



# Construção de Sistemas de Software

*Putting layers together*

+

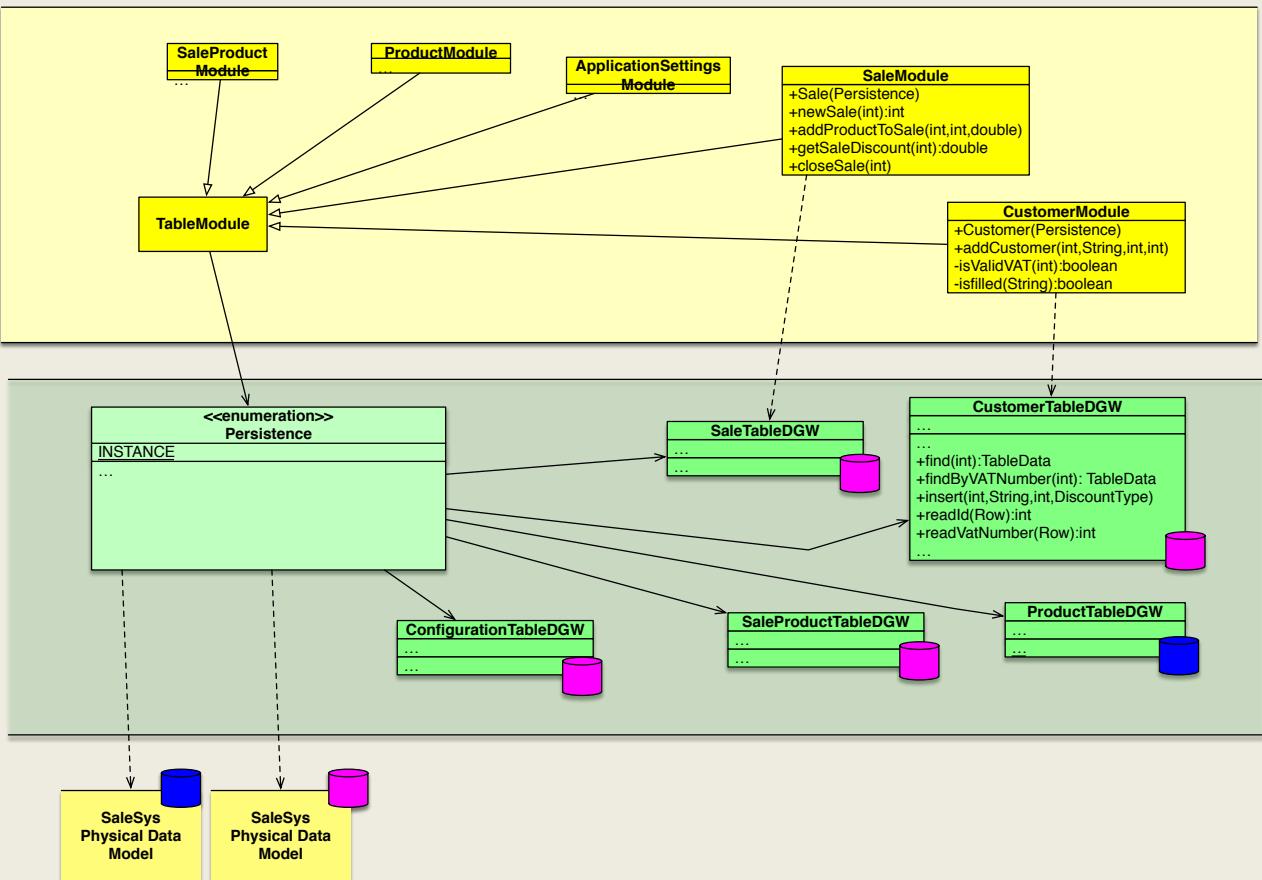
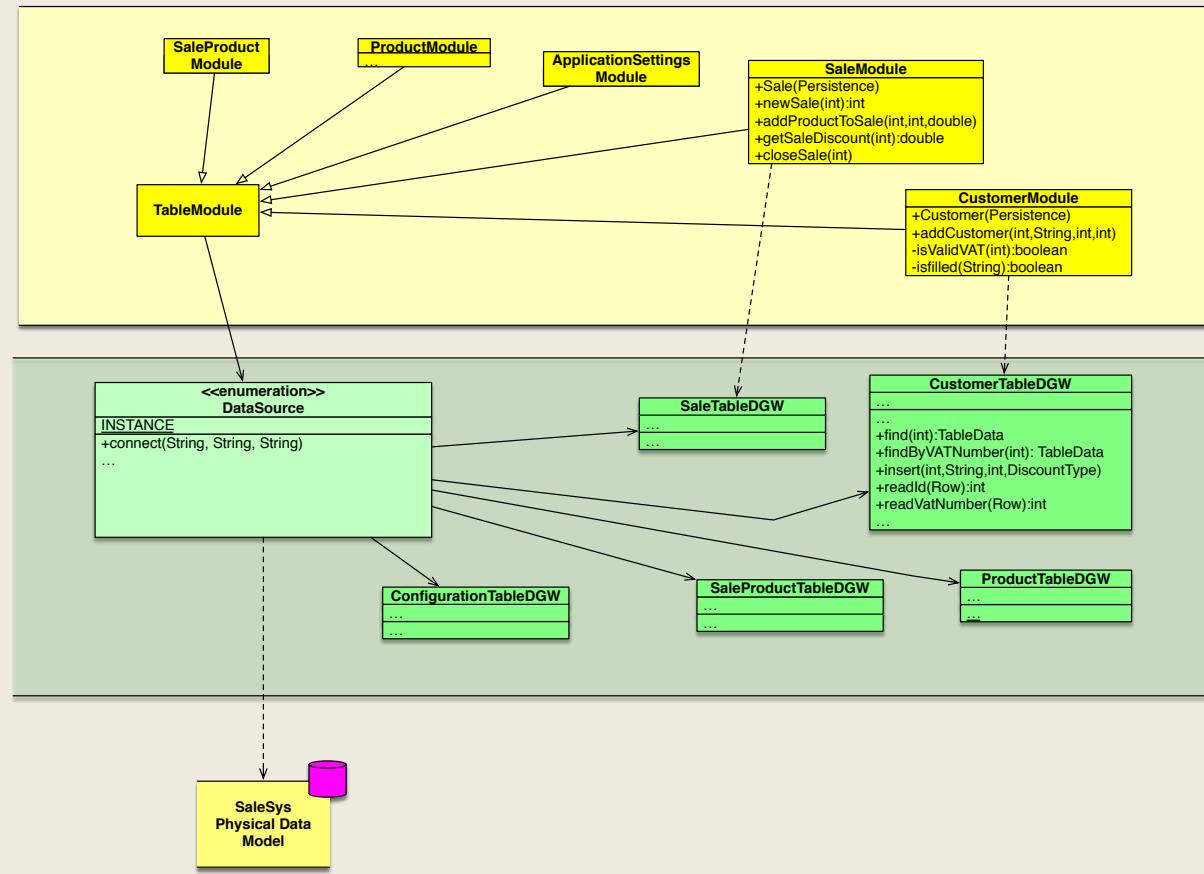
*Active Record & Data Mapper*

