🏠_Home_

Layerd simple.png
layered architecture.png

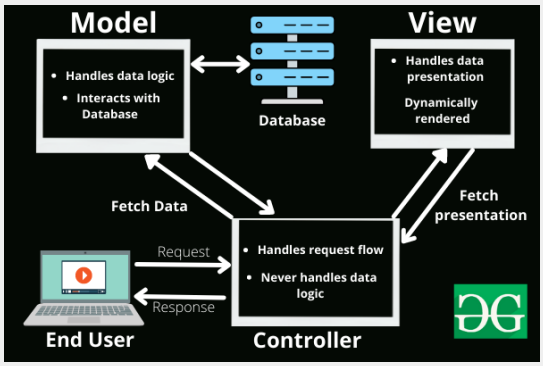
Software Architecture

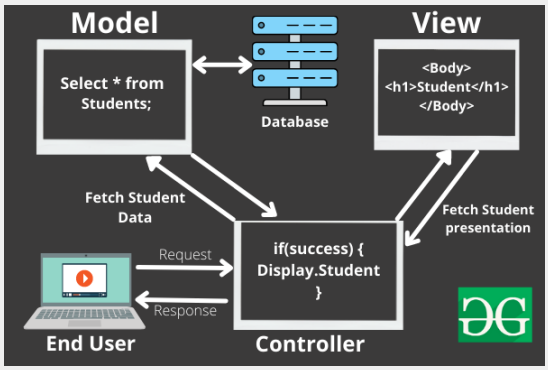
Software Architecture - fundamental structure of a software system
(Software system එකක මූලික ව්‍යුහය)

Architecture Patterns (කතා කරන්නෙ patterns 2ක් ගැන)

1. MVC Architecture
2. Layard Architecture

1. **MVC Architecture** (Model, View, Controller)





(MVC Example)

*** Without MVC,**

* Hard to fix bugs
* Complex code
* Time consuming
* No Reusability

2. **Layered Architecture** (MVC Architecture එකනේ extend කරපු Architecture එකක්)
Layers 4 යි.
    1. **Presentation Layer** → *view*
    2. **Business Layer**/ Service Layer → *(Handle user data using Persistence layer and Data layer, Heart of the application)*
    3. **Persistence Layer**/ Repository / DAO (Data Access Object) → *query*
    4. **Data Layer** → *Database*

Design principles

Design principles - Expert dev. විසින් Software එකක Standard එක maintain කරන්න දාපු Rules.
Design principles 4 ක් තියෙනවා, violate වෙන්න බෑ.

1. Loose coupling
2. Dependency Injection
3. Less Boilerplate codes
4. High Cohesion

1. Loose coupling

යම් class එකක්, තවත් class එකක් මත direct depend වලා නම් - **Tight coupling**

```
public class Demo {
    public static void main(String[] args) {
        B b = new B();
        b.returnA();
    }
}

// top level class // class ගහනාවක පාවිච්චි වෙන class එකක්
class  A {
    public void getA(){
        System.out.println("giving A");
        // top level class එකේ වෙනස්කම් වුනොත් අවුල්.
    }
}

// low level class
class B{
    public void returnA(){
        A a = new A();
        a.getA();
        // tight coupling (low level class එකක්, top level class එකක් මත directly depend වලා තිබීම.)
    }
}
```

යම් class එකක්, තවත් class එකක් මත direct depend වලා නැත්නම් - **Loose coupling**

```
public class Demo{
    public static void main(String[] args) {
        B b = new B();
        b.returnA();
```

```
    }
}

interface SuperA{  // interface එකක් = agreement එකක්
    void getA();
}

// top level class
class  A implements SuperA{ //
    public void getA(){
        System.out.println("giving A");
        // top level class එකේ වෙනස්කම් කරන්න බෑ. (implement වුණු නිසා)
    }
}

// low level class
class B{
    public void returnA(){
        // loosely coupling // run time polymorphism use වෙන්නෙ
        SuperA a = new A();
        a.getA();
    }
}
```

2. Dependency Injection

Class එකක් , තව class එකක් මත depend වලා තියේ නම්, ඒ dependency එක meaningfully implement කරන්න පුලුවන් mechanism එකක්.

DI methods

- **Property injection** - class එක create කරන වලොවම dependency inject කිරීම.

```
public class D1 {
    public static void main(String[] args) {
        Boy b = new Boy();
        b.cattingWithGirl();
    }
}

interface GoodGirl{
    void chatting();
}

class Girl implements GoodGirl{
    @Override
    public void chatting() {
        System.out.println("Hi");
    }
}
```

```
class Boy{
    GoodGirl girl = new Girl(); //(1) property inject

    public void cattingWithGirl(){
        //Loose Coupling Applied
        girl.chatting();
    }
}
```

- **Constructor injection** - Object එක create කරන වෙලාවෙ dependency inject කිරීම.
  (Constructor trough)

```
public class D2 {
    public static void main(String[] args) {
        Boy b = new Boy(new Girl());
        b.cattingWithGirl();
    }
}

interface GoodGirl{
    void chatting();
}

class Girl implements GoodGirl{
    @Override
    public void chatting() {
        System.out.println("Hi");
    }
}

class Boy{
    GoodGirl girl ;

    // (2) constructor injection
    Boy(Girl girl){
        this.girl = girl;
    }

    public void cattingWithGirl(){
        //Loose Coupling Applied
        girl.chatting();
    }
}
```

- **Setter method injection** - Setter method එකක් trough dependency inject කිරීම.

```
public class D3 {
    public static void main(String[] args) {
        Boy b = new Boy();
        b.setInject(new Girl());
        b.cattingWithGirl();
    }
```

```
}

interface GoodGirl{
    void chatting();
}

class Girl implements GoodGirl{
    @Override
    public void chatting() {
        System.out.println("Hi");
    }
}

class Boy{

    //(3) Setter method injection
    GoodGirl girl ;
    public void setInject(Girl girl){
        this.girl = girl;
    }

    public void cattingWithGirl(){
        //Loose Coupling Applied
        girl.chatting();
    }
}
```

