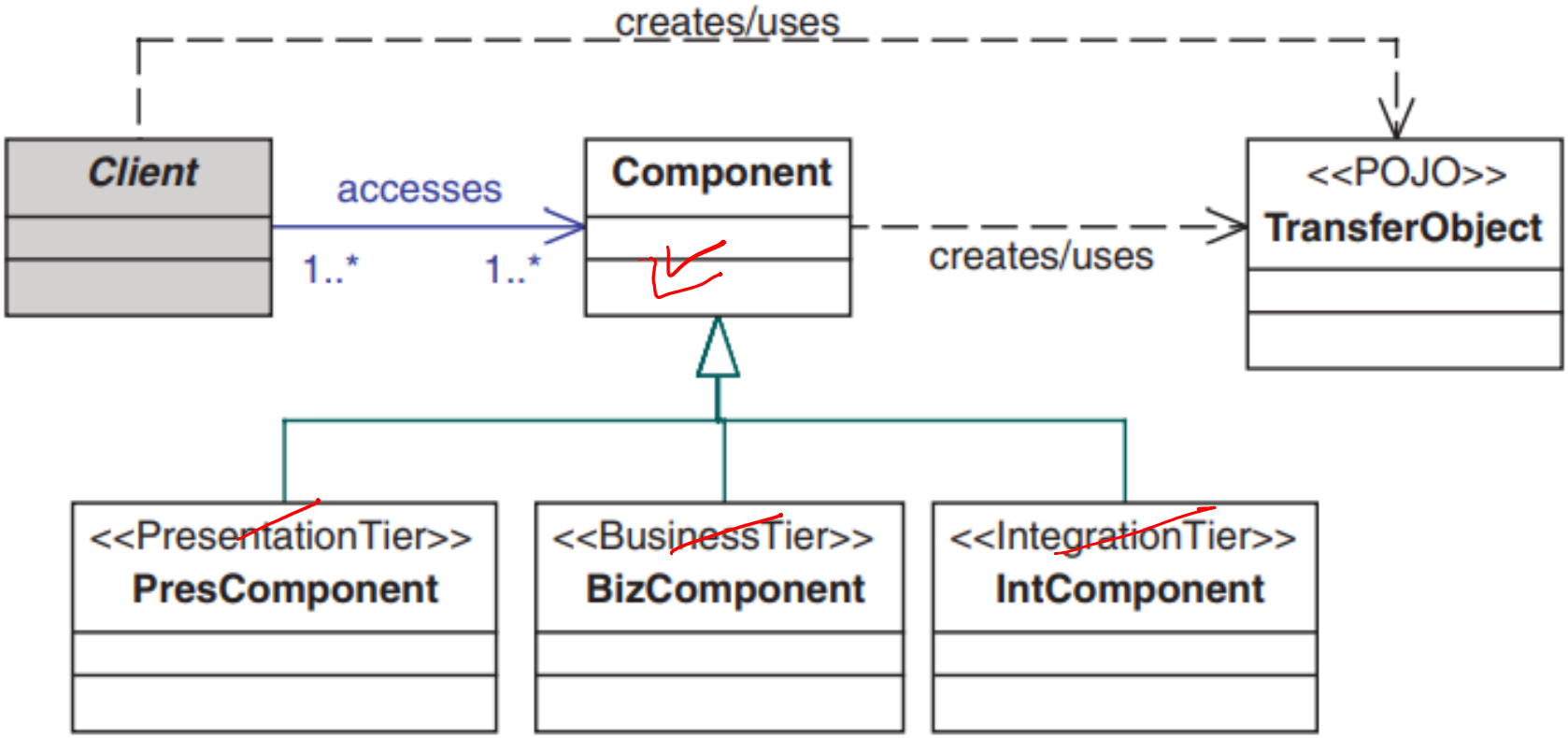
DAO – Sequence Diagram



“Transfer Object” Strategy

- DAO objects may keep data objects which it can not expose; if exposes, it looses its independence of providing “technology independent persistence”?
- because those objects may have dependencies on

