**Implementation**

Following the design process, but not separated from it, comes the implementation. This is the part where all the knowledge obtained in the previous two sections is used to create the final product. A clear understanding of Entafarma’s business situation together with all the decisions reached during system development contributed to a fluid development process.

**Code Standard**

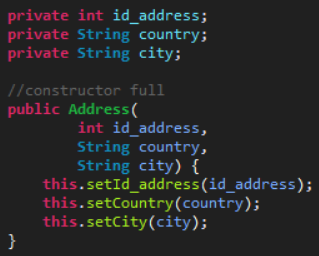
The software product has been developed inside Eclipse IDE using the Java programming language. The server-side consists of a RDMS running on a Microsoft SQL Server. The sqljdbc4.jar class library provided by the Microsoft JDBC Driver for SQL Server is used to enable communication between the server and the client.

In order to reduce the cost of future code improvements and software maintenance, the Code Conventions for the Java Programming Language [http://www.oracle.com/technetwork/java/codeconvtoc-136057.html] recommended by Oracle have been followed. In addition to this, a number of in-house coding conventions have been established to improve the workflow inside the team.

**Code Example**

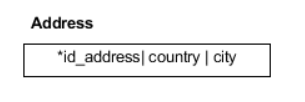
Inside this section, the Address table defined in the Relational Model during the design process will be used to provide an understanding blueprint for the entire software. Following the journey of this table from design to displaying data to the end-user is guaranteed to offer a clear overview inside the code of the software and the reasoning behind it.

1. modelLayer

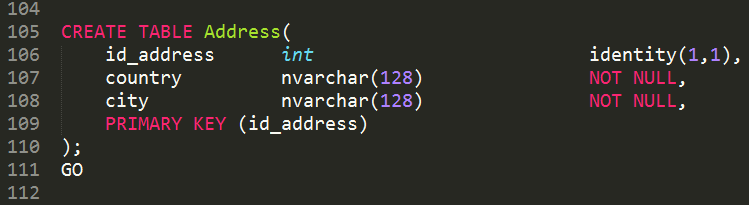


Above, part of the Address class can be seen. It closely follows the database structure. In addition to the full constructor, constructors that don’t take an id\_address, that take just an id\_address and empty constructors are used. Even though some code duplication is present due to this approach, it helps reduce the amount of code in classes that use the modelLayer.Address.

1. dbLayer



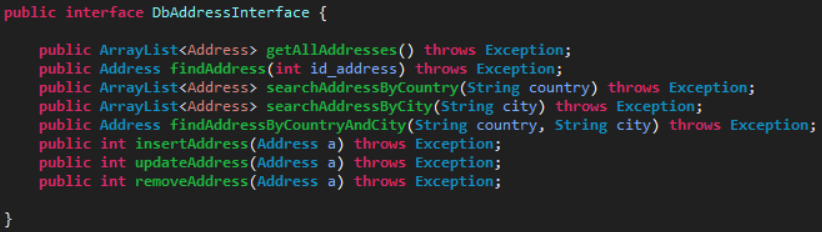
The above image is the Address table, as represented in the Relational Model. To create the table inside the database, the following query is used:



The id\_address is of type int, is auto-incremented and represents the PRIMARY KEY, as defined by identity(1,1).

The country and city columns are both of type nvarchar(128). Also, they do not store NULL values, as every Address tuple is defined by a country and a city.

The type nvarchar has been chosen over varchar because, as explained in the Business part, Entafarma’s customers come from different countries and they use different languages. Varchar is restricted to an 8-bit codepage while nvarchar can store any Unicode data. Even thought it takes more space, the benefit of being able to store any Unicode character is more important for Entafarma than saving disk space.



The above interface shows all the functionality implemented by the dbLayer.DbAddress class. Interfaces are used for all the classes in the dbLayer to allow changes in the implementation without affecting other parts of the software.

The current implementation consist of 4 private functions: String buildQuery(String), Address buildAddress(ResultSet), Address singleWhere(String, boolean) and ArrayList<Address> miscWhere(String, boolean). The implementations of the other dbLayer classes are similar, though some of them have additional private functions.

The queries sent to the database are constructed using prepared statements. This lowers the risk of SQL injection attacks and makes the code easier to read and understand by humans.

Exceptions are not handled at this level. Instead, context is added to them and then they are rethrown. Handling exceptions in the controlLayer makes more sense. There, it is easier to decide what to do with them and how the execution flow should be affected, based on the context in which the exception ocured.

1. controlLayer

The class controlLayer.CtrAddress provides the system with all the functionality needed for handling addresses. The functions are clearly explained in the comments, together with their expected return values and their meanings.



Above is the insertAddress(String, String) function. From the comments it is understood that it will return 1 if successful or a negative value: Errors.INSERT\_ADDRESS if unsuccessful. Only in the case when somthing terribly wrong happens will it throw an exception: the first try fails and the rollbackTransacion() also fails.

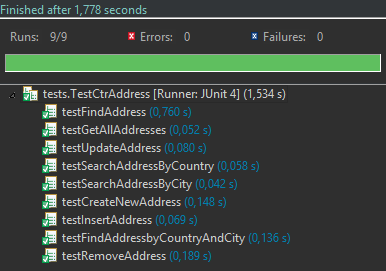
For error codes the Errors enum is used. Every error has it’s own code(a negative value) and message. The enum approach has been considered the prefered approach because it restricts the input paramether, it is a constant declaration and is checked at compile time, making it also typesafe [https://docs.oracle.com/javase/8/docs/technotes/guides/language/enums.html].

1. guiLayer

//adrian

1. JUnitTest

The end of the journey is marked by a JUnitTest. The class tests.TestCtrAddress implements testing methods for all the functionality defined in the controlLayer. The results of the test can be seen below.



**Code Highlights**

1. TRANSACTION\_SERIALIZABLE

As understood from the business analysis, the developed software solution will be used by more than one Entafarma employee at the same time. Because all the instances will work simultaneously with the same database, there is a high probability for Dirty Reads and Non-Repeatable Reads to happen.

To prvent this issue, it has been decided that the software should use transactions to preserve data integrity. By setting the Isolation Level to TRANSACTION\_REPEATABLE\_READ, both Dirty Reads and Non-Repeatable Reads are prevented.

At a later stage in development, it has been noticed that in some rare cases a Phantom Read might happen. After observing this issue, the Isolation Level was raised to the max level: TRANSACTION\_SERIALIZABLE.The above implementation prevents all three possible issues [https://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html].



1. Quadratic Time Complexity

As seen in the business case, Entafarma automatically applies a discount to its customers whenever they place a new sale. The discount is calculated based on the total money spent by the customer with the company until the time of the sale.

In order to retrieve the total amount of money spent by a customer, the system has to get all previous sales made by the customers and loop through them to get the sale price. But every sale contains an ArrayList of sale lines, so in order to get the sale price the system has to loop through the sale lines.

The time complexity of this algorithm is O(n2). The most relevant parts can be seen below.