- **Interface trough injection** - interface එකක් trough dependency inject කිරීම.

```
public class D4 {
    public static void main(String[] args) {
        Boy b = new Boy();
        b.setInject(new Girl());
        b.cattingWithGirl();
    }
}

interface GoodGirl{
    void chatting();
}

class Girl implements GoodGirl{
    @Override
    public void chatting() {
        System.out.println("Hi");
    }
}

// (4)  Interface trough injection
interface DI{
    void setInject(Girl girl);
}
```

```java
class Boy implements DI{
    GoodGirl girl;

    @Override
    public void setInject(Girl girl) {
        this.girl = girl;
    }

    public void cattingWithGirl(){
        //Loose Coupling Applied
        girl.chatting();
    }
}
```

3. Less Boilerplate codes

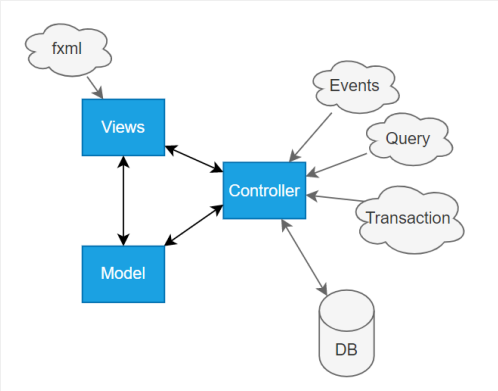
Non-Repetitive & Less Complex Codes

4. High Cohesion

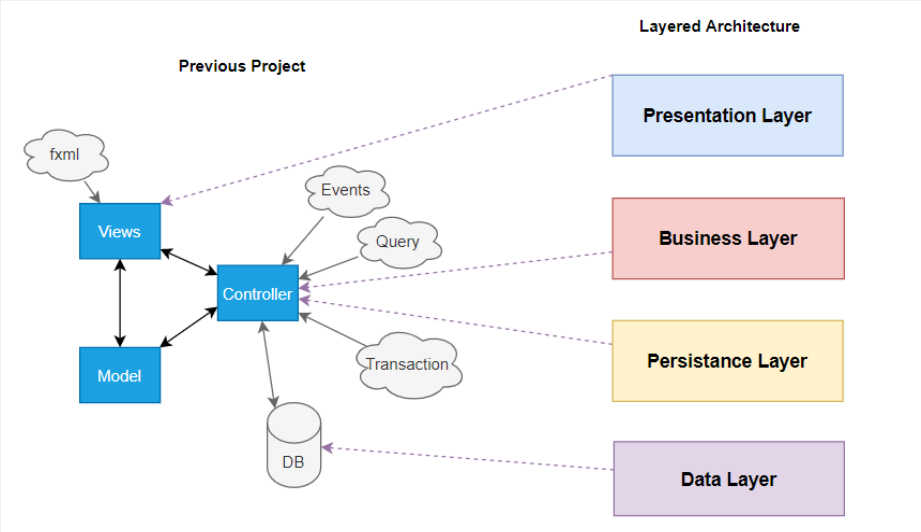
Unit එකක තියෙනෙ ඒකට අදාල දේවල් විතරයි.
(Customer class එක ඇතුලෙ තියෙනෙ customerට අදාල දේවල් විතරයි.)

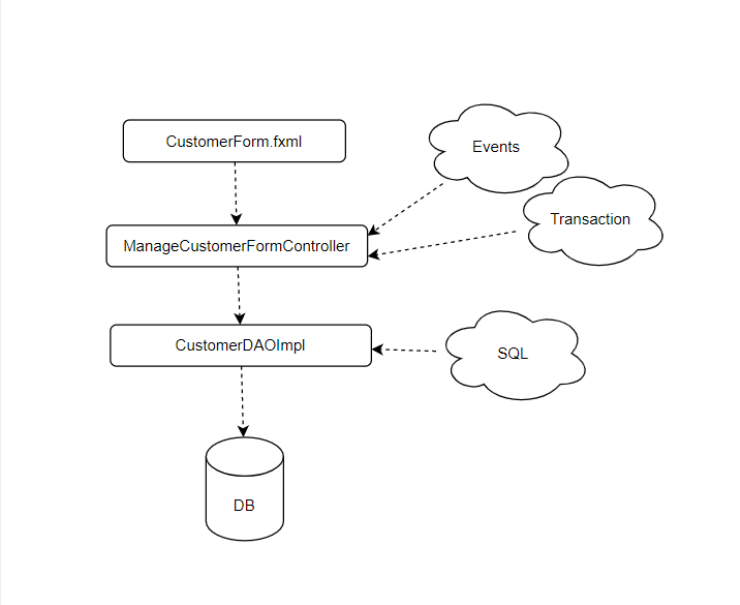
====

Without Layered Architecture (MVC Architecture)