Transfer Object (TO) – Class Diagram



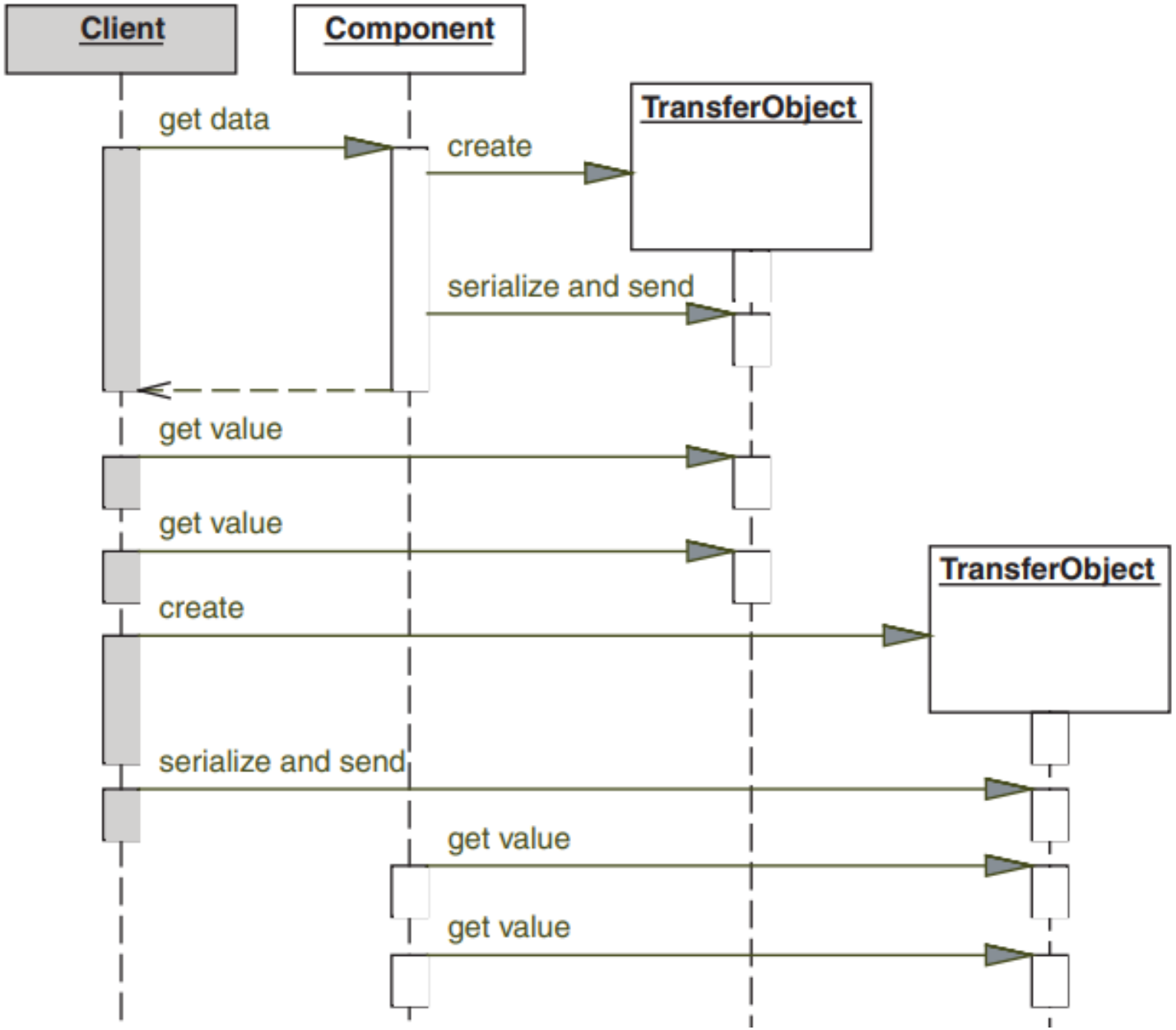
Objects participating in TO strategy

- Client
  - The Client needs to access a Component to send and receive data. Typically, the Client is a Component in another tier.
  - For example, a Component in the presentation tier can act as a client of some business-tier components.
- Component
  - The Component can be any component in another tier that the client accesses to send and receive data.
  - The Component can be in the presentation tier, business tier or integration tier.

