|  |
| --- |
|  |
| UML DOCUMENTATION |
| OO DESIGN – CA2 |
|  |
| **Paul Millar** |
| **4/30/2014** |

|  |
| --- |
|  |

Contents

[Introduction 2](#_Toc386614397)

[Conceptual Class Diagram 3](#_Toc386614398)

[ErrorBase Class 3](#_Toc386614399)

[DAO Class 3](#_Toc386614400)

[IGenericDao Interface 3](#_Toc386614401)

[DaoFactory 4](#_Toc386614402)

[ModelFactory 4](#_Toc386614403)

[AbstractCommand Class (*abstract*) 4](#_Toc386614404)

[InsertCommand Class 4](#_Toc386614405)

[OrderDao Class 4](#_Toc386614406)

[Order Class 4](#_Toc386614407)

[UserDao Class 5](#_Toc386614408)

[User Class 5](#_Toc386614409)

[User Class In-Depth 5](#_Toc386614410)

[Overview 5](#_Toc386614411)

[Constructors 5](#_Toc386614412)

[Other methods 5](#_Toc386614413)

[Order Class In-Depth 6](#_Toc386614414)

[Overview 6](#_Toc386614415)

[Constructors 6](#_Toc386614416)

[Other Methods 6](#_Toc386614417)

[Login Sequence 8](#_Toc386614418)

# Introduction

The system that I have built revolves around a small business that sells, installs and provides services in relation to renewable energy products (solar panels, slates and wind turbines).

The system provides

* Information on various products & services
* Links to events related to solar products and climate change
* Latest local & international news bulletins related to solar products and climate change
* A contact page allowing customers to forward any requests.
* A gallery show casing previous installations.
* An administration area for Inserting, updating and deleting products, services, customers and orders.

To improve this site, we could fully mobilise it, provide the user with the option to enter a billing address as well as a shipping address. We could also provide a facility for users to sign up to a newsletter.

# Conceptual Class Diagram

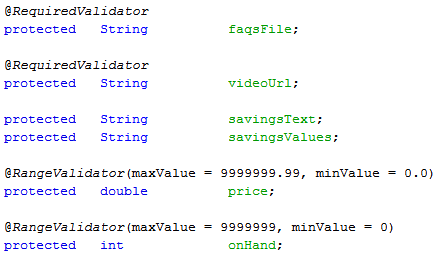
## ErrorBase Class

All DTO classes inherit from this class which facilitates our server side validation. It achieves this by way of annotations and reflection.

Two annotations are currently available …

* RequiredValidator
* RangeValidator

These are simply specified before each property (where necessary) in our DTO classes as follows…



Upon inserting or updating, we simply call the ***isValid()*** method to validate the object. This is achieved by utilising reflection to grab a list of all declared fields in the model and then iterating through these to check them.

## DAO Class

This class is responsible for creating our connection to the database and for freeing any resources such as connections, Prepared Statements and Result Sets. All DAO object inherit from this class.

## IGenericDao Interface

All DAO interfaces extend this interface. It provides 5 basic methods used by all DAO’s.

* Insert
* Update
* Delete
* getAll
* getKey

The advantage of this comes into play with our command classes. Rather than creating a seperate insert command for all of six tables, we can now create a single one to facilitate all of them. The same applies to updating, deleting, getting all records and getting a single record by key. So rather than creating 30 different classes (6 DAO’s \* 5 commands), our system will now run with only 5 commands to achieve the same goals.

## DaoFactory

This class implements the Factory & Singleton Patterns and returns an object of Type - IGenericDao (OrderDao, UserDao, etc..).

## ModelFactory

This class implements the Factory & Singleton Patterns and returns a DTO object (Order, User, etc..). This can return either a brand new DTO object or read parameters from a Request object in order to create the appropriate object prior to inserting, updating or deleting.

## AbstractCommand Class (*abstract*)

This is the super class for all Command classes and is dependent on the ModelFactory and DaoFactory classes. It is responsible for retrieving the correct DAO and DTO objects for interacting with the database.

## InsertCommand Class

This class inherits from the AbstractCommand class and is responsible for inserting information into or database.

## OrderDao Class

This DAO class is responsible for updating the orders table in our database. It extends the DAO class in order to get the appropriate connection object and implements the IOrderDao interface which provides it with a set of methods that it must implement. It utilises the Order class for storing information in memory when retrieving records from the database and as a source of information when writing to the database.

## Order Class

This DTO relates to the Order table in our database and is used for creating objects for each row of information in that table.

## UserDao Class

This DAO class is responsible for updating the User table in our database. It extends the DAO class in order to get the appropriate connection object and implements the IUserDao interface which provides it with a set of methods that it must implement. It utilises the User class for storing information in memory when retrieving records from the database and as a source of information when writing to the database.

## User Class

This DTO relates to the User table in our database and is used for creating objects for each row of information in that table.