## Exercícios (Folha 3)

Suponha que se pretende desenhar a camada de acesso aos dados do SalesSys recorrendo ao padrão *Table Data Gateway* e que essa camada vai ser responsável por realizar a comunicação da aplicação com o SGBD escolhido — Derby — através de um conector JDBC. Apesar de existir atualmente uma única fonte de dados, o desenho da camada deve prever a possibilidade de serem usadas várias fontes de dados. Adicionalmente o desenho da camada deve ter em conta que se pretende usar esta camada por baixo dos *table modules* concebidos anteriormente (folha 2).

- (c) Desenhe uma classe *Persistence* cujos objetos abstraem formas de realizar a persistência e as respetivas fontes dos dados, e que são responsáveis por criar os *gateways* para as tabelas e suporte a execução de transações. Sugestão: Adapte a classe *DataSource* definida no exercício 1 de forma a poder ter várias instâncias.





## Acesso aos Dados com Várias Fontes de Dados

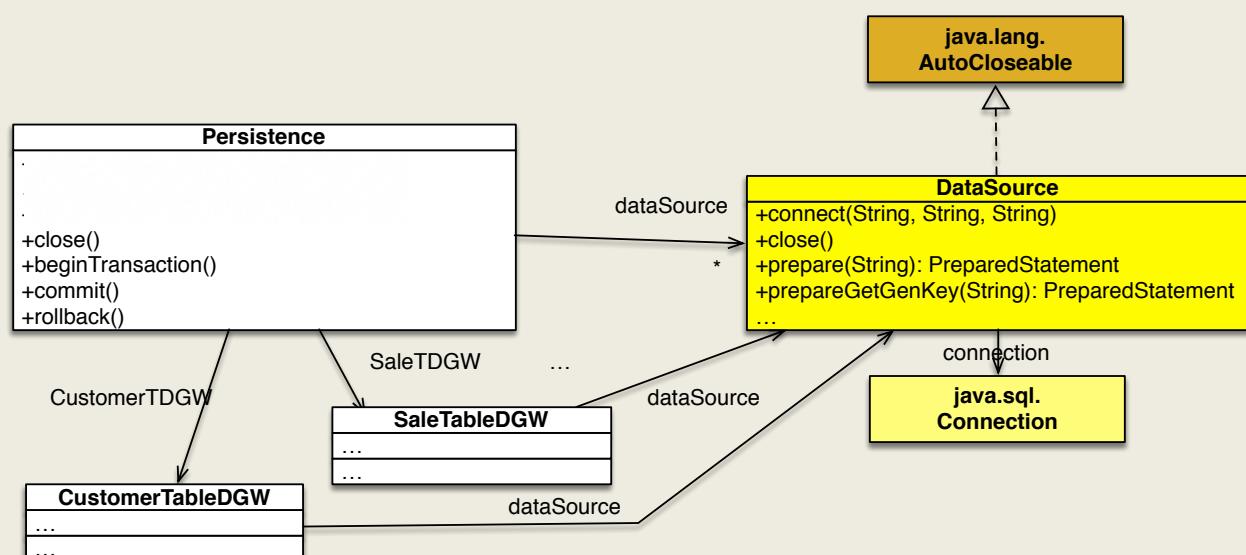
Classe **Persistence** cujos objetos

- abstraem diferentes formas de realizar a persistência e as respetivas fontes dos dados
- são responsáveis por criar os *gateways* para as tabelas e suportar a execução de transações

É necessário adaptar a classe **DataSource**

- deixa de ter uma única instância
- as suas instâncias abstraem uma ligação à base de dados através de um conector JDBC e que são responsáveis por todas as ações sobre essa ligação

## Várias Fontes de Dados



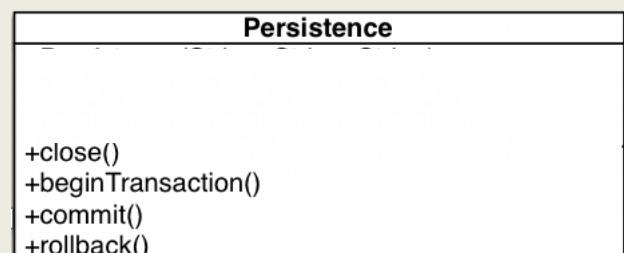
## Ligaçāo à Persistence nos Table Modules

```
^/
public class ProductModule extends TableModule {

    private ProductTableDataGateway table;

    * Constructs a product module given the persistence repository...
    public ProductModule(Persistence persistence) {
        super(persistence);
        table = persistence.productTableGateway;
    }
}
```

## Fontes de Dados e JDBC



- Os métodos que permitem começar uma transação, fazer commit e rollback são programados à custa de chamadas a

