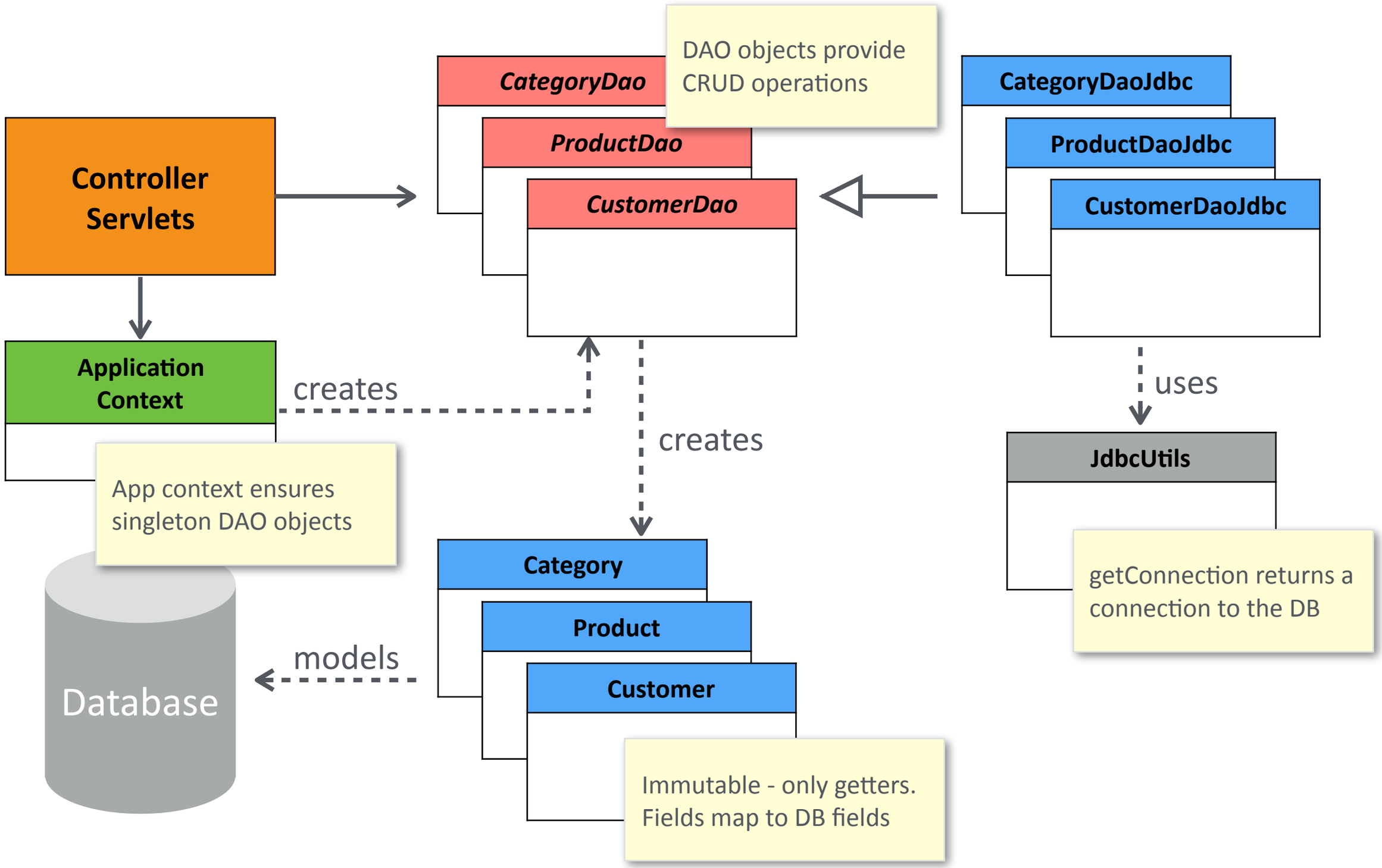