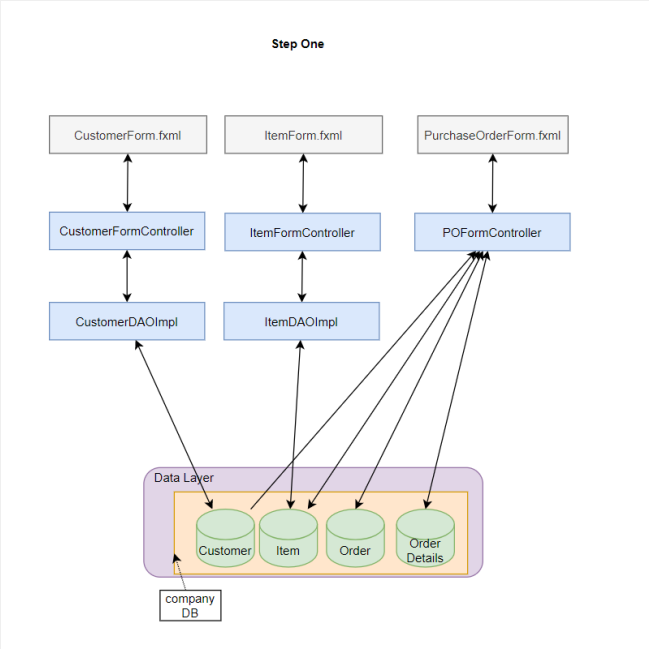


Extending MVC to Layered Architecture

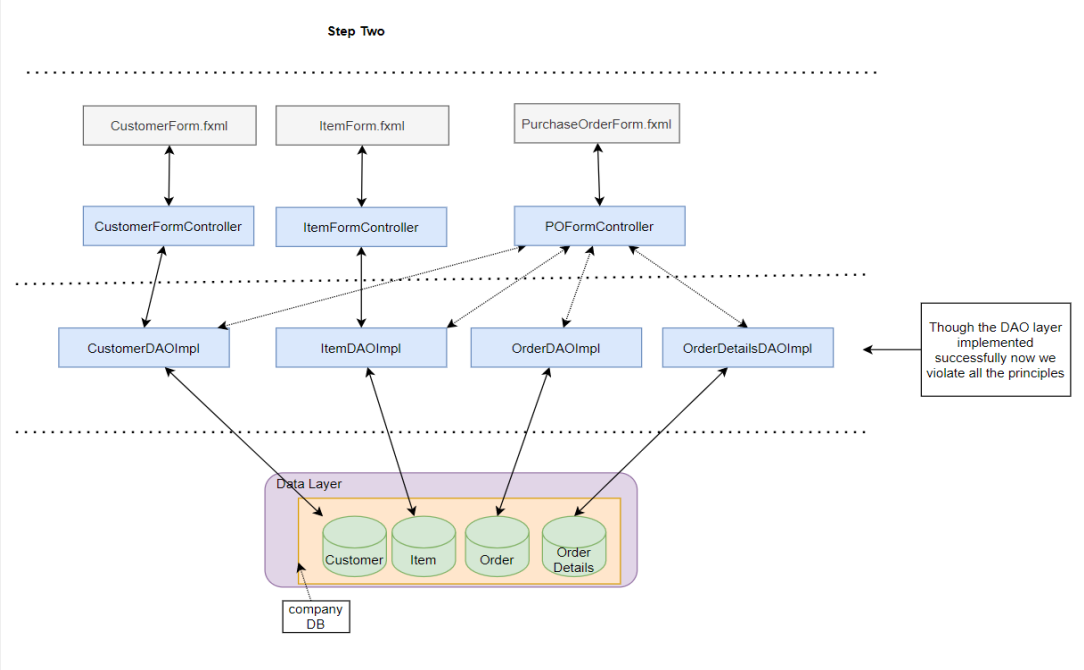




1. **DAO Layer / Persistence Layer** (DAO - Data Access Object)
   Controller එකේ තියනෙ Data layer එක Access කරන්න ඕන SQL Query ටික DAO layer එකට වනේ කිරීම.
   (Database එකේ table වලට අදාලව DAOImpl classes)

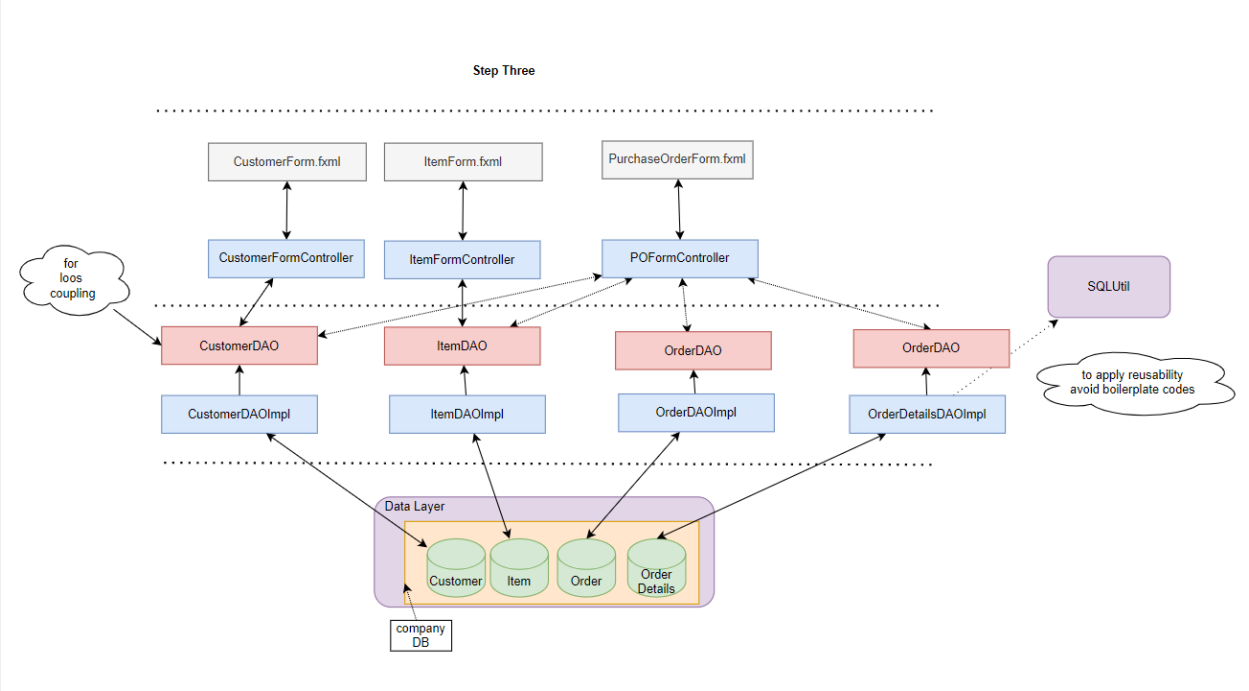


2. Transaction handle කරන්න Order table එකටයි OrderDetails table එකටයි වෙන වෙනම DAOImpl class 2ක් හදනේා.
(මේ වදේදි Design principles violate වලො.)



