Relational databases group process log

DATA PROCESSING
SEMESTER 4 2020/2021

Lex Van Dalen,
Slawomir Twardowski

Docent Jeroen De Kort

INTRODUCTION

In this document we archive the iteration process of our group assignment project during Semester 4 concerning relational databases.

This project includes designing a database on the basis of delivered data files and requirements described by the scenario; developing software required to clean-up and copy given data into the newly designed database; and finally visualizing the data in a dashboard, made with software package chosen by the group.

On the following pages you'll find screenshots and descriptions illustrating the process and choices made with explanation of motivations behind our choices.

CONTENTS

DESIGN

After supplied files and Domino webpage analysis (which is documented in a separate document concerning analysis of the project) we begin by drawing first versions of DB diagrams. For this purpose, we decided to use DRAW.IO. Our choice here was motivated by the simplicity and availability of the tool and it's version control possibilities (easy and free access to GitHub). In this case the simpler the better since being able to quickly iterate and share the changes between group members was key.

1. Illustrating the data within the supplied Mario Data files

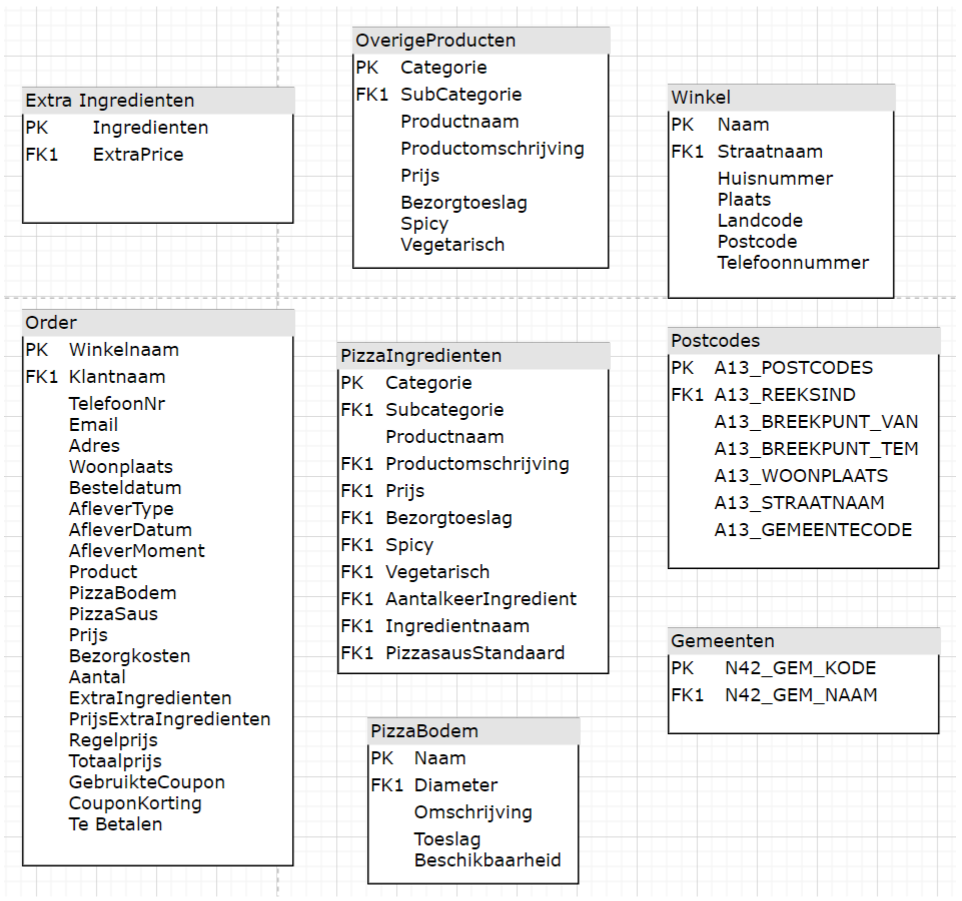
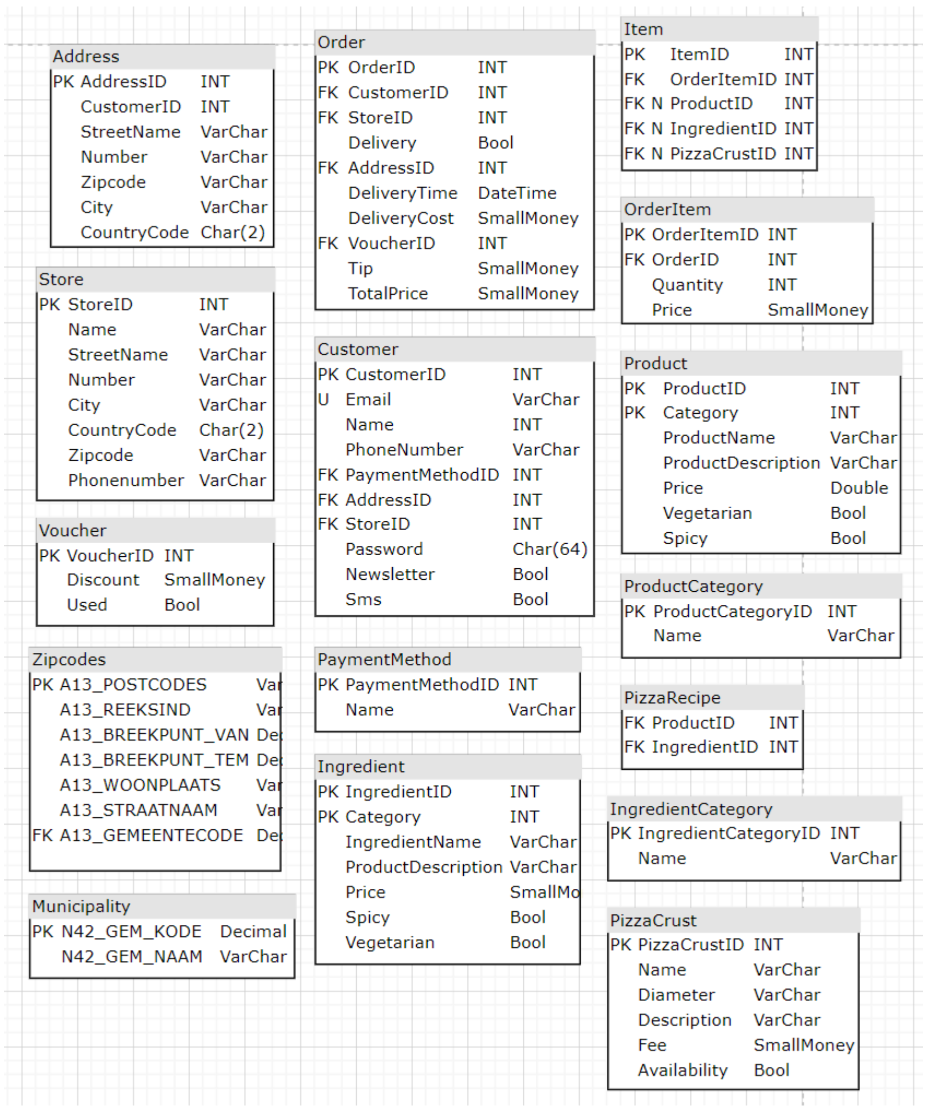