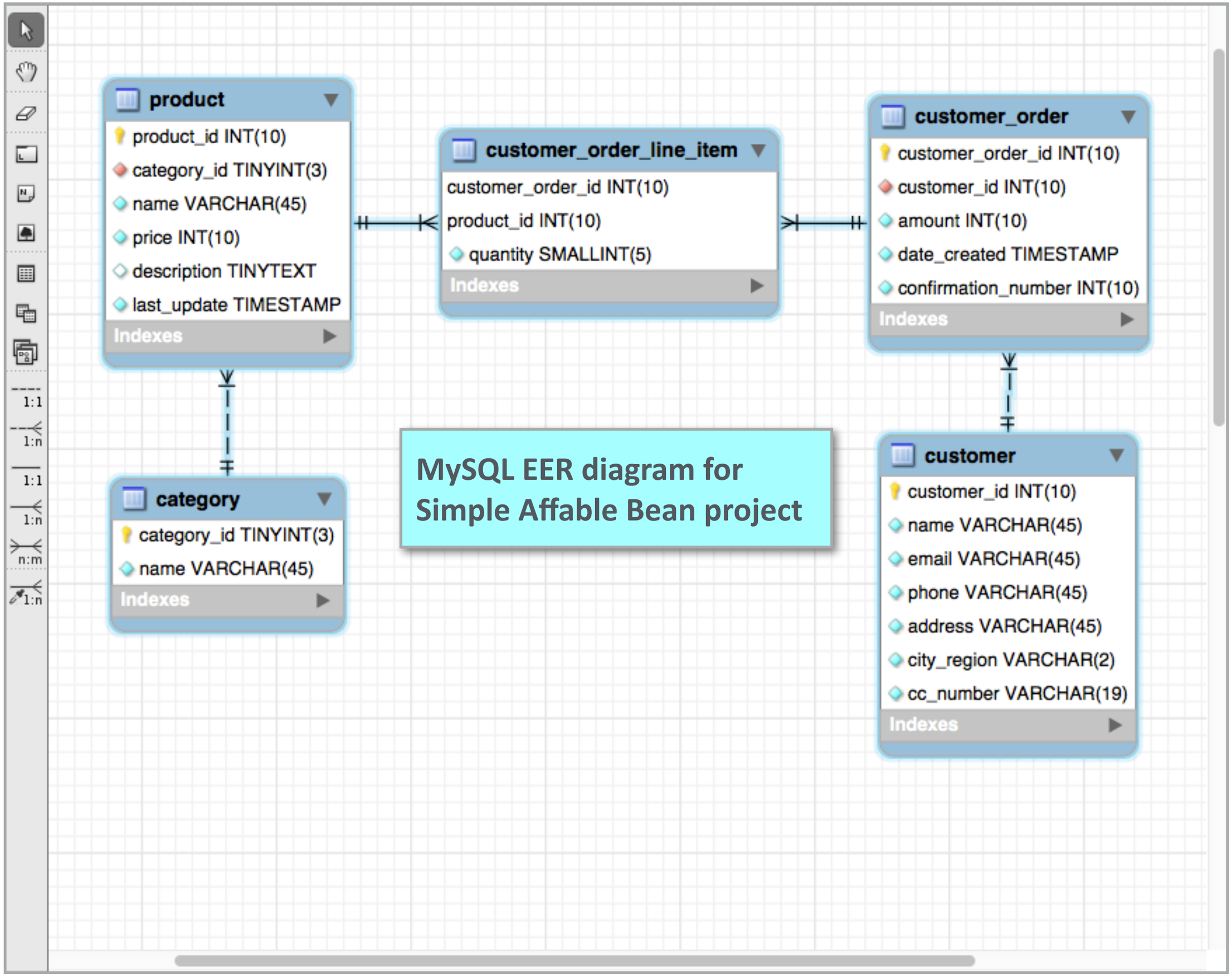
The DAO Pattern
&
Transactions