3. DAO interfaces ⍰ Loose coupling (Avoid Tight Coupling)

SQLUtil ❓ Boiler plate codes අඩු කරන්න, code reusability එක වැඩිකරන්න



Loose coupling

```
                //Add Customer
-               CustomerDAOImpl customerDAO = new CustomerDAOImpl();
+               CustomerDAO customerDAO = new CustomerDAOImpl();
                customerDAO.addCustomer(new CustomerDTO(id,name,address));
```

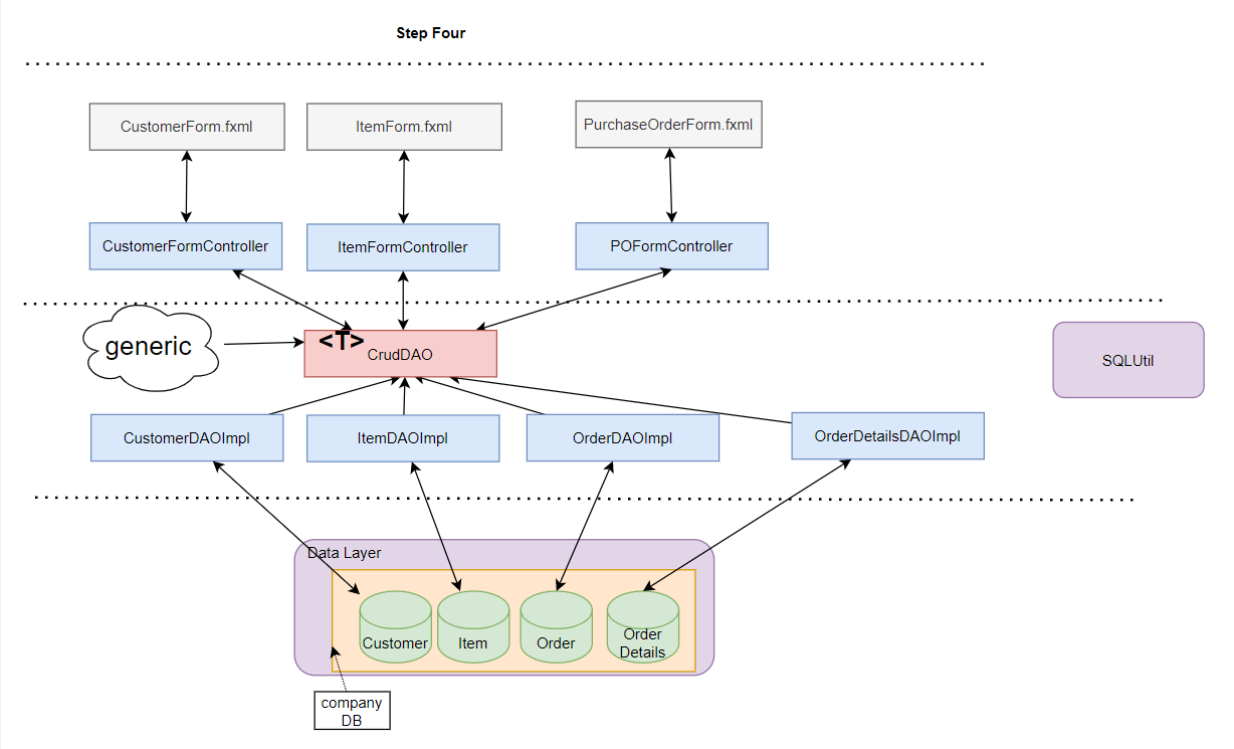
Dependency Injection (property Injection)

```
45
46  +    //DI (Property Injection)
47  +    CustomerDAO customerDAO = new CustomerDAOImpl();
48  +
49       public void initialize() {
50           tblCustomers.getColumns().get(0).setCellValueFactory(new PropertyVal
51           tblCustomers.getColumns().get(1).setCellValueFactory(new PropertyVal
     @@ -74,7 +77,6 @@ private void loadAllCustomers() {
77           tblCustomers.getItems().clear();
78           try {
79               /*Get all customers*/
     -           CustomerDAO customerDAO = new CustomerDAOImpl();
80               ArrayList<CustomerDTO> allCustomers = customerDAO.getAllCustomer
```

SQLUtil for Less Boilerplate codes

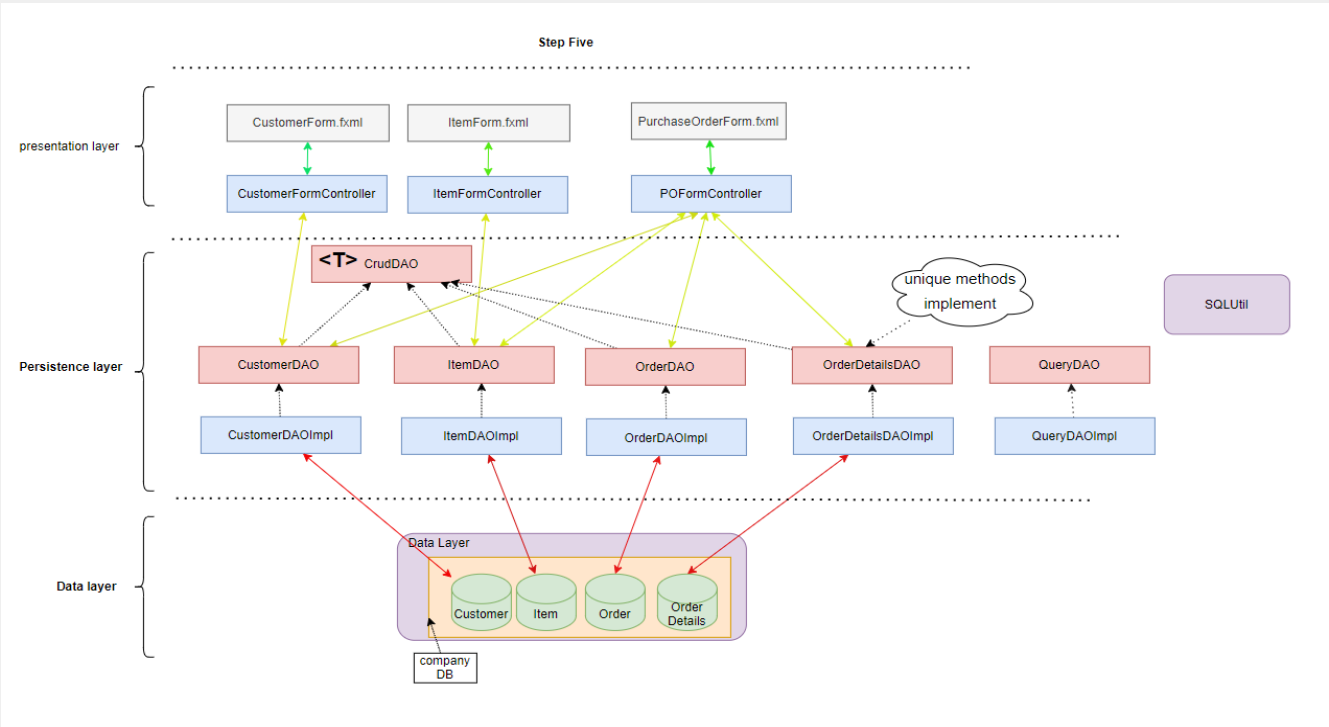
```
13       public ArrayList<CustomerDTO> getAllCustomers() throws SQLException, ClassNotFoundException {
14           ArrayList<CustomerDTO> allCustomers = new ArrayList<>();
     -       Connection connection = DBConnection.getDbConnection().getConnection();
     -       Statement stm = connection.createStatement();
     -       ResultSet rst = stm.executeQuery("SELECT * FROM Customer");
     -
15   +       ResultSet rst =  SQLUtil.execute("SELECT * FROM Customer");
16           while (rst.next()) {
```