# User Class In-Depth

Extends: ErrorBase

Implements: Serializable, Comparable

## Overview

This table represents the User table in our database and is a dependency of the UserDao when updating the database. It is also a dependency of the Order DTO class for relating a user to their orders.

## Constructors

This class contains several constructors for performance and security reasons in relation to transferring data across a network. For example, when we want a list of all users, we only need a few fields to display in a list that identifies each user. However when we want to change a user’s details then we would need to retrieve all of their details. Again, when displaying an order for a user, we don’t need to know their password or when their last details were last updated so this information can be excluded. Initially when we create an order, we only need a User object with their unique id. When the order is inserted into the database, we then retrieve the order with the necessary user details.

## Other methods

This class employs all the usual getters and setters as well as providing overrides for *equals, hashCode* and *toString*. It also provides the compareTo method which *can be used to sort* users based on their last name.

The getFullName method returns a string that concatenates the first and last name.

The getFullAddress method returns a string that concatenates all the address properties.

# Order Class In-Depth

Extends: ErrorBase

Implements: Serializable

## Overview

This table represents the Order table in our database and is a dependency of the OrderDao when updating the database. It is dependent on the User DTO class and the OrderItem DTO class. It utilises a HashMap for storing multiple order items and enforces referential integrity for this. The key used for this hashmap is not the order item id, but rather the product id, this way we can check for duplicates prior to inserting a new order item. If this product already exists, we simply increase the quantity of the order item.

## Constructors

This class also contains several constructors for performance reasons in relation to transferring data across a network. When creating a new order, we actually only need the id of a user, all other values are created by the database. We then have a second constructor that is used for all other requirements.

## Other Methods

**getOrderStatusOptions()**

Provides a hashmap of order status options, e.g. ‘On Order’, ‘Work in Progress’, etc..

**getOrderStatus()**

Returns the current state of the order.

**getLastUpdatedFormatted()**

Returns the last updated date of the order formatted to "dd/MM/yyyy HH:mm:ss".

**getOrderDateFormatted()**

Returns the last updated date of the order formatted to "dd/MM/yyyy HH:mm:ss".

**getOrderItems()**

Returns a collection of related order items.

**addOrderItem (OrderItem orderitem)**

Adds an order item to the order.

**changeOrderItem(OrderItem orderitem)**

Changes the quantity or price of an order item.

**removeOrderItem(int key)**

Removes an order item specified by the product number.

**removeOrderItem(OrderItem oi)**

Removes an order item as specified by the object.

**getExistingOrderItemProduct(int key)**

Returns an order item as specified by the product key.

# Login Sequence

After submitting their email and password, the ‘processRequest()’ within the ‘MainServlet’ class is called. The sequence of actions is as follows …

1. The value of the request parameter ‘cmd’ is assigned to the string ‘command’
2. We get an instance of the CommandFactory
3. Pass in the ‘command’, request & response values into the createCommand method, which then returns back the appropriate command.
4. The ‘execute()’ method of this command is then called which performs the actual action as follows ...
   1. Get the ‘email’ & ‘password’ parameters from the request object.
   2. Get an instance of the UserDao object.
   3. Check that values have been specified for both request parameters.
   4. Call the login() method of the UserDao object which should return a User object.
   5. If the User object is not null, we get their user id, user type and first name and set these in the session object. We then log this successful action and set the value of the ‘forwardPath’ variable to the ‘index.jsp’ page.
   6. If not, we then log this failed action and set the value of the ‘forwardPath’ variable to the ‘login.jsp’ page.
   7. The value of the ‘forwardPath’ variable is then returned to the MainServlet.
5. If the action was performed successfully, the ‘execute()’ method will return the url to forward to.
6. The user is forwarded to the appropriate page.

# Search All Sequence

After requesting to view all products, services or any other table data, the ‘processRequest()’ within the ‘MainServlet’ class is called. The sequence of actions is as follows …

1. The value of the request parameter ‘cmd’ is assigned to the string ‘command’
2. We get an instance of the CommandFactory
3. Pass in the ‘command’, request & response values into the createCommand method, which then returns back the appropriate command.
4. The ‘execute()’ method of this command is then called which performs the actual action as follows ...
   1. We make a call to the getDao() method which gives us the appropriate DAO object.
   2. We then call the getAll() method of this DAO which returns a collection of all data from a table. (Remember, all Dao’s implement IGenericDao interface so they all have this method by contract)
   3. If the user has requested to view categories, products or services, thn we also need to make a call to either the loadCategories() or loadServices() methods of the DataAccessService class. These return collections that are stored in session variables and used for the side bars on those pages.
   4. The value of the ‘forwardPath’ variable is then returned to the MainServlet.
5. If the action was performed successfully, the ‘execute()’ method will return the url to forward to.
6. The user is forwarded to the appropriate page.