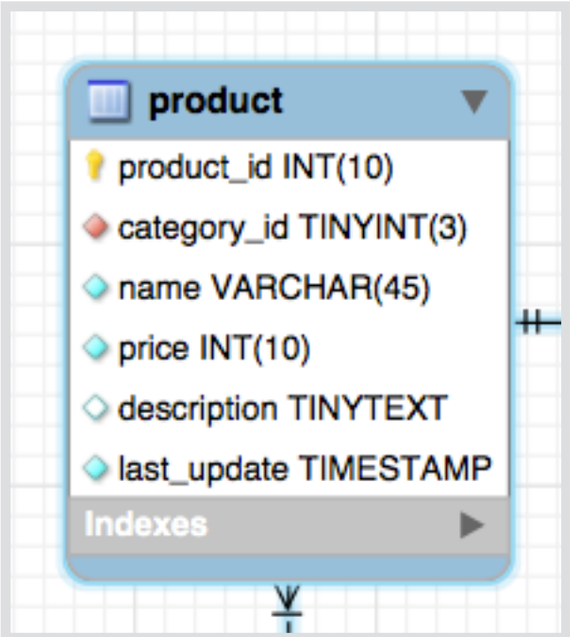
Data Access Object Pattern





MySQL EER diagram for Simple Affable Bean project

Model Class from a DB Table




```
public class Product {

    private long productId;
    private long categoryId;
    private String name;
    private int price;
    private String description;
    private Date lastUpdate;

    public Product(long productId,
                   long categoryId, ...) {
        this.productId = productId;
        this.categoryId = categoryId;
        ...

    public long getProductId() { return productId; }
    public long getCategoryId() { return categoryId; }
    public String getName() { return name; }
    public int getPrice() { return price; }
    public String getDescription() { return description; }
    public Date getLastUpdate() { return lastUpdate; }
}
```

DB fields become fields in model object

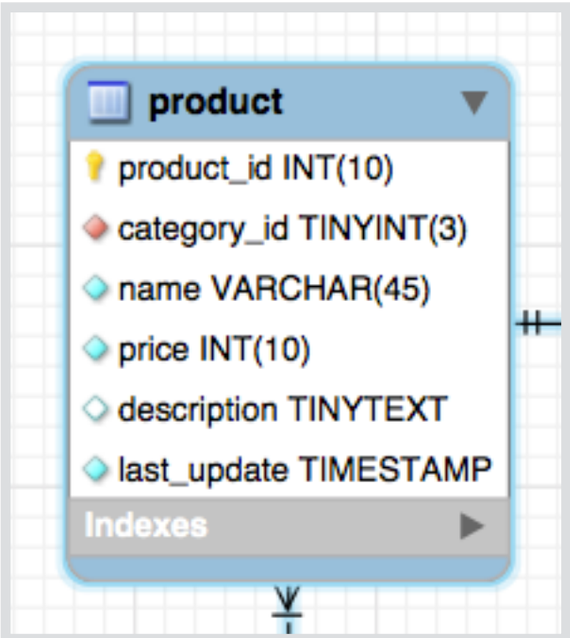
Constructor takes all fields

Getters for all fields

SQL to Java Type Mapping

SQL	Java
INT (primary or foreign key)	long
INT / SMALLINT	int
VARCHAR	String
TIMESTAMP / DATE	java.util.Date
BOOLEAN / TINYINT(1)	boolean

Constructing a DAO Interface




```
public interface ProductDao {

    public List<Product> findAll();
    public Product findByProductId(long productId);
    public List<Product> findByCategoryId(long categoryId);

}
```

A findAll method that returns a list of model objects

A findBy method that takes a primary key and returns a model object

A findBy method for each foreign key that takes a key and returns a list of model objects

Implementing a DAO interface using JDBC

```
public class ProductDaoJdbc {

    private static final String FIND_ALL_SQL =
        "SELECT product_id, category_id, name, price, last_update " +
        "FROM product";

    vate static final String FIND_BY_PRODUCT_ID_SQL =
        "SELECT product_id, category_id, name, price, last_update " +
        "FROM product WHERE product_id = ?";

    private static final String FIND_BY_CATEGORY_ID_SQL =
        "SELECT product_id, category_id, name, price, last_up
        "FROM product WHERE category_id = ?";

    public List<Product> findAll() { ... }
    public Product findByProductId(long productId) { ... }
    public List<Product> findByCategoryId(long categoryId) { ... }

    private Product readProduct(ResultSet resultSet) throws SQLException {
        Product result;
        long productId = resultSet.getLong("product_id");
        String name = resultSet.getString("name");
        int price = resultSet.getInt("price");
        Date lastUpdate = resultSet.getTimestamp("last_update");
        result = new Product(productId, name, price, lastUpdate);
        return result;
    }
}
```