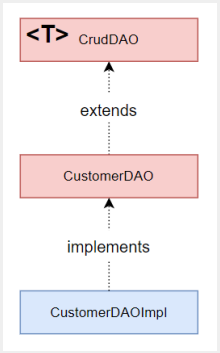
4. CrudDAO - Generics use කරල , පොදු method (Crud operations tika) ටික දාල CrudDAO interface එක හදනෙ.



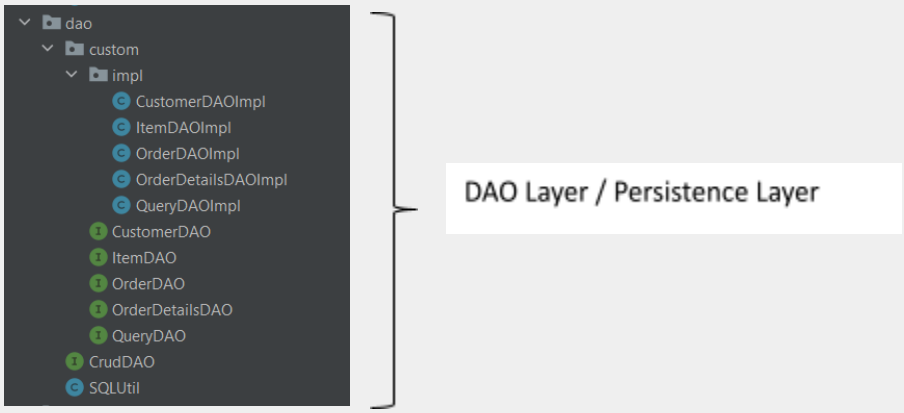

```
10  - public class CustomerDAOImpl implements CustomerDAO {
10  + public class CustomerDAOImpl implements CrudDAO<CustomerDTO> {
```

5. Unique methods දාන්න ඕන උනොත් ඒව දාන්න DAO interfaces ආයෙත් හදනෙ; ඒව crudDAO interface එකත් එක්ක extend වලෙ තියනේනෙ.
   - QuarryDAO - Join Quarry දාන්න.





CrudDAO interface එකෙ methods & CustomerDAO interface එකෙ methods
CustomerDAOImpl class එකට override කරන්න පුලුවන්.



Business Layer

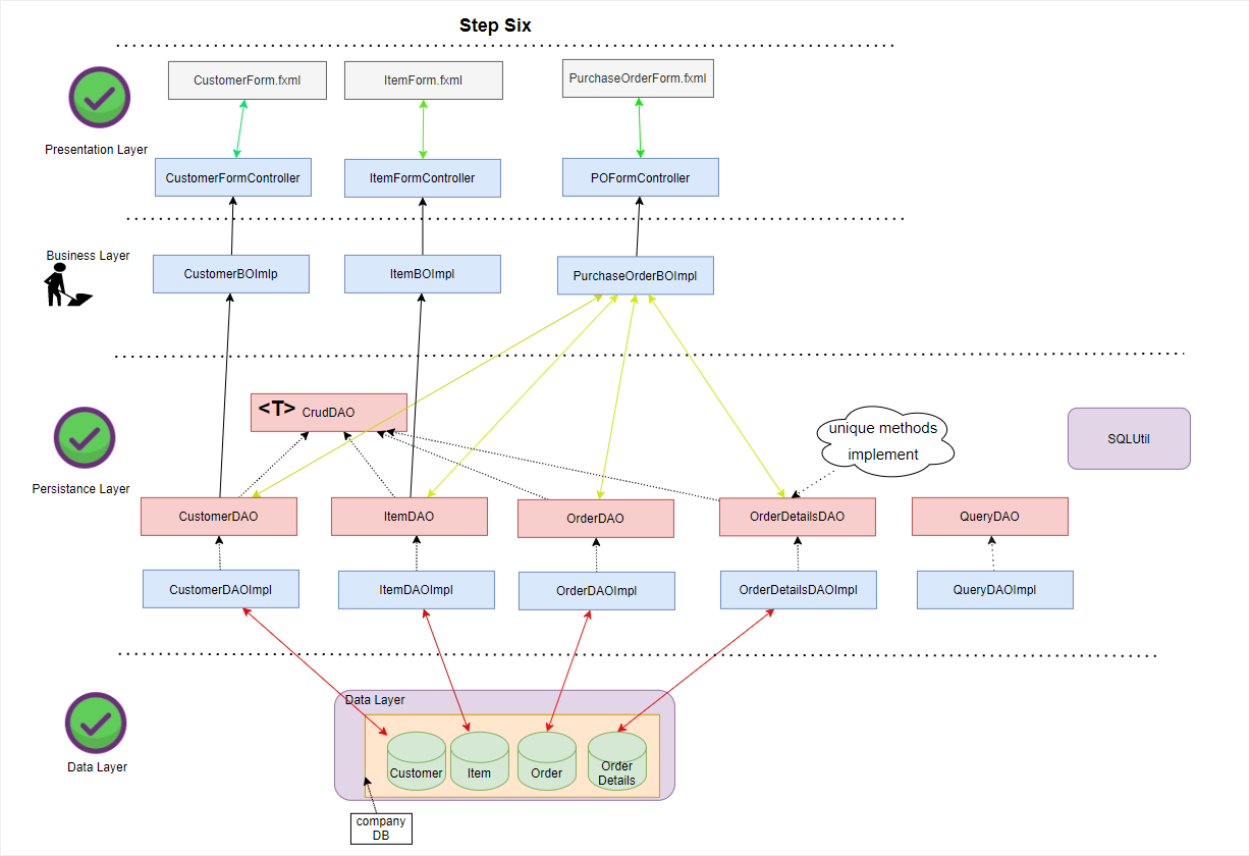
6.