Fig. 1- Diagram mirroring existing files

Our first commit is just a simple illustration of the tables supplied within the Mario Data files. From this point on we can think of arranging the data in a way that would best suit the requirements that came forth from the analysis (PK/FK are not properly applied and should be ignored at this stage, since the data will be completely re-arranged anyways)

2. Re-arranging the data and deciding on data-types



Address

PK	AddressID	INT
	CustomerID	INT
	StreetName	VarChar
	Number	VarChar
	Zipcode	VarChar
	City	VarChar
	CountryCode	Char(2)

Store

PK	StoreID	INT
	Name	VarChar
	StreetName	VarChar
	Number	VarChar
	City	VarChar
	CountryCode	Char(2)
	Zipcode	VarChar
	Phonenumber	VarChar

Voucher

PK	VoucherID	INT
	Discount	SmallMoney
	Used	Bool

Zipcodes

PK	A13_POSTCODES	Var
	A13_REEKSIND	Var
	A13_BREEKPUNT_VAN	De
	A13_BREEKPUNT_TEM	De
	A13_WOONPLAATS	Var
	A13_STRAATNAAM	Var
FK	A13_GEMEENTECODE	De

Municipality

PK	N42_GEM_KODE	Decimal
	N42_GEM_NAAM	VarChar

Order

PK	OrderID	INT
FK	CustomerID	INT
FK	StoreID	INT
	Delivery	Bool
FK	AddressID	INT
	DeliveryTime	DateTime
	DeliveryCost	SmallMoney
FK	VoucherID	INT
	Tip	SmallMoney
	TotalPrice	SmallMoney

Customer

PK	CustomerID	INT
U	Email	VarChar
	Name	INT
	PhoneNumber	VarChar
FK	PaymentMethodID	INT
FK	AddressID	INT
FK	StoreID	INT
	Password	Char(64)
	Newsletter	Bool
	Sms	Bool

PaymentMethod

PK	PaymentMethodID	INT
	Name	VarChar

Ingredient

PK	IngredientID	INT
PK	Category	INT
	IngredientName	VarChar
	ProductDescription	VarChar
	Price	SmallMo
	Spicy	Bool
	Vegetarian	Bool