Create constants for each SQL query

Question marks (?) represent parameters

readProduct returns the product in the current row of the result set

Note: the result set is effectively a table

The try-with-resources Statement

- The **try-with-resources** statement is a try statement that declares one or more resources
- A resource is an object that **must be closed** after the program is finished with it
- The try-with-resources statement **ensures that each resource is closed** at the end of the statement
- Any object that **implements java.lang.AutoCloseable** can be used as a resource

– The Java Tutorials

The findAll Method

```
@Override
public List<Product> findAll() {
    List<Product> result = new ArrayList<>();
    try (Connection connection = getConnection();
         PreparedStatement statement = connection.prepareStatement(FIND_ALL_SQL);
         ResultSet resultSet = statement.executeQuery()) {
        while (resultSet.next()) {
            Product p = readProduct(resultSet);
            result.add(p);
        }
    } catch (SQLException e) {
        throw new SimpleAffableQueryDbException(
                "Encountered problem finding all products", e);
    }
    return result;
}
```

This try-by-resources statement has three resources: connection, statement, and resultSet

The result set is essentially an iterator over the rows of the table returned by the SQL query

result is what the method is returning: a list of products

The findByProductId Method

```
@Override
public Product findByProductId(long productId) {
    Product result = null;
    try (Connection connection = getConnection();
         PreparedStatement statement = connection.prepareStatement(FIND_BY_PRODUCT_ID_SQL)) {
        statement.setLong(1, productId);
        try (ResultSet resultSet = statement.executeQuery()) {
            if (resultSet.next()) {
                result = readProduct(resultSet);
            }
        }
    } catch (SQLException e) {
        throw new SimpleAffableQueryDbException(
                "Encountered problem finding product by product i
    }
    return result;
}
```

This try-by-resources statement has two resources

The FIND_BY_PRODUCT_ID_SQL query string has one question mark (takes a parameter), so a setter method must be called to set it

Another try-with-resources statement is used for the resultSet

The if statement says if the result of the query contains something, it must be the desired product

The same catch block catches both try statements

The findByCategoryId Method

```
@Override
public Product findByCategoryId(long categoryId) {
    List<Product> result = new ArrayList<>();
    try (Connection connection = getConnection();
         PreparedStatement statement = connection.prepareStatement(FIND_BY_CATEGORY_ID_SQL)) {
        statement.setLong(1, categoryId);
        try (ResultSet resultSet = statement.executeQuery()) {
            while (resultSet.next()) {
                Product p = readProduct(resultSet);
                result.add(p);
            }
        }
    } catch (SQLException e) {
        throw new SimpleAffableQueryDbException(
                "Encountered problem finding products by category id", e);
    }
    return result;
}
```

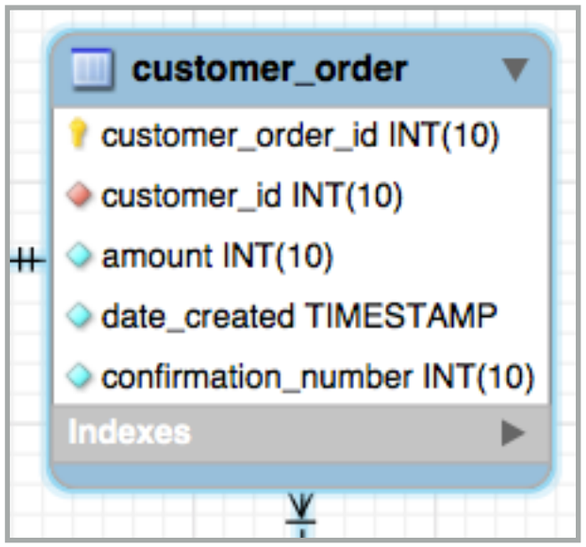
Finding products by the foreign key category id is similar to finding products by the product id, except that here a list of products is returned