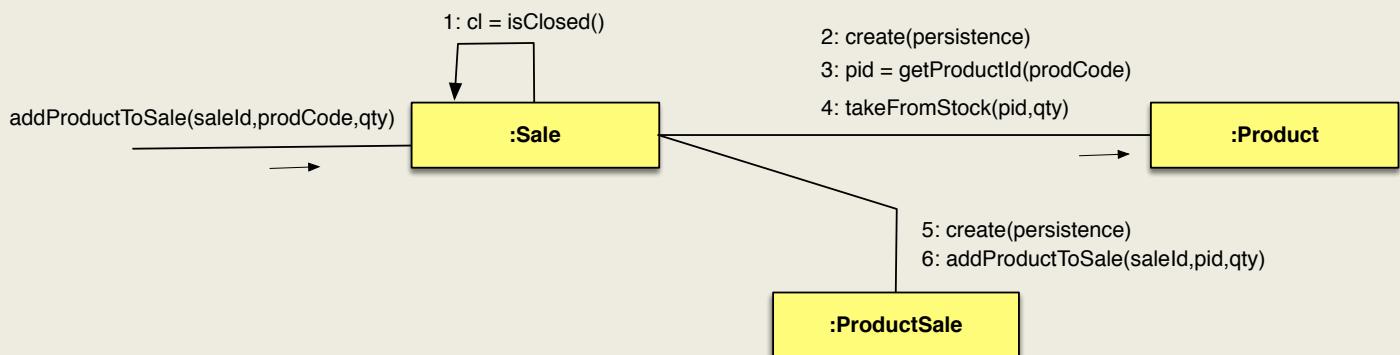
`connection.setAutoCommit(false)`

`connection.commit(); connection.setAutoCommit(true)`

`connection.rollback(); connection.setAutoCommit(true)`



## AddProductToSale: Solução com Table Module + TDGW



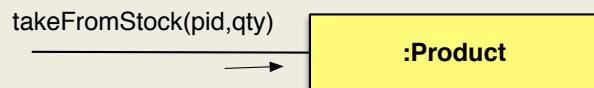
Ciências  
ULisboa

## AddProductToSale: Transações

```
SaleModule.java
14  */
75@  public void addProductToSale(int saleId, int productCode, double qty)
76      throws ApplicationException {
77      // Business rule: products can be only added to open sales
78      if (isClosed (saleId))
79          throw new ApplicationException("Cannot add products to a closed sale.");
80      // take units from the stock
81      int productId = 0;
82      try {
83          persistence.beginTransaction();
84          ProductModule product = new ProductModule(persistence);
85          productId = product.getProductId(productCode);
86          product.takeFromStock (productId, qty);
87          // add the product to the sale
88          new SaleProductModule(persistence).addProductToSale(saleId, productId, qty);
89          persistence.commit();
90      } catch (PersistenceException e) {
91          try {
92              persistence.rollback();
93          } catch (PersistenceException e1) {
94              throw new ApplicationException(INTERNAL_ERROR_WITH_SELLING_PRODUCT_ID + productId, e1);
95          }
96          throw new ApplicationException(INTERNAL_ERROR_WITH_SELLING_PRODUCT_ID + productId, e);
97      }
98      catch (ApplicationException e) {
99          try {
100              persistence.rollback();
101          } catch (PersistenceException e1) {
102              throw new ApplicationException(INTERNAL_ERROR_WITH_SELLING_PRODUCT_ID + productId, e1);
103          }
104          throw e;
105      }
106 }
```

## ***takeFromStock***

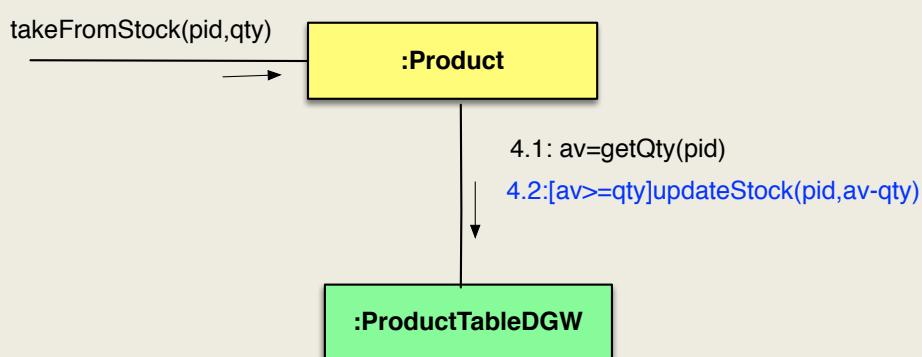
---



- Verificar que há quantidade suficiente do produto em stock
- Alterar a quantidade do produto em stock

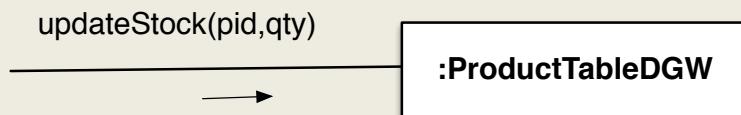
## ***takeFromStock***

---



O que acontece se a aplicação estiver a servir pedidos concorrentes?

## Classe *ProductTableDGW*: *updateStock(productId,qty)*

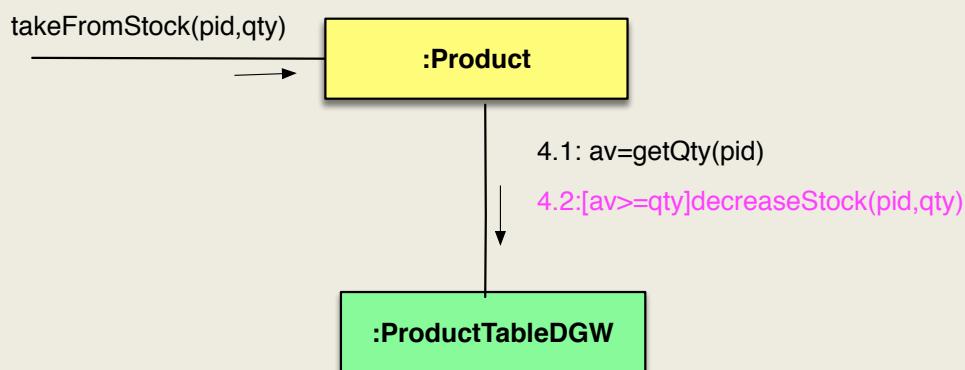


- Definir constante com *string* SQL que faz *update*

```
private static final String UPDATE_STOCK_SQL =
    "update " + TABLE_NAME + " "
    "set " + QTY_COLUMN_NAME + " = ? "
    "where " + ID_COLUMN_NAME + " = ?";
```

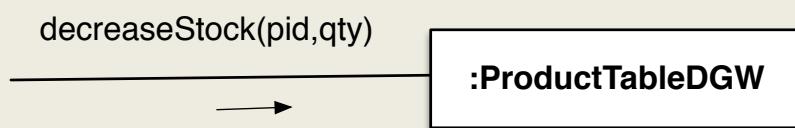
- Pedir à respetiva *DataSource* para criar *PreparedStatement* com esta *string*
- Preencher o 1º ? com *qty*
- Preencher o 2º ? com *pid*
- Não funciona se houver pedidos concorrentes!