Objects participating in TO strategy

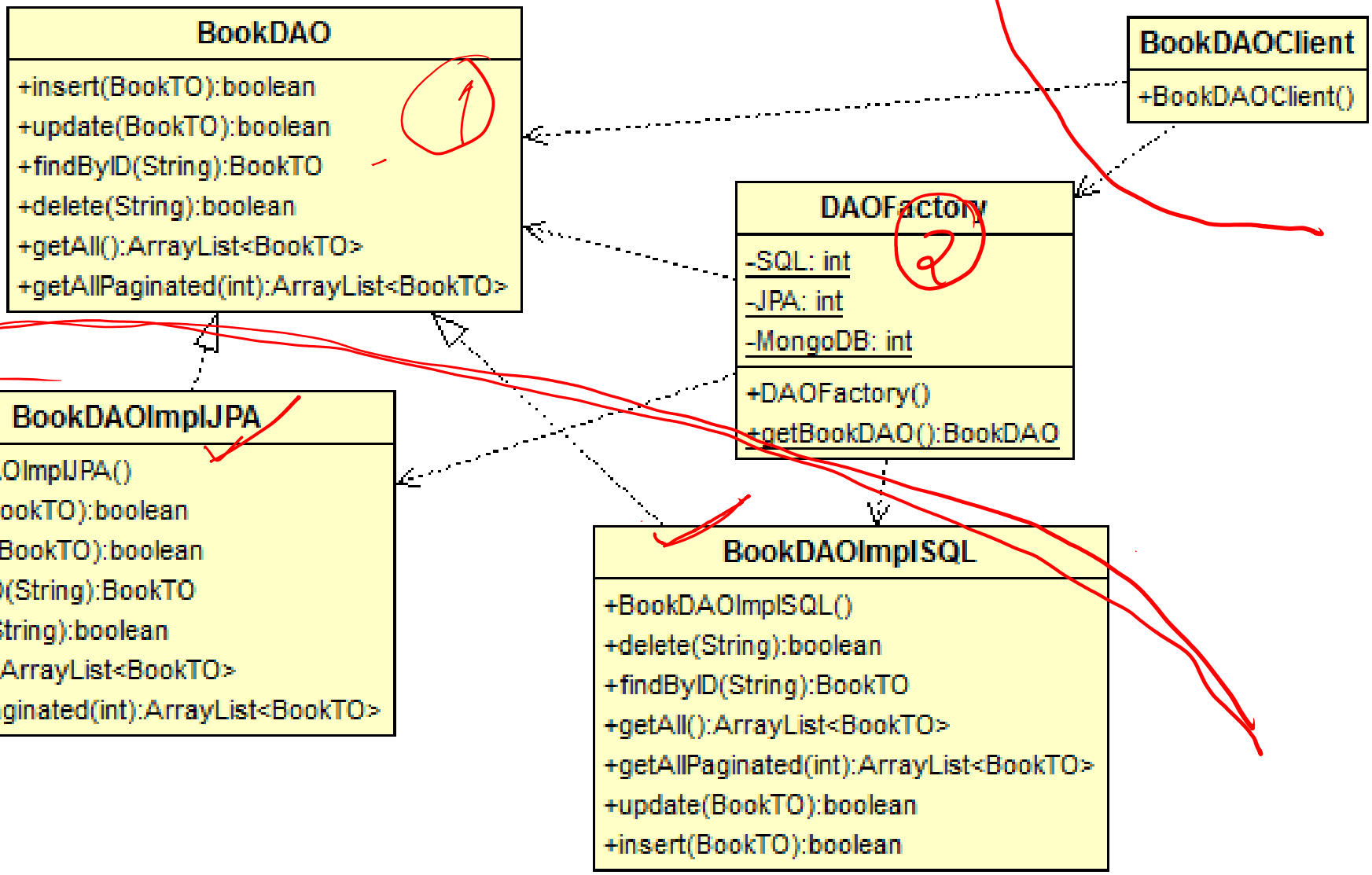
- **PresComponent**: Helper Object (HO), Business Delegate (BD), a Command Object (CO), and so on.
- **BizComponent** is a component in the business tier, such as a Business Object (BO), Application Service, Service Facade, and so on.
- **IntComponent**: is a component in the integration tier, such as a Data Access Object
- **TransferObject**: Transfer Object is a serializable plain old Java object that contains several members to aggregate and carry all the data in a single method call.