The create Method Interface

```
public interface CustomerOrderDao {

    public long create(final Connection connection, long customerId,
                       int amount, int confirmationNumber);

    public List<CustomerOrder> findAll();
    public CustomerOrder findByCustomerOrderId(long customerOrderId);
    public List<CustomerOrder> findByCustomerId(long customerId);
```



4 items view cart

english česky

the affable bean

checkout

In order to purchase the items in your shopping cart, please provide us with the following information:

name: Mozart

email: wolfy@hotmail.com

phone: 251640793

address: Opera House

Prague 1

credit card number: 5555555555554444

submit purchase

- Next-day delivery is guaranteed
- A € 3.00 delivery surcharge is applied to all purchase orders

subtotal:	€ 8.52
delivery surcharge:	€ 3.00
total:	€ 11.52

0 items

the affable bean

Your order has been successfully processed and will be delivered within 24 hours.

Please keep a note of your confirmation number: **280296519**
If you have a query concerning your order, feel free to contact us.

Thank you for shopping at the Affable Bean Green Grocer!

order summary

product	quantity	price
cheese	1	€ 2.39
sausages	1	€ 3.55
broccoli	2	€ 2.58
	delivery surcharge:	€ 3.00
	total:	€ 11.52

date processed: 4/19/17 6:46 PM

delivery address

Mozart
Opera House
Prague 1

email: wolfy@hotmail.com
phone: 251640793

Shop again

admin console

view all customers

view all orders

log out

customers

customer id	name	email	phone
1	Einstein	emc2@cuni.cz	224491850
2	Kafka	vermin@books.cz	224934203
3	Mozart	wolfy@hotmail.com	251640793

view all customers

view all orders

log out

orders

order id	confirmation number	amount	date created
1	492945651	€ 9.07	4/19/17 6:37 PM
2	965900691	€ 9.73	4/19/17 6:41 PM
3	280296519	€ 11.52	4/19/17 6:46 PM

Customer Data (from form)

- ✓ **name**: Mozart
- ✓ **email**: wolfy@hotmail.com
- ✓ **phone**: 251-640-793
- ✓ **address**: Opera House
- ✓ **region**: 1
- ✓ **cc number**: 5555555555554444

Order Data (from shopping cart)

- ✓ **product$_1$**: cheese
- ✓ **product$_2$**: sausages
- ✓ **product$_3$**: broccoli (x2)

customer_order_line_item

customer_order_id	product_id	quantity
1	10	1
1	12	1
1	14	1
2	8	1
2	13	2

customer_order

customer_order_id	customer_id	amount	date created	confirmation_number
1	1	907	2017-04-19 18:37:48	492945651
2	2	973	2017-04-19 18:41:30	965900691

customer

customer_id	name	email	phone	address	city_region	cc_number
1	Einstein	emc2@cuni.cz	224-491-850	Charles University	1	6011111111111117
2	Kafka	vermin@books.cz	224-934-203	Courthouse	1	4111111111111111

Customer Data (from form)

- ✓ **name**: Mozart
- ✓ **email**: wolfy@hotmail.com
- ✓ **phone**: 251-640-793
- ✓ **address**: Opera House
- ✓ **region**: 1
- ✓ **cc number**: 5555555555554444

Order Data (from shopping cart)

- ✓ **product$_1$**: cheese
- ✓ **product$_2$**: sausages
- ✓ **product$_3$**: broccoli (x2)

In case of error the data is not entered, the next primary key will be 4

The customer **create** method reserves a row in the table and a reserves and **returns** its key

customer_order_line_item

customer_order_id	product_id	quantity
1	10	1
1	12	1
1	14	1
2	8	1
2	13	2

customer_order

customer_order_id	customer_id	amount	date created	confirmation_number
1	1	907	2017-04-19 18:37:48	492945651
2	2	973	2017-04-19 18:41:30	965900691

customer

customer_id	name	email	phone	address	city_region	cc_number
1	Einstein	emc2@cuni.cz	224-491-850	Charles University	1	6011111111111117
2	Kafka	vermin@books.cz	224-934-203	Courthouse	1	4111111111111111
3	Mozart	wolfy@hotmail.com	251-640-793	Opera House	1	5555555555554444