## *takeFromStock*



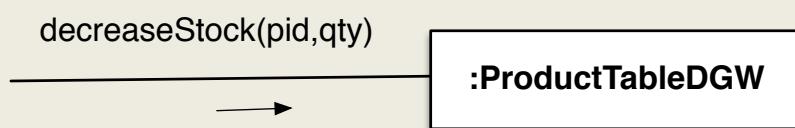
## Classe *ProductTableDGW*: *decreaseStock(productId,qty)*

---



## Classe *ProductTableDGW*: *decreaseStock(productId,qty)*

---



- A leitura do valor corrente e a alteração do valor tem de ser feita de forma atómica. Alternativas

```
private static final String UPDATE_STOCK_SQL =
    "update " + TABLE_NAME + " " +
    "set " + QTY_COLUMN_NAME + " = " + QTY_COLUMN_NAME + "- ? " +
    "where " + ID_COLUMN_NAME + " = ?";
```

ou

```
private static final String SELECTANDUPDATE_STOCK_SQL =
    "select * " +
    "from " + TABLE_NAME + " " +
    "where " + ID_COLUMN_NAME + " = ? " +
    "for update";
```

## Classe *ProductTableDGW*: *decreaseStock(productId,qty)*

ProductTableDataGateway.java

```
59     */
60     private static final String SELECTANDUPDATE_STOCK_SQL =
61         "select * " +
62             "from " + TABLE_NAME + " " +
63             "where " + ID_COLUMN_NAME + " = ? " +
64             "for update";
```

DataSource.java

```
88     }
89
90     public PreparedStatement prepareUpdate(String sql) throws PersistenceException {
91         try {
92             return connection.prepareStatement(sql,ResultSet.TYPE_FORWARD_ONLY,
93                 ResultSet.CONCUR_UPDATABLE, ResultSet.CLOSE_CURSORS_AT_COMMIT);
94         } catch (SQLException e) {
95             throw new PersistenceException("Error preparing command", e);
96         }
97     }
```

## Alterações concorrentes do stock

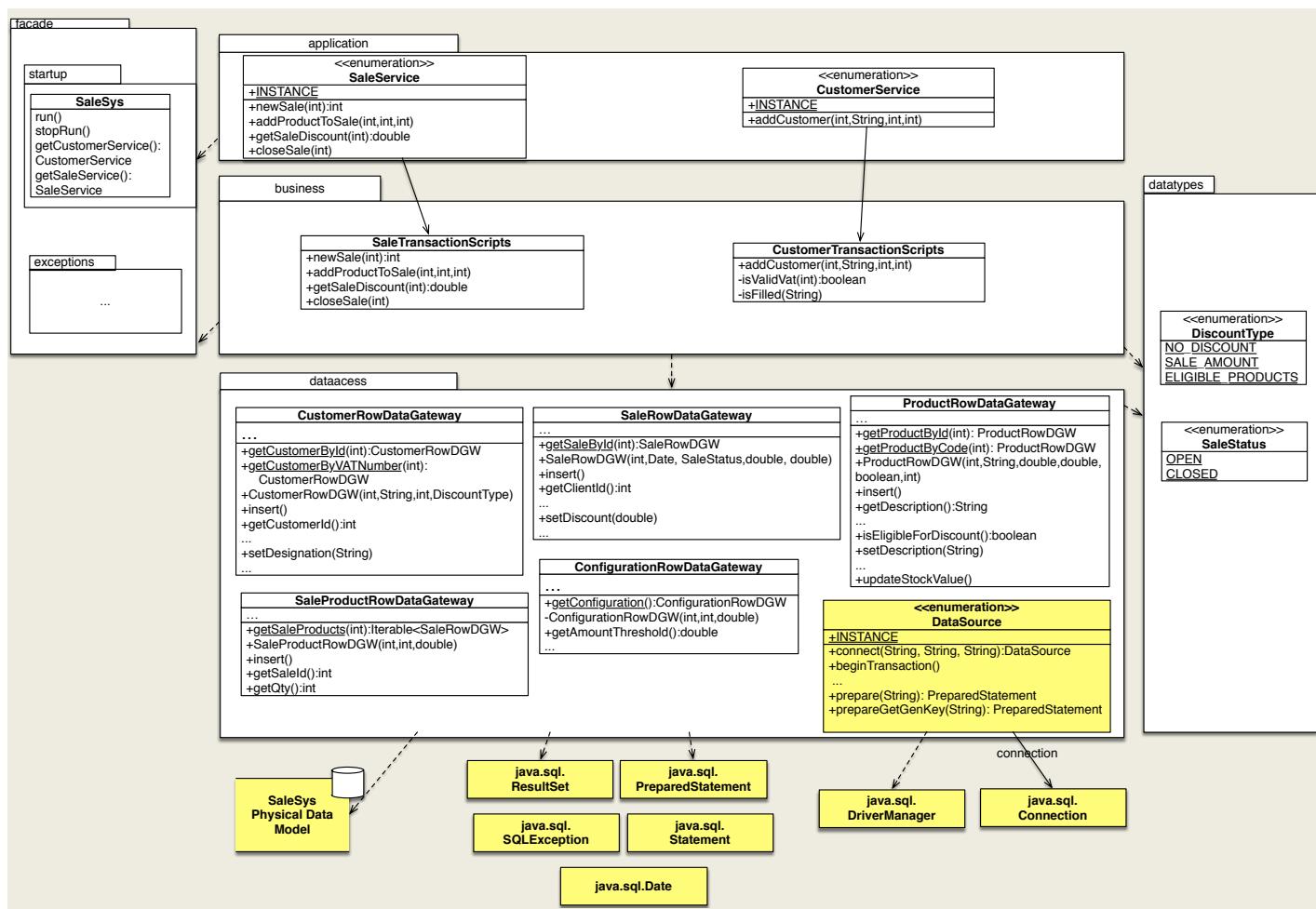
ProductTableDataGateway.java

```
/***
 * Decreases the quantity in stock of a product id by a given qty,
 * if there is enough quantity in stock (atomic check&update)
 */
public void decreaseStock(int productId, double qty) throws PersistenceException {
    try (PreparedStatement statement = dataSource.prepareStatement(SELECTANDUPDATE_STOCK_SQL)) {
        // set statement arguments
        statement.setInt(1, productId);
        // execute SQL
        try (ResultSet rs = statement.executeQuery()) {
            if(rs.next()) {
                double current = rs.getDouble(QTY_COLUMN_NAME);
                //another alternative would be to rely on the schema constraint over stock values
                if (current - qty >= 0) {
                    //since the resultset was defined as updatable
                    rs.updateDouble(QTY_COLUMN_NAME, current-qty);
                    rs.updateRow();
                }
                else
                    throw new PersistenceException(
                        "Product with internal id " + productId + " has (" + current +
                        ") units, not enough for taking " + qty + " units.");
            }
            else
                throw new PersistenceException("Internal error updating product stock");
        }
    } catch (SQLException e) {
        throw new PersistenceException("Internal error updating product stock. ", e);
    }
}
```