Item

PK	ItemID	INT
FK	OrderItemID	INT
FK N	ProductID	INT
FK N	IngredientID	INT
FK N	PizzaCrustID	INT

OrderItem

PK	OrderItemID	INT
FK	OrderID	INT
	Quantity	INT
	Price	SmallMoney

Product

PK	ProductID	INT
PK	Category	INT
	ProductName	VarChar
	ProductDescription	VarChar
	Price	Double
	Vegetarian	Bool
	Spicy	Bool

ProductCategory

PK	ProductCategoryID	INT
	Name	VarChar

PizzaRecipe

FK	ProductID	INT
FK	IngredientID	INT

IngredientCategory

PK	IngredientCategoryID	INT
	Name	VarChar

PizzaCrust

PK	PizzaCrustID	INT
	Name	VarChar
	Diameter	VarChar
	Description	VarChar
	Fee	SmallMoney
	Availability	Bool

Fig. 2 - Data re-arrangement

The following step in the project is re-arranging the data into new tables of our own design and deciding upon the data types used in the DB. We have also set Primary and Foreign Keys.

We intend to use SmallMoney type (SQL Server) where money amounts are concerned, INT type for PK/FK, Date or Time for fields concerning time and VARCHAR of varying length to accommodate names, descriptions and any fields where numbers are mixed with alphabet.