- එක UI එකකට එක Business layer class එකක් තියනේා.
- business layer එක ඇතුලෙ business එක fulfill කරන ඔක්කොම method තියනේන ඕන. business layer එකෙ common method ලියන්නෙ නෑ, (add, delete... වගේ ඒවා)

- Presentation Layer එකයි DAO Layer එකයි අතර connection තියනේන බෑ.
  Business Layer එක හරහා connection හදාගන්නෙ.

```
CustomerDAO customerDAO = new CustomerDAOImpl();
Replaced with
CustomerBOImpl customerBO = new CustomerBOImpl();
```

- Controller එකෙ තියනේන පුලුවන් event ටික විතරයි. (transaction එක Business layer එකට refactor කරා)



7.

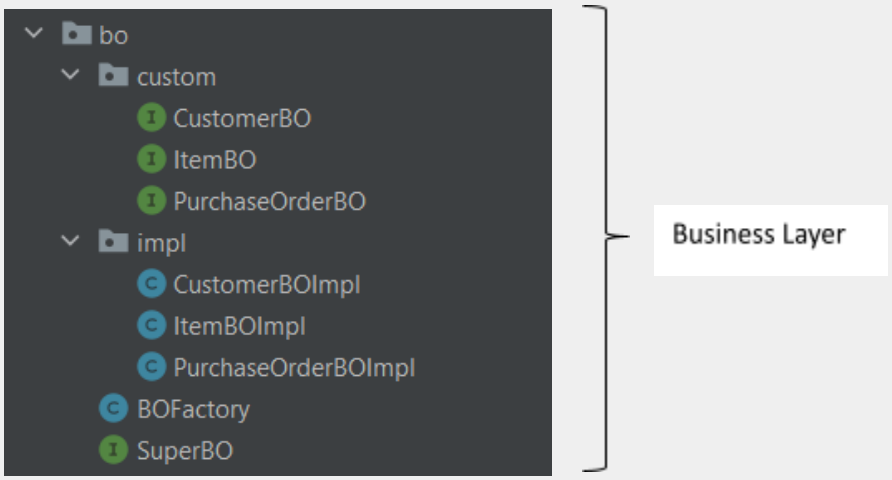
- Loose coupling applied.
- එක layer එකක object creation තව layer එකක කරන්නෙ නෑ. ඒක නවත්තන්න BOFactory class එක use කරනේා.
  (singleton applied)

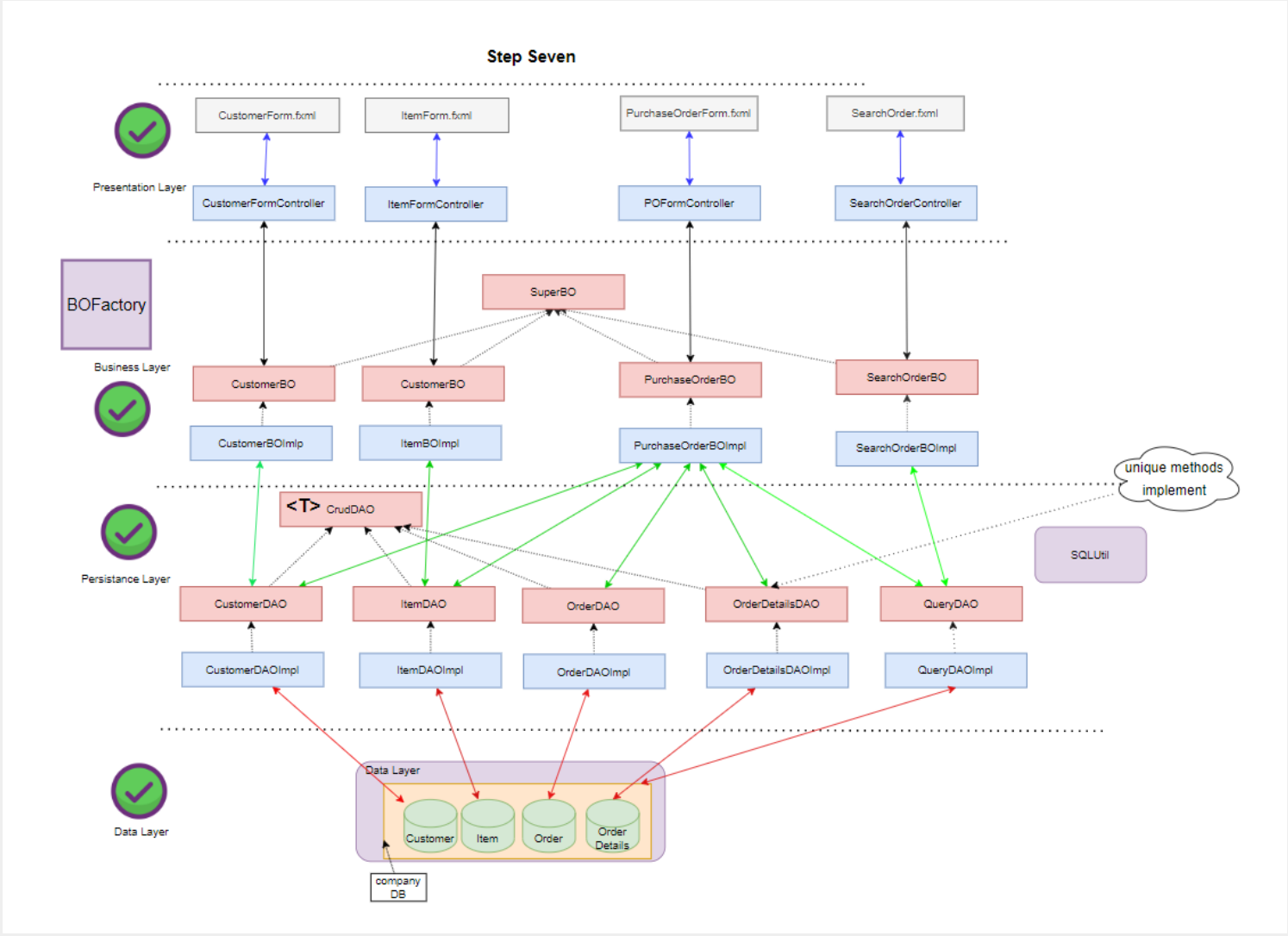
```
CustomerDAO customerDAO = new CustomerDAOImpl();
Replaced with
CustomerBO customerBO = (CustomerBO)
BOFactory.getBoFactory().getBOType(BOFactory.BOTypes.CUSTOMER);
```