## Exercício

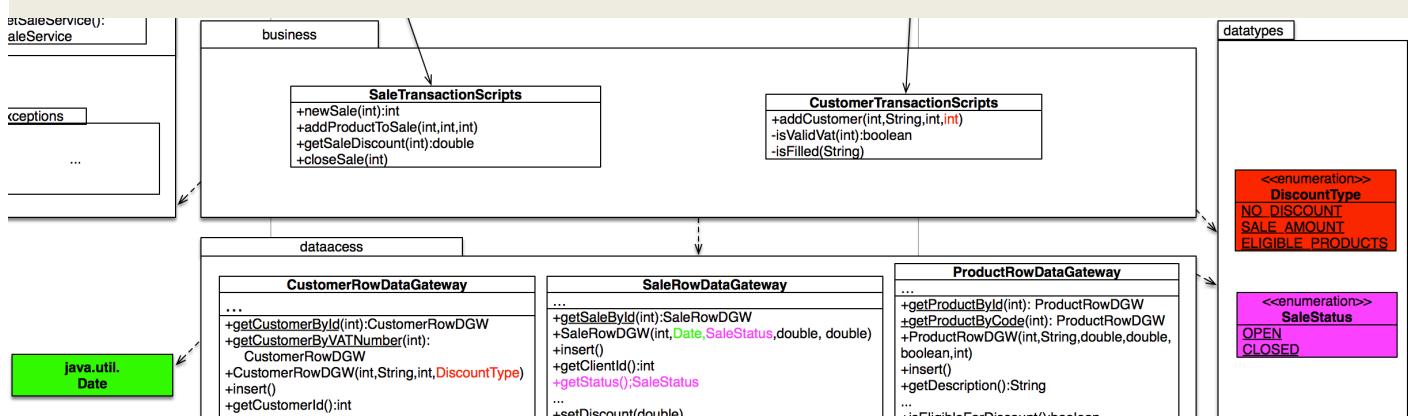
Considere a solução de desenho do SaleSys fornecida no segundo trabalho prático, a qual resulta de combinar a camada de negócios desenhada de acordo com o *Transaction Script* com a camada de acesso aos dados desenhada de acordo com o *Row Data Gateway*. Recorde que o sistema resultante é constituído por um componente SaleSys App que comunica através de um conector JDBC com uma base de dados Derby SaleSys DB.

- (a) Analise o desenho das várias operações do sistema focado desta vez nos casos desfavoráveis (por exemplo, o caso em que o vat fornecido no newSale não pertence a nenhum cliente no sistema, o idSale fornecido no addProductToSale não pertence a nenhuma venda ou o prodCode não é o código de nenhum produto). Reveja as exceções que devem ser levantadas pelo código de cada uma das camadas e a forma como deve ser feita a sua propagação.



## Conversão de Representações e Tipos de Dados

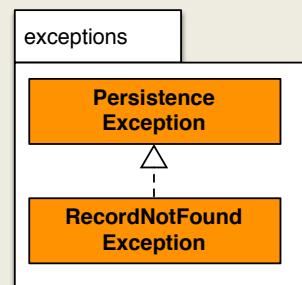
- Tipo dos descontos
  - Possivelmente três representações diferentes (uma em cada camada)
- *Sale status*
  - Duas representações diferentes
- Datas
  - `java.util.Date`
  - `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`



## Casos Desfavoráveis

- Anteriormente ignorámos os casos desfavoráveis, por exemplo:
  - o vat fornecido no `newSale` não pertence a nenhum cliente no sistema
  - o `idSale` fornecido no `addProductToSale` não identifica nenhuma venda ou o `prodCode` não é o código de nenhum produto
- É na **camada de acesso aos dados** que devem ser
  - definidas exceções apropriadas para sinalizar os problemas que possam ocorrer no acesso aos dados (devem ser independentes do JDBC)
  - e converter para o tipo adequado as exceções lançadas pelo JDBC

`java.sql.SQLException`



## Casos Desfavoráveis

### dataaccess.exceptions

- ▶ PersistenceException.java
- ▶ RecordNotFoundException.java

```
public void connect (String server, String database, String userName, String password)
    throws PersistenceException {
    try {
        connection = DriverManager.getConnection (getSpecificURL(server, database), userName, password);
    } catch (SQLException e) {
        throw new PersistenceException("Cannot connect to database", e);
    }
}

public void insert () throws PersistenceException {
    try (PreparedStatement statement = DataSource.INSTANCE.prepareStatement(INSERT_CUSTOMER_SQL)) {
        statement.setInt(1, vat);
        statement.setString(2, designation);
        statement.setInt(3, phoneNumber);
        statement.setInt(4, discountId);
        statement.executeUpdate();
        try (ResultSet rs = statement.getGeneratedKeys()) {
            rs.next();
            id = rs.getInt(1);
        }
    } catch (SQLException e) {
        throw new PersistenceException ("Internal error!", e);
    }
}
```

## Versão 1: propagação da RecordNotFoundException

```
public ProductRowDataGateway find(int id) throws PersistenceException {
    try (PreparedStatement statement = DataSource.INSTANCE.prepareStatement(GET_PRODUCT_BY_ID_SQL)) {
        // set statement arguments
        statement.setInt(1, id);
        // execute SQL
        try (ResultSet rs = statement.executeQuery()) {
            // creates a new product with the data retrieved from the database
            return loadProduct(rs);
        }
    } catch (SQLException e) {
        throw new PersistenceException("Internal error getting product with id" + id, e);
    }
}

private static SaleRowDataGateway loadSale(ResultSet rs) throws RecordNotFoundException {
    try {
        rs.next();
        SaleRowDataGateway newSale = new SaleRowDataGateway(rs.getInt(CUSTOMER_ID_COLUMN_NAME),
            0, 0, rs.getDate(DATE_COLUMN_NAME), null);
        newSale.id = rs.getInt(ID_COLUMN_NAME);
        newSale.total = rs.getDouble(TOTAL_COLUMN_NAME);
        newSale.discount = rs.getDouble("discount_" + TOTAL_COLUMN_NAME);
        newSale.status = rs.getString(STATUS_COLUMN_NAME);
        return newSale;
    } catch (SQLException e) {
        throw new RecordNotFoundException ("Sale does not exist ", e);
    }
}
```