Customer Data (from form)

- ✓ **name**: Mozart
- ✓ **email**: wolfy@hotmail.com
- ✓ **phone**: 251-640-793
- ✓ **address**: Opera House
- ✓ **region**: 1
- ✓ **cc number**: 5555555555554444

Order Data (from shopping cart)

- ✓ **product$_1$**: cheese
- ✓ **product$_2$**: sausages
- ✓ **product$_3$**: broccoli (x2)

In case of error the data is not entered, the next primary key will be 4

The customer key is used in customer order **create**. It also returns its key

The customer **create** method reserves a row in the table and a reserves and **returns** its key

customer_order_line_item

customer_order_id	product_id	quantity
1	10	1
1	12	1
1	14	1
2	8	1
2	13	2

customer_order

customer_order_id	customer_id	amount	date created	confirmation_number
1	1	907	2017-04-19 18:37:48	492945651
2	2	973	2017-04-19 18:41:30	965900691
3	3	1152	2017-04-19 18:46:32	280296519

customer

customer_id	name	email	phone	address	city_region	cc_number
1	Einstein	emc2@cuni.cz	224-491-850	Charles University	1	6011111111111117
2	Kafka	vermin@books.cz	224-934-203	Courthouse	1	4111111111111111
3	Mozart	wolfy@hotmail.com	251-640-793	Opera House	1	5555555555554444

Customer Data (from form)

- ✓ **name**: Mozart
- ✓ **email**: wolfy@hotmail.com
- ✓ **phone**: 251-640-793
- ✓ **address**: Opera House
- ✓ **region**: 1
- ✓ **cc number**: 5555555555554444

Order Data (from shopping cart)

- ✓ **product$_1$**: cheese
- ✓ **product$_2$**: sausages
- ✓ **product$_3$**: broccoli (x2)

The customer order key is used in customer order line item **create**. This table does not have a distinct primary key, so it cannot return one

customer_order_line_item

customer_order_id	product_id	quantity
1	10	1
1	12	1
1	14	1
2	8	1
2	13	2
3	2	1
3	8	1
3	15	2

In case of error the data is not entered, the next primary key will be 4

The customer key is used in customer order **create**. It also returns its key

customer_order

customer_order_id	customer_id	amount	date created	confirmation_number
1	1	907	2017-04-19 18:37:48	492945651
2	2	973	2017-04-19 18:41:30	965900691
3	3	1152	2017-04-19 18:46:32	280296519

The customer **create** method reserves a row in the table and a reserves and **returns** its key

customer

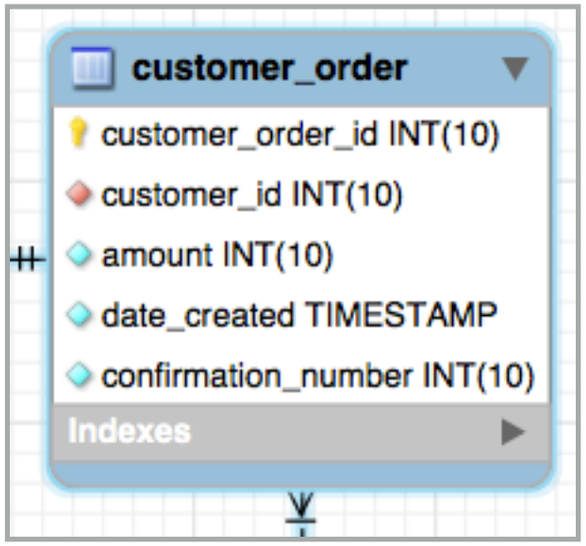
customer_id	name	email	phone	address	city_region	cc_number
1	Einstein	emc2@cuni.cz	224-491-850	Charles University	1	6011111111111117
2	Kafka	vermin@books.cz	224-934-203	Courthouse	1	4111111111111111
3	Mozart	wolfy@hotmail.com	251-640-793	Opera House	1	5555555555554444