3. Tables connections

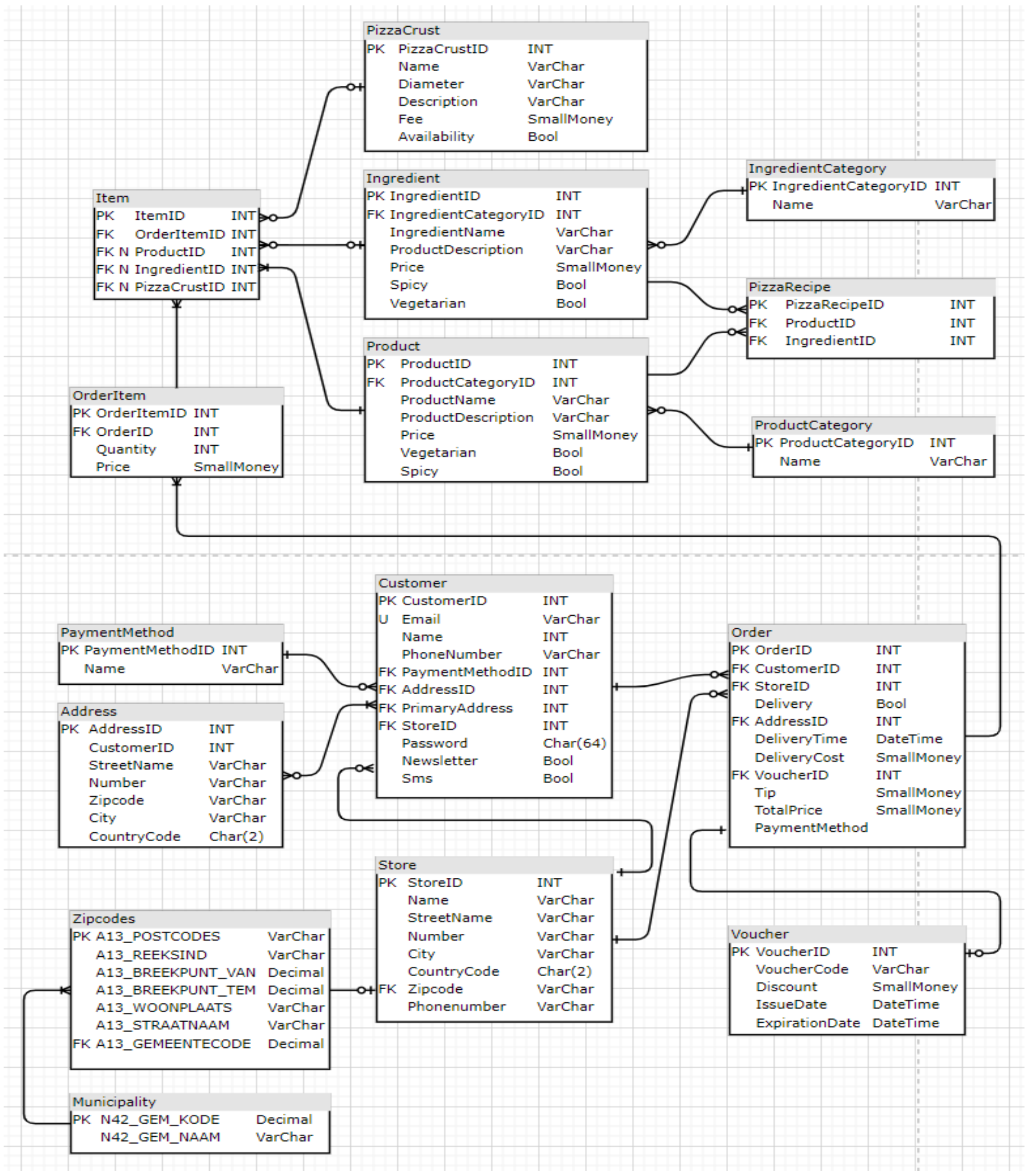


Fig. 3 - Connecting the tables

Connecting the tables with one another is the crucial thing. It's the deciding factor of the DB flexibility. We went for an ID Primary Key approach which is then mirrored as Foreign Key in other tables to make the connection. That way we couple tables together and don't have to rely on comparing data types other than simple Integer types. This makes using coupling tables more consistent with the overall design.

4. Final iteration

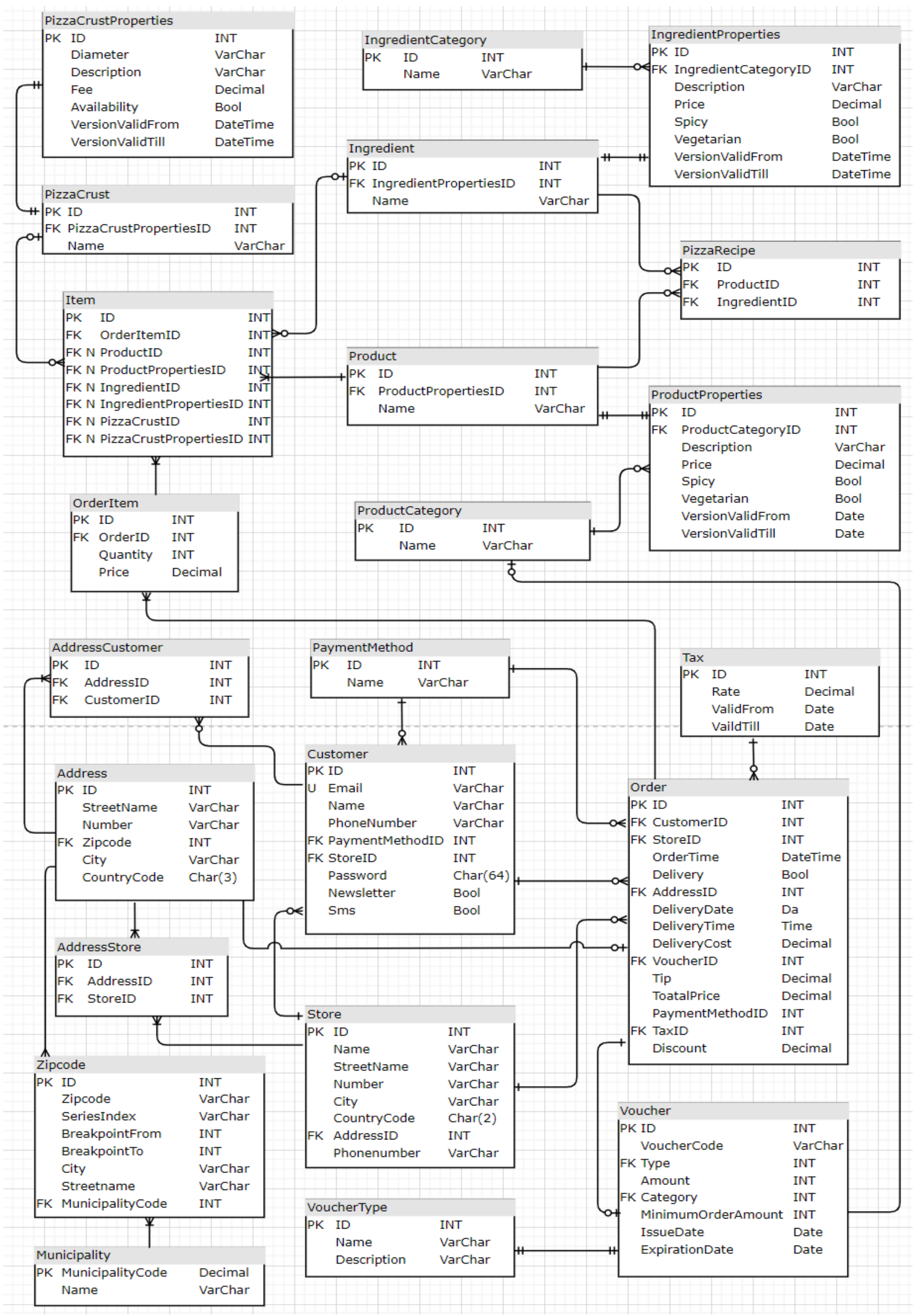


Fig. 4 - Final diagram

(description on p.9)