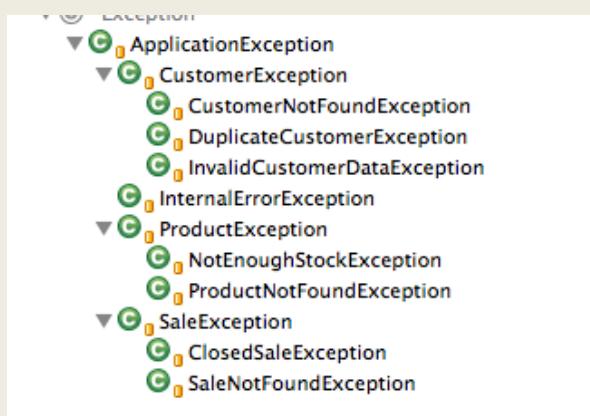
## Versão 2: utilização do Optional<T>

```
public static Optional<CustomerRowDataGateway> findWithVATNumber (int vat) {
    try (PreparedStatement statement = DataSource.INSTANCE.prepareStatement(GET_CUSTOMER_BY_VAT_NUMBER_SQL)) {
        statement.setInt(1, vat);
        try (ResultSet rs = statement.executeQuery()) {
            return Optional.of(loadCustomer(rs));
        }
    } catch (SQLException | PersistenceException e) {
        log.log(Level.SEVERE, "Internal error getting a customer by its VAT number", e);
        return Optional.empty();
    }
}

private static CustomerRowDataGateway loadCustomer(ResultSet rs) throws RecordNotFoundException {
    try {
        rs.next();
        CustomerRowDataGateway newCustomer = new CustomerRowDataGateway(rs.getInt(VAT_NUMBER_COLUMN_NAME),
            rs.getString(DESIGNATION_COLUMN_NAME), rs.getInt(PHONE_NUMBER_COLUMN_NAME),
            toDiscountType(rs.getInt(DISCOUNT_ID_COLUMN_NAME)));
        newCustomer.id = rs.getInt(ID_COLUMN_NAME);
        return newCustomer;
    } catch (SQLException e) {
        throw new RecordNotFoundException ("Customer does not exist", e);
    }
}
```

## Casos Desfavoráveis

- A **camada de negócio** deve
  - definir exceções apropriadas para sinalizar os problemas que possam ocorrer e que fazem sentido ao nível do negócio
  - apanhar todo o tipo de exceção que possa ser lançada pela camada de acesso aos dados e sinalizar o problema com exceção específica
  - porque essas exceções vão ser usadas pela camada acima, são colocadas na **Facade**



## Casos Desfavoráveis

---

```
public void addCustomer(int vat, String denomination, int phoneNumber, int discountCode)
    throws CustomerException {
    // Checks if it is a valid VAT number
    if (!isValidVAT (vat))
        throw new InvalidCustomerDataException ("Invalid VAT number: " + vat);

    // Checks if the discount code is valid.
    if (discountCode < 0 || discountCode >= DiscountType.values().length)
        throw new InvalidCustomerDataException ("Invalid Discount Code: " + discountCode);

    // Checks that denomination and phoneNumber are filled in
    if (!isFilled(denomination) || phoneNumber == 0)
        throw new InvalidCustomerDataException(
            "Both denomination and phoneNumber must be filled");

    // Insert the customer. Duplicates are handled by the database (unique constraint)
    try {
        CustomerRowDataGateway newCustomer
            ...
        newCustomer.insert();
    } catch (PersistenceException e) {
        throw new DuplicateCustomerException("Customer with VAT number " + vat + " already exists",
    }
}
```



## Exercício

---

- (b) Praticamente todas as classes da camada de acesso aos dados lidam com recursos que precisam de ser fechados; por exemplo, objetos do tipo Connection, Statement ou ResultSet. Explique de que forma se pode endereçar este problema recorrendo a instruções **try-with-resources**.



## Fecho de Recursos

---

- Um recurso é um objeto que precisa de ser fechado quando não é mais necessário.
  - Estes tipos devem ser definidos como implementando o interface **java.lang.AutoCloseable**
  - Exemplos: **java.sql.CallableStatement, Connection, PreparedStatement, Statement, ResultSet**
- A instrução **try-with-resources** é uma instrução que serve para embrulhar um bloco de código de forma a que todos os recursos declarados nesse bloco sejam automaticamente fechados no final da execução desse bloco (mesmo que sejam levantadas exceções). Os recursos são fechados na ordem inversa à da sua criação.



## Fecho de Recursos (com **try-with-resources**)

---

```
try (Connection con = ds.getConnection();
     PreparedStatement ps = con.prepareStatement(sql)) {

    try (ResultSet rs = ps.executeQuery();) {
        while (rs.next()) {
            list.add(rs.getInt("id"));
        }
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```



## Fecho de Recursos (antes do Java 7....)

```
Connection con =null;
PreparedStatement ps = null;
ResultSet rs = null;
try {
    con = ds.getConnection();
    ps = con.prepareStatement(sql);

    rs = ps.executeQuery();
    while (rs.next()) {
        list.add(rs.getInt("id"));
    }
} catch (SQLException e) {
    e.printStackTrace();
}
finally {
    if (rs != null) {
        try {
            rs.close();
        } catch (SQLException e) { }
    }
    if (ps != null) {
        try {
            ps.close();
        } catch (SQLException e) { }
    }
    if (conn != null) {
        try {
            conn.close();
        } catch (SQLException e) { }
    }
}
```



## Fecho de Recursos no SaleSys