- BOFactory class එකේ enum class එකක් හදල ඉල්ලන type එකට අදාල object එක return කරනේා.
  return type එකට Object දාන්න පුලුවන් ,

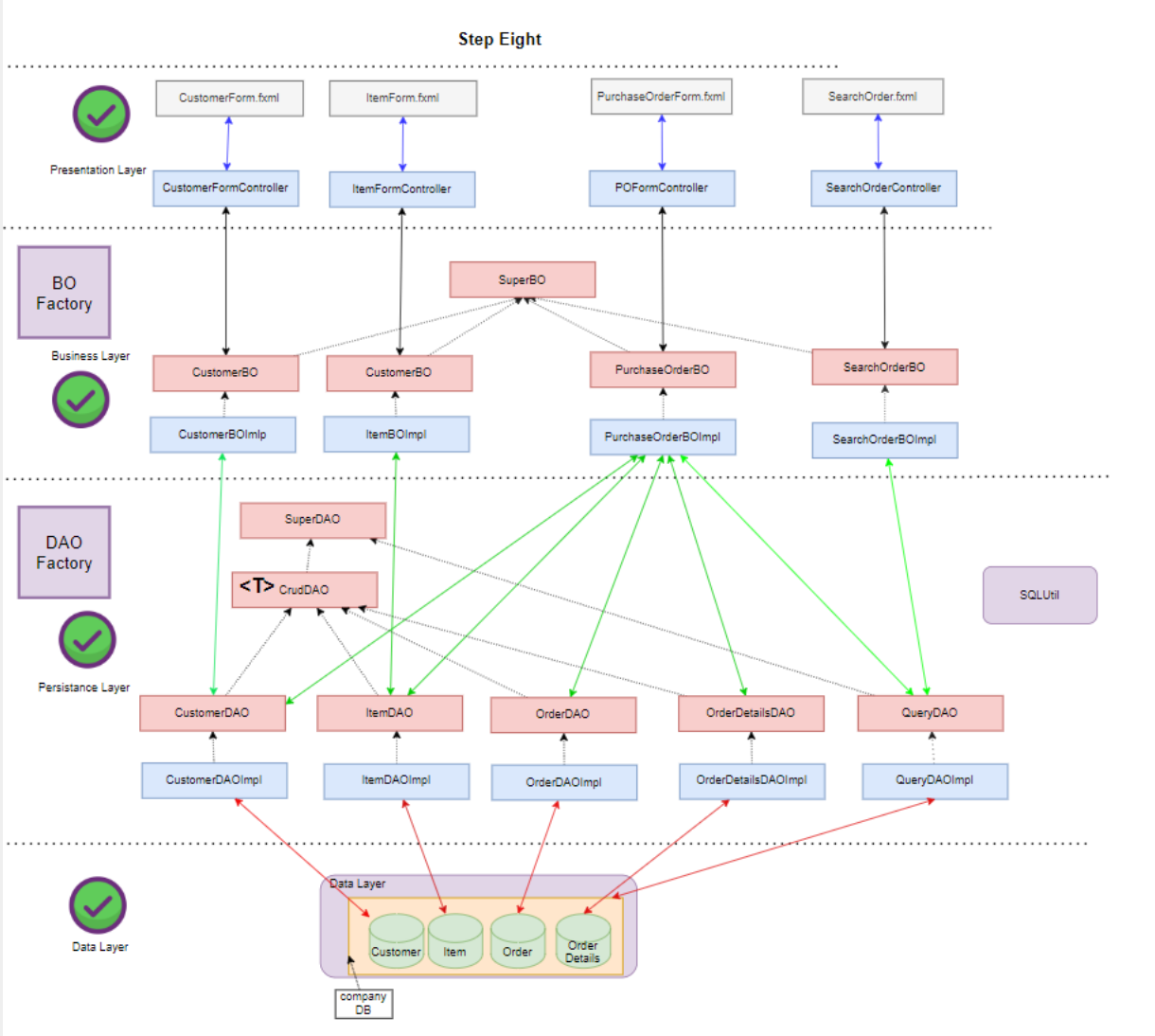
```java
public enum BOTypes {
    CUSTOMER, ITEM, PO
}

public SuperBO getBOType(BOTypes types) {
    switch (types) {
        case CUSTOMER:
            return new CustomerBOImpl();
        case ITEM:
            return new ItemBOImpl();
        case PO:
            return new PurchaseOrderBOImpl();
        default: return null;
    }
}
```

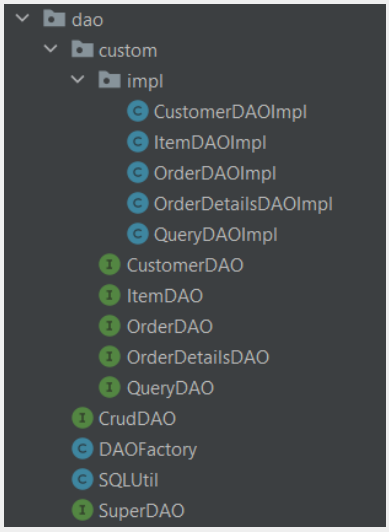
SuperBO interface එකක් හදල ඒක extend කරනෝ අනිත් BO interfaces වලට. Object වනුවට SuperBO return type එක use කරනෝ .(return කරන්නෙ object වර්ග 3ක් විතරක් නිසා)// most super class





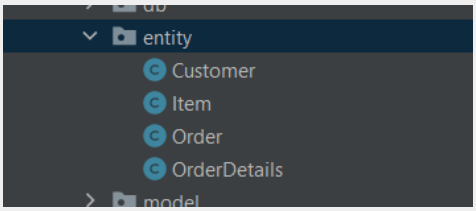
8. Fully Refactored DAO layer with the DAOFactory





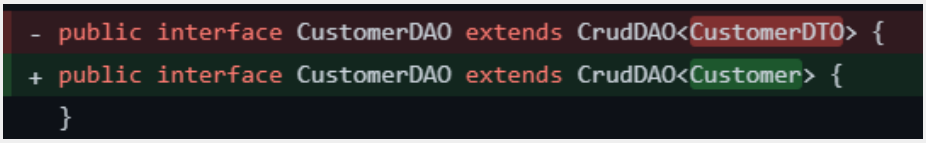
9.