Description concerning Fig. 4 from p.8

SmallMoney type has been replaced by Decimal. This is due to the imprecision and possible calculation errors using this type of data which we found after researching the topic. We've decided to use following naming convention for best readability: "ID" name alone as PK; and FK in a "*SourceTable*ID" format. We use coupling tables to allow things like multiple Items in an Order or multiple addresses for customers and stores. This gives flexibility and future proofs the database making adding multiple addresses for customer possible. Properties tables store product qualities like descriptions, whether product is spicy or vegetarian, and whether product is valid or not. That way we can store products bought in the past in their original form and can accommodate things like product name changes without affecting archived orders. We have split the pizza ingredient's into it's own table, but included pizza's in the Product table. We think this gives better overview and fits the business requirements better. Pizza's are the core of the business, but at the same time they are a product just as any other. To discern between the variety of available products we use a Category table, same goes for Ingredient Category which discerns between ingredients on top of the pizza or sauces.
Tax table has been included to facilitate proper tax handling, which also includes ValidFrom/ValitTill fields to be able to specify time period in which a given tax rate has been applicable.
We think this data base design leaves room for future growth and fulfills the requirements set by existing MarioPizza business.

5.Helper diagram with data source:target routing

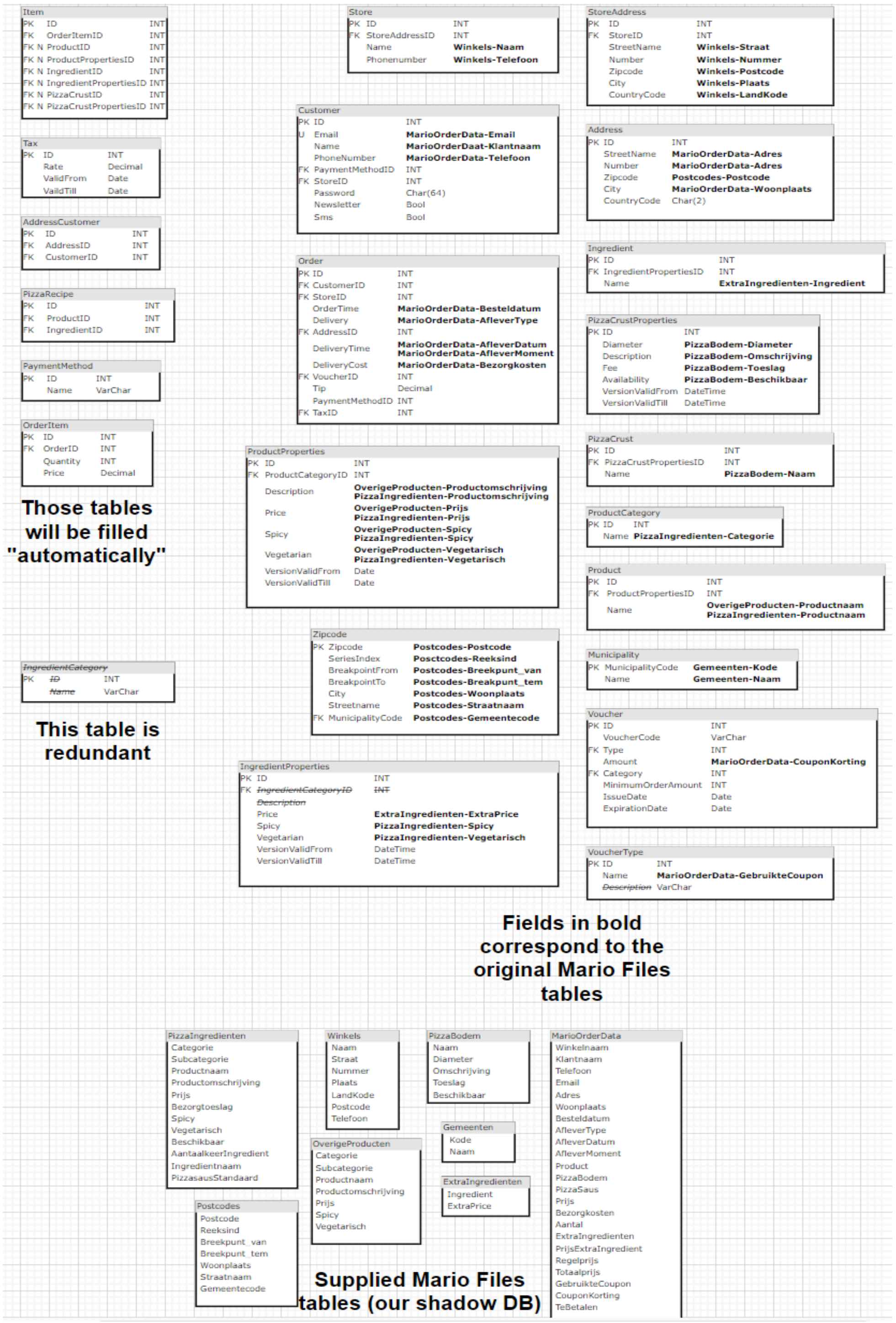


Fig. 5 - Helper diagram

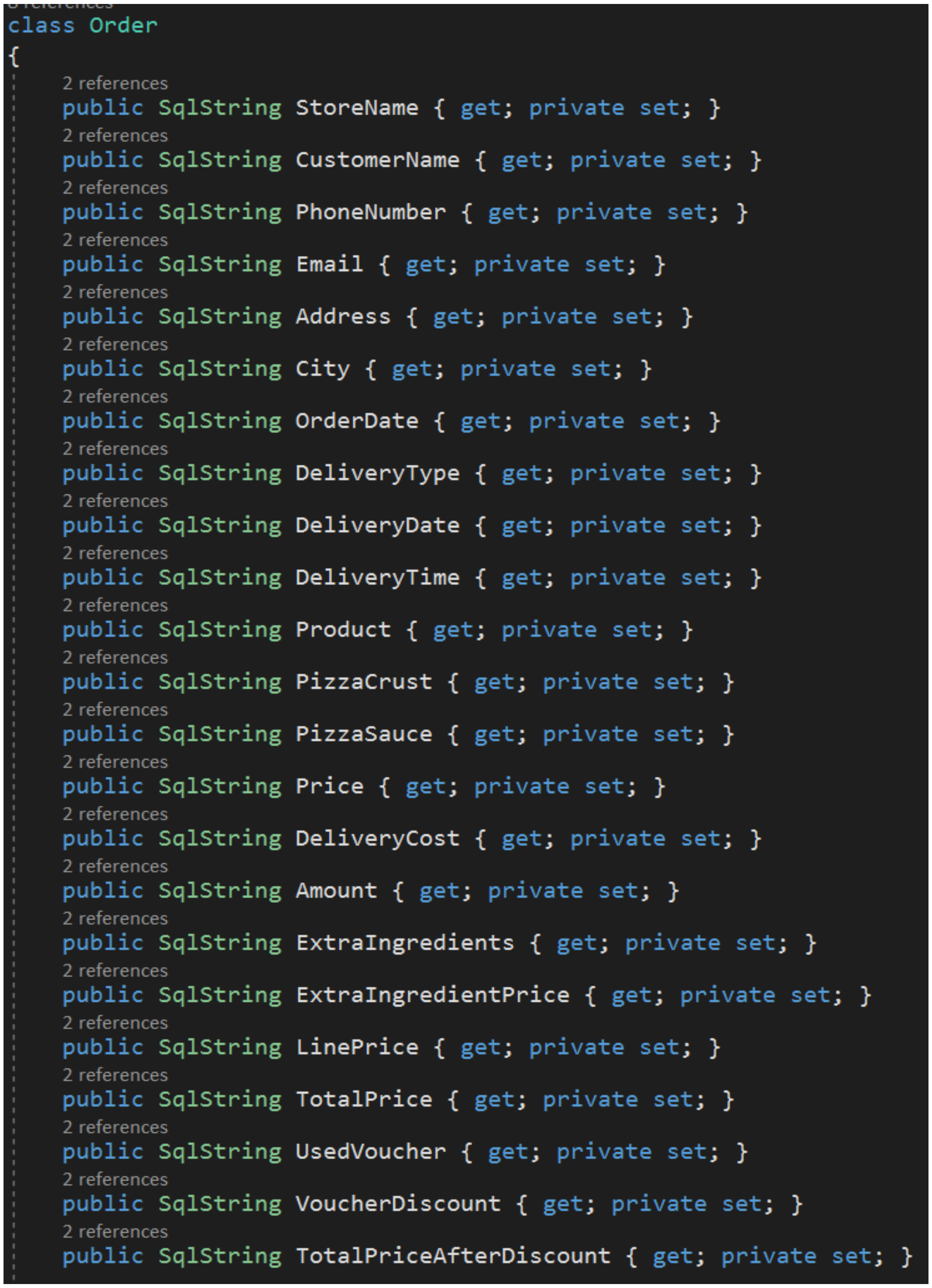
Helper diagram used to determine which data goes where. We use it to illustrate the routing of the data that'll need to be transferred in our Stored Procedures queries. It's just a simple overview to keep trace of the data sources.

IMPLEMENTATION

For implementation we settled on Microsoft platform as it gives a coherent package to tackle the presented scenario. We use C# programming language to develop the software needed for cleaning-up and importing the supplied data as we are familiar with Java and C# seems similar in syntax. It has built in packages to connect to SQL Server which is our database of choice. Both C# and SQL Server have very good documentation and since they are very popular solutions there's a lot of instruction material available online.
Visual Studio IDE which we'll use for development, as well as SQL Server Management Studio are free for non-commercial use which makes it a perfect choice for the purpose of this project. We realize that for commercial use there are drawbacks too – like the pricing or the fact that SQL Server doesn't have an integrated version control capability (those are available from 3rd parties for additional charges).