```
public SaleRowDataGateway find (int id) throws PersistenceException {
    try (PreparedStatement statement = dataSource.prepareStatement(...)) {
        // set statement arguments
        statement.setInt(1, id);
        // executes SQL
        try (ResultSet rs = statement.executeQuery()) {
            // creates a new sale with the data retrieved from the database
            return rowGatewayFactory.newSale(rs);
        }
    } catch (SQLException e) {
        throw new PersistenceException("Internal error getting a sale", e);
    }
}
```



## Exercício

---

- (c) A criação de objetos *Row Data Gateway*, em vez de ser feita através de métodos de classe na própria classe, pode ser feita através da utilização de um objeto *Finder* específico a cada tabela da base de dados. Discuta as vantagens das duas alternativas.
- (d) Faça um *refactoring* da camada de acesso aos dados de forma ter a criação de objetos *Row Data Gateway* a ser feita por objetos *Finder*.



## Criação de *RDGWs* nas pesquisas: métodos de classe vs *Finders*

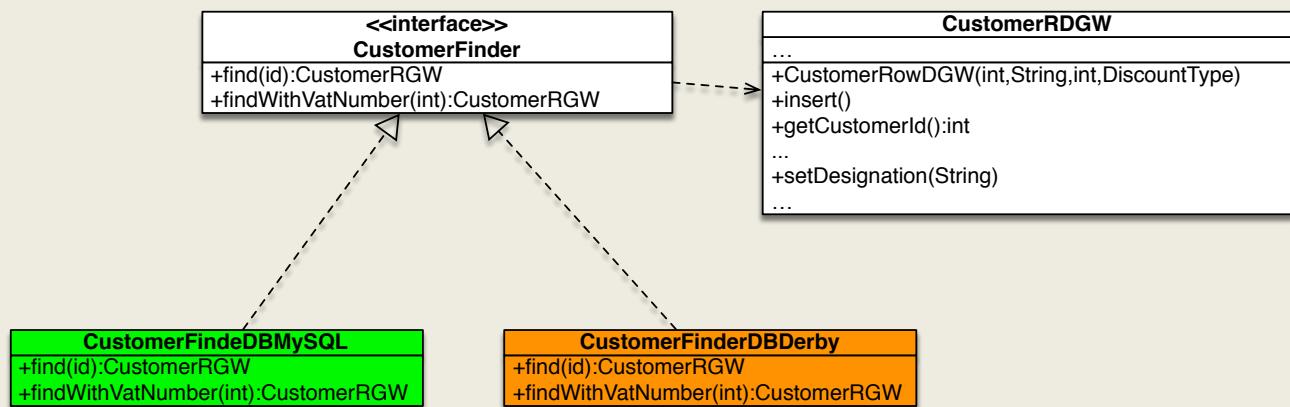
---

- Com métodos de classe
  - mais simples de implementar
  - não permitem ter polimorfismo

<b>CustomerRowDataGateway</b>
...
+getCustomerById(int):CustomerRowDGW
+getCustomerByVatNumber(int):CustomerRowDGW
+CustomerRowDGW(int, String, int, DiscountType)
+insert()
+getCustomerId():int
...
+setDesignation(String)
...

## Criação de RDGWs nas pesquisas: métodos de classe vs *Finders*

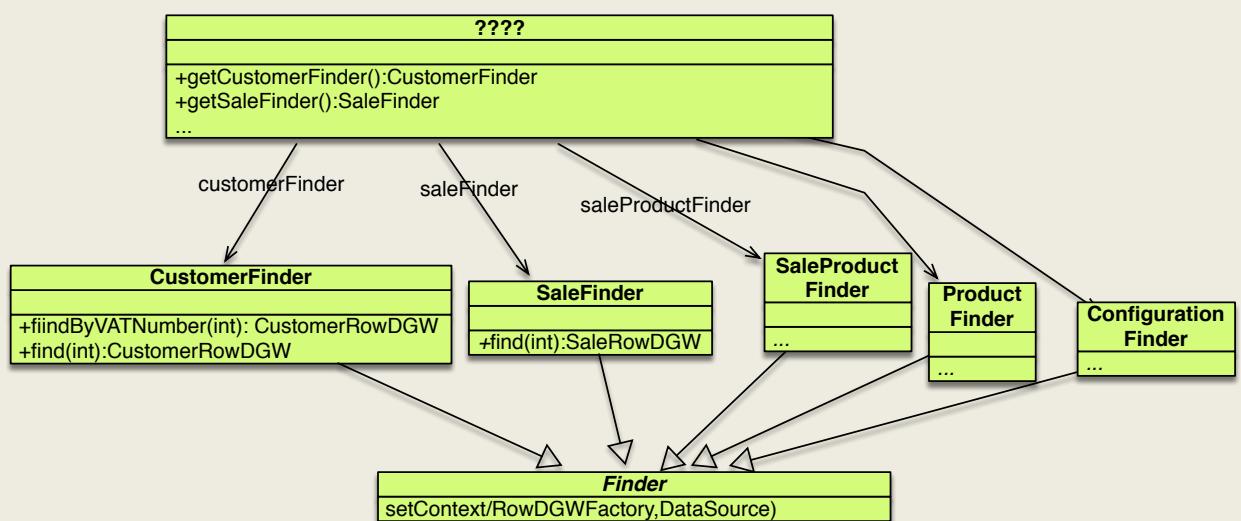
- A definição de interfaces *Finders* protege-nos mais relativamente a variações no SGBD
- O resto das classes podem ficar dependentes apenas do interface, e não das classes concretas que os implementam



## Modificabilidade relativamente à fonte de dados

### • Criação dos *Finders*

- Deixar a criação dos *Finders* concretos aos *Transaction Scripts* vai espalhar a decisão de usar uma fonte de dados/SGBD específica por várias classes
- Para localizar esta mudança, recorre-se antes a uma classe a quem se dá a responsabilidade de criar e conhecer os *Finders*



## Exercício

- (e) Suponha que é decidido passar a usar o *MySQL* em vez do *Derby*. Identifique as modificações que teria de fazer no código.
- (f) Faça um *refactoring* da camada de acesso aos dados de forma a assegurar que qualquer futura mudança no SGBD vai poder ser realizada sem esforço. Prepare em particular uma solução que permita facilmente decidir se queremos arrancar com uma versão do sistema que usa o *MySQL* ou o *Derby*.



Ciências  
ULisboa

| Informática

## Derby vs MySQL

```
INSERT_INTO + CustomerRowDataGateway.TABLE_NAME + " " +
"(" + CustomerRowDataGateway.ID_COLUMN_NAME + ", " +
CustomerRowDataGateway.VAT_NUMBER_COLUMN_NAME + ", " +
CustomerRowDataGateway.DESIGNATION_COLUMN_NAME + ", " +
CustomerRowDataGateway.PHONE_NUMBER_COLUMN_NAME + ", " +
CustomerRowDataGateway.DISCOUNT_ID_COLUMN_NAME +
") " +
VALUES + "(DEFAULT, ?, ?, ?, ?);
```

```
INSERT_INTO + CustomerRowDataGateway.TABLE_NAME + " " +
"(" + CustomerRowDataGateway.VAT_NUMBER_COLUMN_NAME + ", " +
CustomerRowDataGateway.DESIGNATION_COLUMN_NAME + ", " +
CustomerRowDataGateway.PHONE_NUMBER_COLUMN_NAME + ", " +
CustomerRowDataGateway.DISCOUNT_ID_COLUMN_NAME +
") " +
VALUES + "(?, ?, ?, ?);
```

```
public void createCSSDerbyDB() throws IOException, SQLException, PersistenceException {
    DataSource.INSTANCE.connect(SaleSys.DB_CONNECTION_STRING + ";create=true", "SaleSys", "");
    RunSQLScript.runScript(DataSource.INSTANCE.getConnection(), "data/scripts/createDDL-Derby.sql");
    DataSource.INSTANCE.close();
}
```

```
String DB_CONNECTION_STRING = "jdbc:derby:data/derby/cssdb";
```

## Derby vs MySQL: Scripts de carregamento de dados