TO – Sequence Diagram



BookBO

Case Study – Book DAO



Case Study – Book DAO

```
public static void main(String[] args) throws Exception {

    System.out.println("DAO Tester runs ...");

    try {

        BookDAO book_db = DAOFactory.getBookDAO();

        BookTO b1 = book_db.findByID("1001");
        System.out.println( b1.getIsbn() + "," + b1.getTitle());

        Object x = book_db.getAll();
        ArrayList<BookTO> stock = (ArrayList<BookTO>) x;
        for(BookTO b : stock) {
            System.out.println( b.getIsbn() + ", " + b.getTitle());
        }

    }
    catch(DAOException e) {
        System.out.println("Error: " + e.getMessage() );
        throw e;
    }
    catch(Exception e) {
```

[BookDAOClient.java]

ISBN

SELECT * FROM Books

Observations

- Client sees no smell of SQL or anything underlying database technology
- Should be encapsulated in “Data Access Objects”
- No return of objects like Database Result Sets, SQL Exceptions, etc
  - Return objects like Array List
  - Throw exceptions like **`DAOException`** or so
- Client does not depend on any of concrete Database technology
  - Following Advise of “Dependency Inversion”
  - “Abstract Factory” is a standard strategy that is used for hiding all concrete classes from clients

“Abstract Factory” strategy!

Abstract Factory as strategy to implement “Dependency Inversion”; and hide concrete dependencies from clients!

```
public static void main(String[] args) throws Exception {

    System.out.println("DAO Tester runs ...");

    try {

        BookDAO book_db = new BookDAOImplSQL();

        BookTO b1 = book_db.findByID("1001");
        System.out.println( b1.getIsbn() + "," + b1.getTitle());

        Object x = book_db.getAll();
        ArrayList<BookTO> stock = (ArrayList<BookTO>) x;
        for(BookTO b : stock) {
            System.out.println( b.getIsbn() + "," + b.getTitle());
        }
```

DAO Factory – implementation insight

```
public static BookDAO getBookDAO() throws DAOException {

    BookDAO dao = null;
    //read persistance type from config file, and assign
    //Let us say, it comes JPA
    int persistanceType = DAOFactory.SQL;
    try {
        switch (persistanceType) {
            case DAOFactory.SQL:
                dao = new BookDAOImplSQL();
                break;
            case DAOFactory.JPA:
                dao = new BookDAOImplJPA();
                break;
            case DAOFactory.MongoDB:
                //dao = new BookDAOImplMongoDB();
        }
    }
    catch(SQLException e ) {
        throw new DAOException("Data Access Error ");
    }
    catch(Exception e) {
        throw new DAOException( "Data Access Error" );
    }
    return dao;
```

DAO Implementation Strategy

- Interface BookDAO
- Concrete Implementation (Whatever we want)
  - SQL
  - JPA
  - Mongo, or
  - Whatever

Interface Book DAO

- Interface here is common, still indicative; can have anything that is required for specific business BO