The create Method Interface

```java
public interface CustomerOrderDao {

    public long create(final Connection connection, long customerId,
                       int amount, int confirmationNumber);

    public List<CustomerOrder> findAll();
    public CustomerOrder findByCustomerOrderId(long customerOrderId);
    public List<CustomerOrder> findByCustomerId(long customerId);
```



The create method does not take a primary key (customer order ID) or a date created parameter. Both of these are generated automatically by the DB when a new record is added to the customer_order table.

A result of type long is returned because we need the customer order ID for use in a line item create method during a transaction. The methods used in the DAOs are driven by our needs in the presentation layer

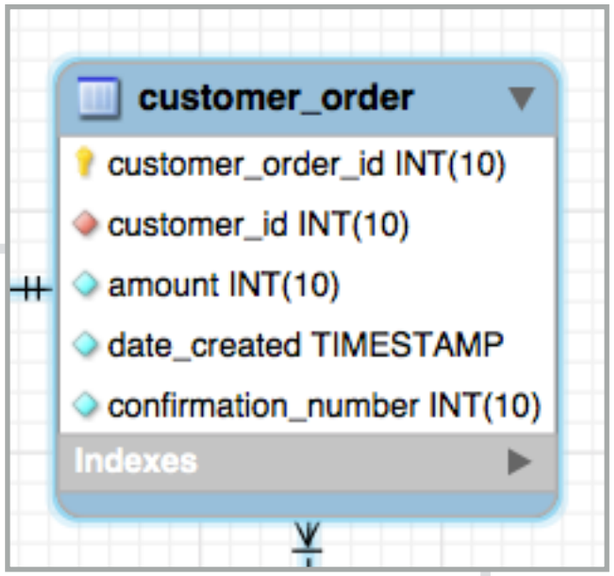
Question: Why are we passing in a connection object to the create method instead of declaring it as a resource in our try-with-resources statement?

Note: Passing in a connection object breaks encapsulation as not all data sources require a connection object (but all JDBC-compatible data sources do)

Answer: We want to be able to create a customer order as part of a larger transaction. To do this atomically, we defer committing all updates until we have them. In effect, this create method will reserve this update (along with the auto-generated primary key) until it is committed by the client

The SQL INSERT statement has three parameters: one for each value

The create Method




```java
private static final String CREATE_ORDER_SQL =
    "INSERT INTO customer_order " +
        "(amount, customer_id, confirmation_number) " +
        "VALUES (?, ?, ?)";


@Override
public long create(final Connection connection, long customerId,
                   int amount, int confirmationNumber) {
    try (PreparedStatement statement = /* statement based on CREATE_ORDER_SQL */ ) {
        // set parameters: 1-amount, 2-customerId, 3-confirmationNumber
        int affected = statement.executeUpdate();
        if (affected != 1) {
            // throw an exception
        }
        long customerOrderId = 0;
        // use statement to get result set with generated keys
        if ( /* result set is not empty */ ) {
            customerOrderId = /* key from result set */
        } else {
            // throw an exception
        }
        return customerOrderId;
    } catch (SQLException e) {
        // throw an exception
    }
}
```

A prepared statement is generated using the connection and the SQL string. The statement is completed by setting the thee parameters missing from the string

The executeUpdate call executes the SQL INSERT statement and returns the number of rows that were affected

The statement object holds the keys generated by the update. Here, there is only 1