```
1 CREATE TABLE APPCONFIG (ID INTEGER PRIMARY KEY NOT NULL, TOTALAMOUNTPERCENTAGE DOUBLE NOT NULL, AMOUNTTHRESHOLD DOUBLE, ELIG
2 CREATE TABLE CUSTOMER (ID INTEGER GENERATED BY DEFAULT AS IDENTITY (START WITH 1, INCREMENT BY 1) PRIMARY KEY NOT NULL, DESI
3 CREATE TABLE UNIT (ID INTEGER GENERATED BY DEFAULT AS IDENTITY (START WITH 1, INCREMENT BY 1) NOT NULL, ABBREVIATION VARCHAR
4 CREATE TABLE PRODUCT (ID INTEGER GENERATED BY DEFAULT AS IDENTITY (START WITH 1, INCREMENT BY 1) PRIMARY KEY NOT NULL, DESC
5 CREATE TABLE SALEPRODUCT (ID INTEGER GENERATED BY DEFAULT AS IDENTITY (START WITH 1, INCREMENT BY 1) PRIMARY KEY NOT NULL, Q
6 CREATE TABLE SALE (ID INTEGER GENERATED BY DEFAULT AS IDENTITY (START WITH 1, INCREMENT BY 1) PRIMARY KEY NOT NULL, DATE DAT
```

```
1 CREATE TABLE APPCONFIG (ID INTEGER PRIMARY KEY NOT NULL, TOTALAMOUNTPERCENTAGE DOUBLE NOT NULL, AMOUNTTHRESHOLD DOUBLE, ELIG
2 CREATE TABLE CUSTOMER (ID INTEGER AUTO_INCREMENT NOT NULL, DESIGNATION VARCHAR(255) NOT NULL, PHONENUMBER INTEGER, VATNUMBER
3 CREATE TABLE UNIT (ID INTEGER AUTO_INCREMENT NOT NULL, ABBREVIATION VARCHAR(255), DESCRIPTION VARCHAR(255), PRIMARY KEY (ID)
4 CREATE TABLE PRODUCT (ID INTEGER AUTO_INCREMENT NOT NULL, DESCRIPTION VARCHAR(255), DISCOUNTELIGIBILITY TINYINT(1) default 0
5 CREATE TABLE SALEPRODUCT (ID INTEGER AUTO_INCREMENT NOT NULL, QTY DOUBLE, PRODUCT_ID INTEGER, SALE_ID INTEGER, PRIMARY KEY (ID)
6 CREATE TABLE SALE (ID INTEGER AUTO_INCREMENT NOT NULL, DATE DATE, TOTAL DOUBLE, DISCOUNT_TOTAL DOUBLE, STATUS VARCHAR(255),
```

## Exercício

### *Active Record*

- Explique em que consiste o padrão *Active Record* focando a sua explicação na forma como é feita a abstração da base de dados e nas responsabilidades atribuídas a cada objeto do padrão.
- Explique as implicações da escolha deste padrão no que diz respeito a organização em camadas.

### *Data Mapper*

Explique em que consiste o padrão *Data Mapper* focando a sua explicação na forma como é feita a abstração da base de dados e nas responsabilidades atribuídas a cada objeto do padrão.

### *O/R Mapping*

- O que designa o termo *Object-relational impedance mismatch*?
- Identifique quais as diferenças fundamentais entre o modelo relacional e o modelo orientado a objetos e quais os principais problemas de mapeamento entre os dois modelos que os *Data Mappers* têm de resolver.

## Exercício

---

### Active Record vs Data Mapper

- (a) O Express é um *web framework* para Node.js com o qual, nesta altura, já deve estar familiarizado. Relacione as boas práticas para estruturar aplicações desenvolvidas com o Express descritas aqui, e parcialmente reproduzidas abaixo, com os padrões arquiteturais abordados em CSS.



## Folders <-> Camadas ?

---

Let's see what files and folders there are at the root of our project with a brief explanation of what each of them is about:

- **controllers/** – defines your app routes and their logic
- **helpers/** – code and functionality to be shared by different parts of the project
- **middlewares/** – Express middlewares which process the incoming requests before handling them down to the routes
- **models/** – represents data, implements business logic and handles storage
- **public/** – contains all static files like images, styles and javascript
- **views/** – provides templates which are rendered and served by your routes
- **tests/** – tests everything which is in the other folders
- **app.js** – initializes the app and glues everything together
- **package.json** – remembers all packages that your app depends on and their

A thing to bear in mind here: it is important not only how you structure your files and folders, but also what each file is responsible for and what it should know about the outside world.



## Padrão?

---

### Models

Models are the files where you interact with your database. They contain all the methods and functions which will handle your data. This includes not only the methods for creating, reading, updating and deleting items, but also any additional business logic. For example, if you had a car model, you could have a mountTyres method.

You should create at least one file for each type of data in your database. In our example, we have users and comments, therefore we have a user model and a comment model. Sometimes, when a model file becomes too large, it might be better to break it into several files, based on some internal logic.

You should also try to make your models independent from the outside world. They don't need to know about other models and they should never include them. They don't need to know about controllers or who uses them. They should never receive request or response objects. They should never return http errors, but they should return model specific errors.

Considere o excerto abaixo retirado do livro *Patterns of Enterprise Application Architecture* do Martin Fowler.

"An OO domain model will often look similar to a database model, yet it will still have a lot of differences. A Domain Model mingles data and process, has multivalued attributes and a complex web of associations, and uses inheritance. As a result I see two styles of Domain Model in the field. A simple Domain Model looks very much like the database design with mostly one domain object for each database table. A rich Domain Model can look different from the database design, with inheritance, strategies, and other GOF patterns, and complex webs of interconnected objects. A rich Domain Model is better for more complex logic, but is harder to map to the database. A simple Domain Model can use Active Record, whereas a rich Domain Model requires Data Mapper."

Compare a complexidade do modelo de domínio orientado a objetos do sistema *AulasGes* com o sistema de informação que vão desenvolver no projeto de PSI para gerir as reservas de uma cadeia de hóteis e relate essa complexidade com os padrões arquiteturais que são propostos utilizar em cada caso.

## OO Domain Model?

- Epic1: Informações sobre os hotéis da cadeia
  - US1 - Como cliente quero saber detalhes sobre os diferentes tipos de quartos de um hotel para escolher o quarto com melhores condições
  - US2 - Como cliente quero saber os preços por noite dos diferentes tipos de quartos de um hotel para escolher um quarto dentro do meu orçamento
  - US3 - Como cliente quero saber a disponibilidade temporal dos diferentes tipos de quartos de um hotel para escolher um quarto que esteja livre no período em que quero fazer a reserva
- Epic2: Gerir reservas de quartos
  - US4 - Como cliente quero poder reservar uma estadia num hotel através do site para não ter de gastar tempo com telefonemas ou deslocações até ao hotel
  - US5 - Como cliente quero saber as datas de check-in e check-out da cada uma das minhas reservas para não me enganar nas datas
  - US6 - Como cliente quero saber o preço de cada uma das minhas reservas para poder gerir o meu orçamento
  - US7 - Como cliente quero mudar as datas das minhas reservas para que alguma alteração na minha agenda não me obrigue a cancelar a reserva