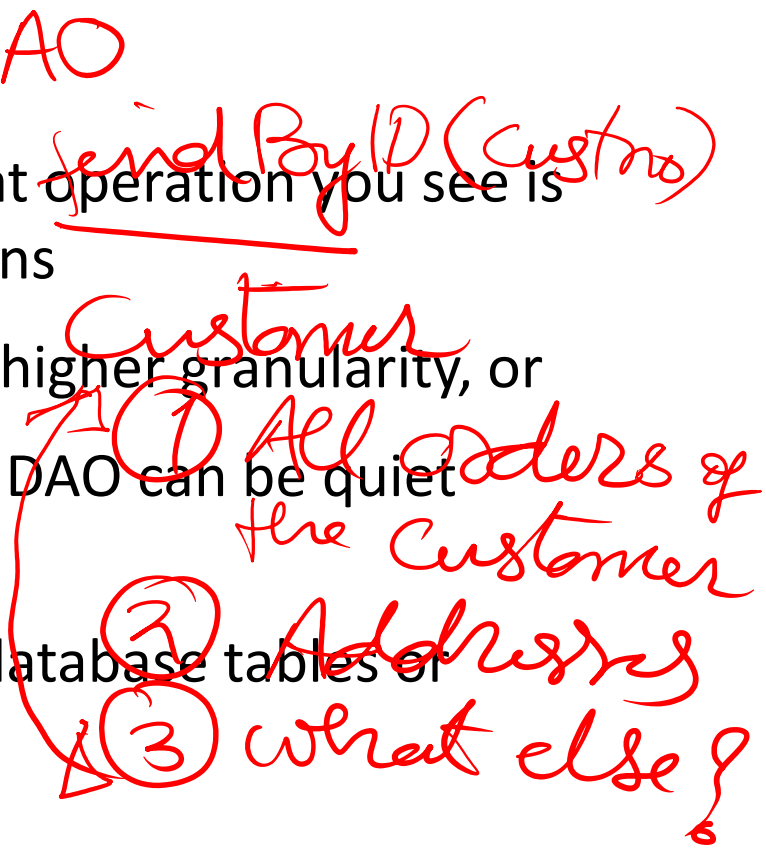
```
public interface BookDAO {

    public void insert(BookTO bk)
            throws DAOException;
    public void update(BookTO bk)
            throws BookNotFound, DAOException;
    public BookTO findByID(String id)
            throws BookNotFound, DAOException;
    public void delete(String isbn)
            throws BookNotFound, DAOException;
    public ArrayList<BookTO> getAll()
            throws DAOException;
    public ArrayList<BookTO> getAllPaginated(int page)
            throws DAOException;

}
```

Interface DAO

- Note that in previous interface, what operation you see is basically performing CRUD operations
- However it can be any operation of higher granularity, or
- Business Objects that are passed to DAO can be quiet complex; for instance Order, and
- Saving Order may update multiple database tables or collections
  - Update Order Table
  - Update Order Details Table
  - Update Customer Account, and Item tables, etc



A typical concrete DAO - SQL

```
public class BookDAOImplSQL implements BookDAO {

    @Override
    public BookTO findByID(String isbn) throws BookNotFound, DAOException {
        BookTO bk = null;
        try {
            Connection con = DBConnection.getInstance().getConnection();
            Statement stmt = con.createStatement();
            String strSQL = "SELECT * FROM books WHERE isbn ='" + isbn + "'";
            ResultSet rs = stmt.executeQuery( strSQL );
            if ( rs.next() ) {
                bk = new BookTO();
                bk.setIsbn( isbn );
                bk.setTitle(rs.getString("title"));
                bk.setPrice(rs.getDouble("price"));
                bk.setStock(rs.getInt("stock"));
            }
        }
        catch(SQLException e) {
            throw new DAOException(e.getMessage());
        }
        return bk;
    }
```

A typical concrete DAO - SQL

```
@Override
public ArrayList<BookTO> getAllPaginated(int page) throws DAOException {
    ArrayList<BookTO> book_list = new ArrayList<>();
    BookTO bk;
    Statement stmt;
    try {
        Connection con = DBConnection.getInstance().getConnection();
        stmt = con.createStatement();
        //use offset and limit to return required page
        int offset = (page-1)*page_size;
        String sql = "SELECT * FROM BOOKS limit " + page_size + " offset " + offset;
        ResultSet rs = stmt.executeQuery( sql );
        while ( rs.next() ) {
            bk = new BookTO();
            bk.setIsbn( rs.getString("isbn") );
            bk.setTitle(rs.getString("title"));
            bk.setPrice(rs.getDouble("price"));
            bk.setStock(rs.getInt("stock"));
            book_list.add( bk );
        }
    }
    catch(SQLException e) {
        throw new DAOException(e.getMessage());
    }
```

(Re) Sources

[1] Primarily sourced from chapter 8 of book:
Core J2EE Patterns: Best Practices and Design Strategies, Deepak Alur, Dan Malks, John Crupi, Prentice Hall