Our approach to importing the data is to first copy all supplied data into a shadow DB as VARCHAR type. Since the shadow data base is just a temporary one, we are not concerned with any performance considerations at this point. During this process data will be cleaned by our software to get rid of any unexpected symbols, so that it's ready for further converting process into expected data types during the final import into the target Mario data base. Final import will be realized using Stored Procedures.
It seems to us that this approach offers us a good base and leaves room for any improvements/adjustments that might turn out necessary during the whole process. The use of Stored Procedures will hopefully allow us to limit our workload at further stages to SQL Server Management Studio and will make introducing changes more manageable.

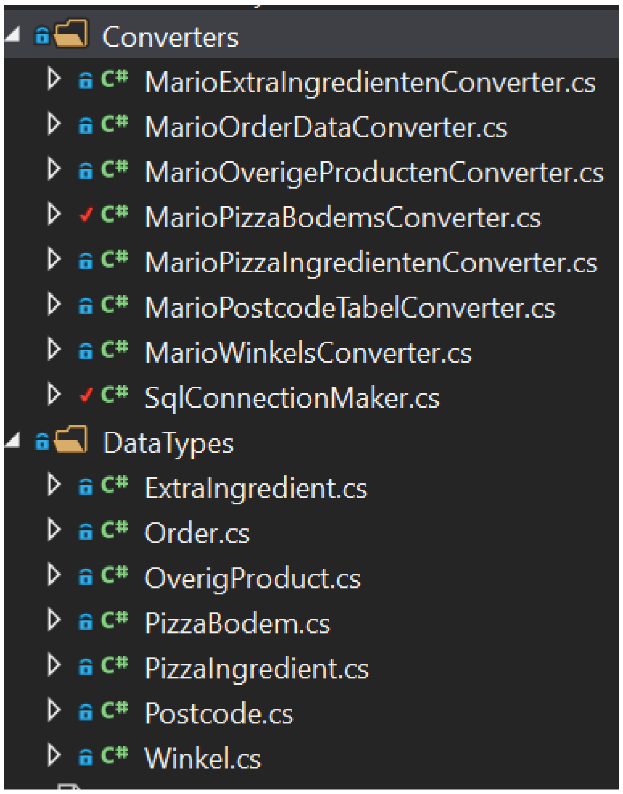
1. C# Mario Data Conversion Tool



```
class Order
{
    2 references
    public SqlString StoreName { get; private set; }
    2 references
    public SqlString CustomerName { get; private set; }
    2 references
    public SqlString PhoneNumber { get; private set; }
    2 references
    public SqlString Email { get; private set; }
    2 references
    public SqlString Address { get; private set; }
    2 references
    public SqlString City { get; private set; }
    2 references
    public SqlString OrderDate { get; private set; }
    2 references
    public SqlString DeliveryType { get; private set; }
    2 references
    public SqlString DeliveryDate { get; private set; }
    2 references
    public SqlString DeliveryTime { get; private set; }
    2 references
    public SqlString Product { get; private set; }
    2 references
    public SqlString PizzaCrust { get; private set; }
    2 references
    public SqlString PizzaSauce { get; private set; }
    2 references
    public SqlString Price { get; private set; }
    2 references
    public SqlString DeliveryCost { get; private set; }
    2 references
    public SqlString Amount { get; private set; }
    2 references
    public SqlString ExtraIngredients { get; private set; }
    2 references
    public SqlString ExtraIngredientPrice { get; private set; }
    2 references
    public SqlString LinePrice { get; private set; }
    2 references
    public SqlString TotalPrice { get; private set; }
    2 references
    public SqlString UsedVoucher { get; private set; }
    2 references
    public SqlString VoucherDiscount { get; private set; }
    2 references
    public SqlString TotalPriceAfterDiscount { get; private set; }
```

Fig. 6 - C# class mirroring supplied data (example)

We start by modelling the data supplied within the files. Each Class mirrors a table from the files ignoring data types.