- entity package added.



entity classes ▯ data layer එකේ තියෙන tables and properties ගැන code level එකේ idea එකක් ගන්න create කරන classes

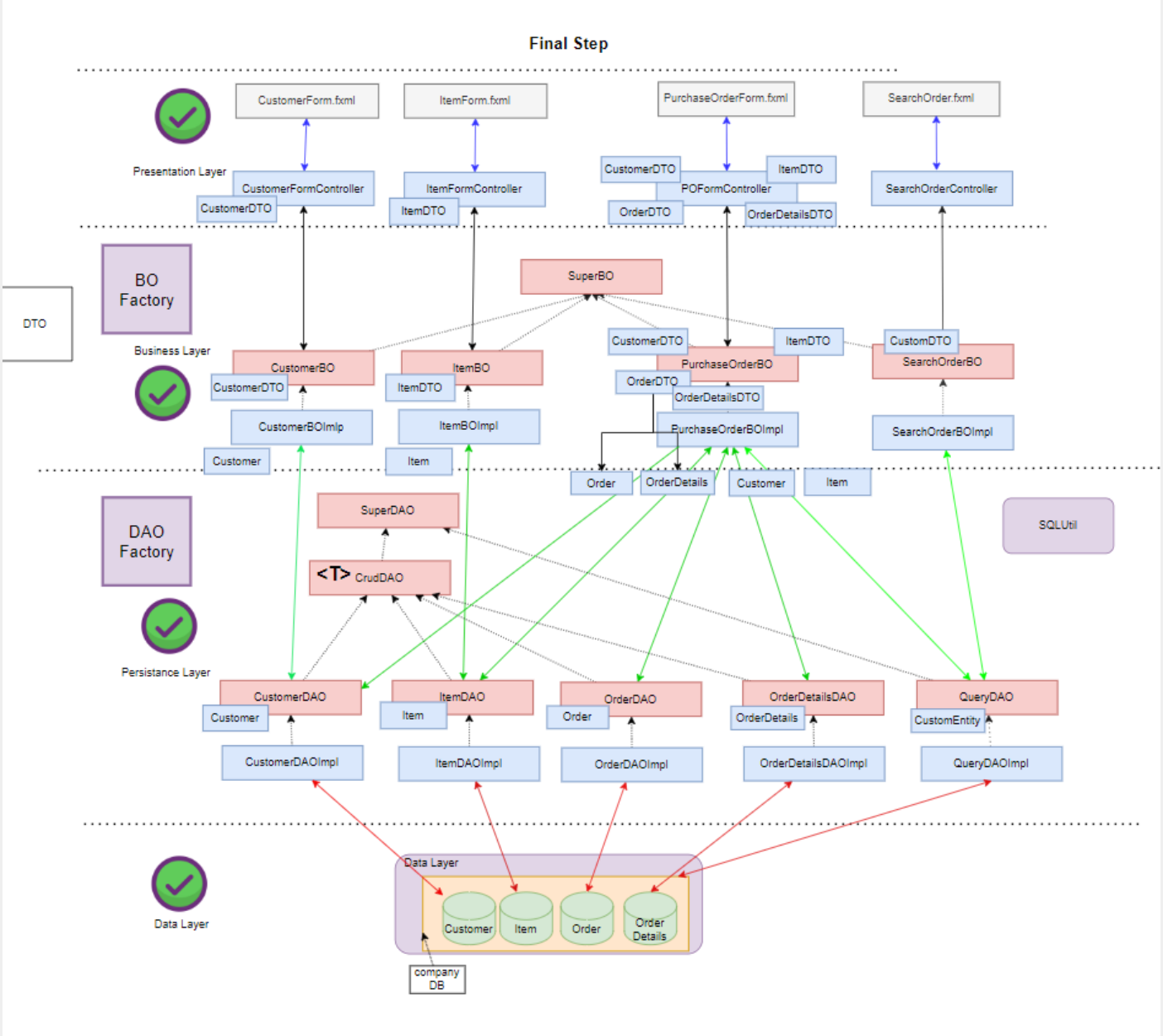
entity class එකක් = data layer එකේ table එකක්
class එකේ properties = table එකේ තියෙන Attributes
dao layer එක ඇතුලෙ entity class use කරන්නෙ.

- 

DAO layer එකෙ DTO class වෙනුවට entity class දානෙ.

- Join query වලට CustomEntity එක හැදුවා.

- DTO - Data Transfer Object
Usage - userගෙ requirement වල තියෙන data (UI එකේ data) අපිට ලේසි format එකකට business layer එක දක්වා transfer කිරීම.
(dto වලට restrictions නෑ, කැමති විදියටකට හදාගන්න පුලුවන්.)



Design Patterns

Design pattern ⍰ Expert developers ලා හොයාගනිපු **Common problem** එකකට තියනෙ **common solution** එකක්.

• Singleton design pattern

Used for ⍰ එකම object එක reuse වෙන එක නවත්තන්න
Used classes ⍰ DBConnection, DAOFactory, BOFactory

• Factory design pattern

Used for ⍰ object creation logic එක hide කරන්න
Used classes ⍰ DAOFactory, BOFactory

• Facade design pattern

Used for ⍰ Most common methods, එක තැනකට (interface එකකට) gather කිරීම. (for reusability)
Used interface ⍰ CrudDAO

• Strategy design pattern

Used for ⍰ runtime logic selection // unique method implement කරන්න.
Used interfaces ⍰ CustomerDAO, ItemDAO, OrderDAO, OrderDetailsDAO