Assuming the insert is successful, the rows affected should be 1

If no problems occur, that key is returned

A Complete create Method

```
@Override
public long create(final Connection connection, long customerId,
                   int amount, int confirmationNumber) {
    try (PreparedStatement statement =
            connection.prepareStatement(CREATE_ORDER_SQL,
                                        Statement.RETURN_GENERATED_KEYS)) {
        statement.setInt(1, amount);
        statement.setLong(2, customerId);
        statement.setInt(3, confirmationNumber);
        int affected = statement.executeUpdate();
        if (affected != 1) {
            throw new SimpleAffableUpdateDbException(
                "Failed to insert an order, affected row count = " + affected);
        }
        long customerOrderId;
        ResultSet resultSet = statement.getGeneratedKeys();
        if (resultSet.next()) {
            customerOrderId = resultSet.getLong(1);
        } else {
            throw new SimpleAffableQueryDbException(
                "Failed to retrieve customerOrderId auto-
        }
        return customerOrderId;
    } catch (SQLException e) {
        throw new SimpleAffableUpdateDbException(
            "Encountered problem creating a new customer ", e);
    }
}
```

Remember, we expect the connection object to have auto-commit set to false. If that is the case, then even after this method ends, the DB will not allow the record to be accessed. The client handling the complete transaction will have to commit first.

Customer Order Service Interface

```
public interface CustomerOrderService {

    long placeOrder(String name, String email, String phone, String address,
                    String cityRegion, String ccNumber, ShoppingCart cart);

    CustomerOrderDetails getOrderDetails(long customerOrderId);
}
```

The CustomerOrderService component
1. performs a transaction for an order, and
2. allows clients to retrieve information associated with that order

placeOrder takes all the information needed to perform a transaction for an order, and returns the customer order ID for the order created in the database. If the transaction fails, no order is created and the method returns 0

getOrderDetails takes a customer order ID and returns all the relevant order details

Default Customer Order Service (placeOrder)

```
public class DefaultCustomerOrderService implements CustomerOrderService {

    private CustomerOrderDao customerOrderDao;
    private CustomerOrderLineItemDao customerOrderLineItemDao;
    private CustomerDao customerDao;
    private ProductDao productDao;
    private Random random = new Random();

    @Override
    public long placeOrder(String name, String email, String phone, String address,
                           String cityRegion, String ccNumber, ShoppingCart cart) {
        try (Connection connection = JdbcUtils.getConnection()) {
            return performPlaceOrderTransaction(name, email, phone, address,
                                                cityRegion, ccNumber, cart, connection);
        } catch (SQLException e) {
            throw new SimpleAffableDbException("Error during close connection ...
        }
    }

    private long performPlaceOrderTransaction(String name, String email, ... ) { ... }

    private int generateConfirmationNumber() { return random.nextInt(999999999); }
    public void setCustomerOrderDao(CustomerOrderDao customerOrderDao) {
        this.customerOrderDao = customerOrderDao;
    }
    ...
}
```

The DAO's are needed to create data during the transaction

The placeOrder method gets a JDBC connection and then calls through to performPlaceOrderTransaction

Question: Why don't we use the ApplicationContext to get the DAO implementations?

Answer: If we restrict the use of the ApplicationContext to the presentation layer, we avoid excessive coupling in the business layer. Also, allowing clients to set custom DAOs facilitates unit testing.

Default Customer Order Service (placeOrder)

```
private long performPlaceOrderTransaction(String name, String email, String phone,
                                          String address, String cityRegion,
                                          String ccNumber, ShoppingCart cart,
                                          Connection connection) {
    try {
        connection.setAutoCommit(false);
        long customerId = customerDao.create(connection, name, email, phone,
                                             address, cityRegion, ccNumber);
        long customerOrderId = customerOrderDao.create(connection, customerId,
                                                       cart.getTotal(),
                                                       generateConfirmationNumber());
        for (ShoppingCartItem item : cart.getItems()) {
            customerOrderLineItemDao.create(connection, customerOrderId,
                                            item.getProductId(), item.getQuantity()));
        }
        connection.commit();
        return customerOrderId;
    } catch (Exception e) {
        try {
            connection.rollback();
        } catch (SQLException e1) {
            throw new SimpleAffableDbException("Failed to roll back transaction", e1);
        }
        return 0;
    }
}
```

Do not automatically commit the data to the database when you create it. Once you have created all the data associated with the transaction, try to commit it. If the commit succeeds, return the order ID; if it fails, rollback and return 0