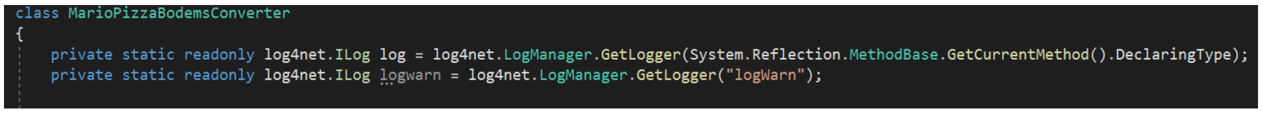


Converters
- MarioExtraIngredientenConverter.cs
- MarioOrderDataConverter.cs
- MarioOverigeProductenConverter.cs
- MarioPizzaBodemsConverter.cs
- MarioPizzaIngredientenConverter.cs
- MarioPostcodeTabelConverter.cs
- MarioWinkelsConverter.cs
- SqlConnectionMaker.cs

DataTypes
- ExtraIngredient.cs
- Order.cs
- OverigProduct.cs
- PizzaBodem.cs
- PizzaIngredient.cs
- Postcode.cs
- Winkel.cs

Fig. 7 - C# Classes overview

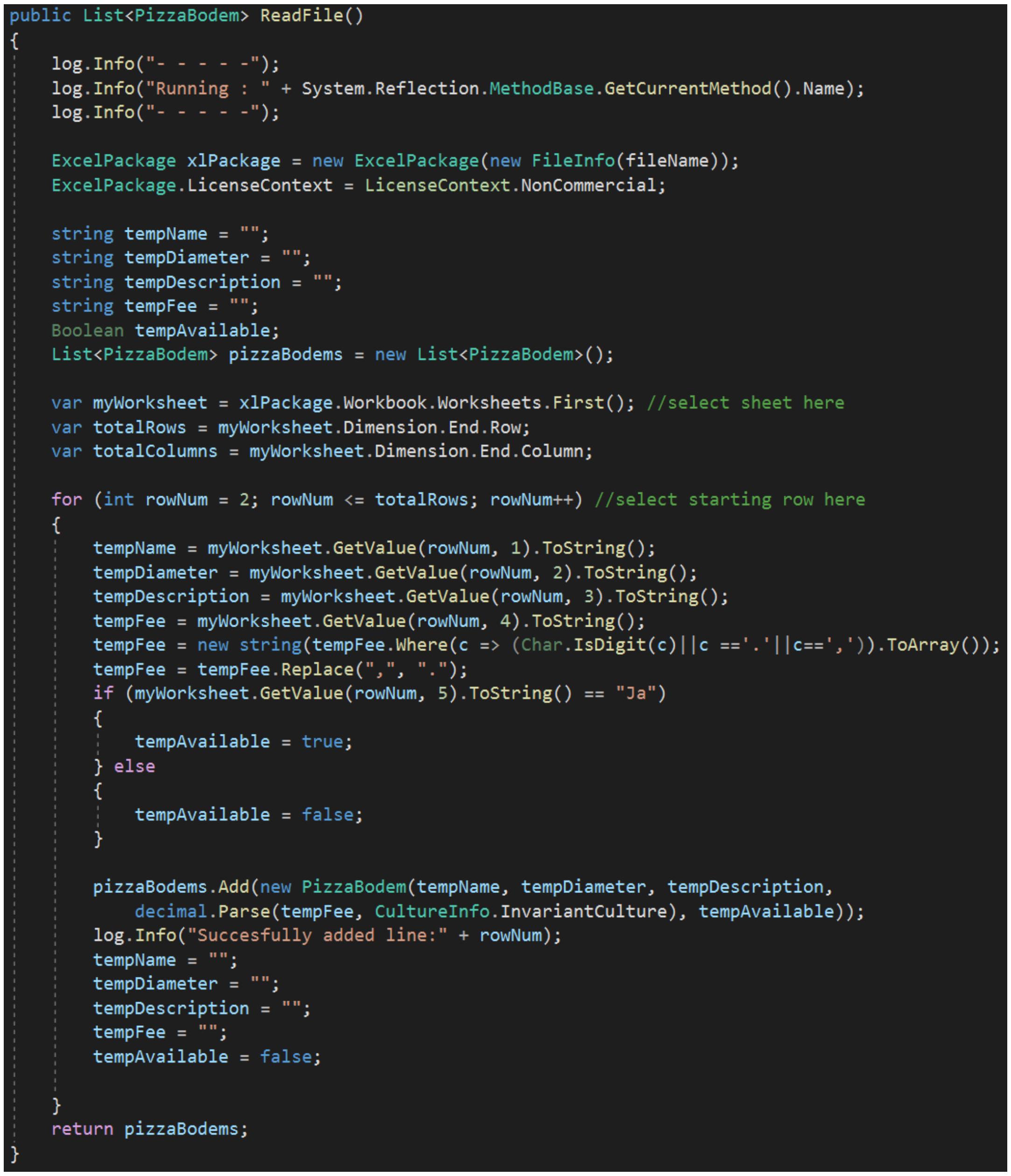
Each of the supplied files is being mirrored and each of them gets it's own Converter class which takes care of data cleaning and uploading into the shadow data base. The methods within each class are similar in structure. We will present one example from the PizzaBodems class to illustrate the approach we've taken.



```
class MarioPizzaBodemsConverter
{
    private static readonly log4net.ILog log = log4net.LogManager.GetLogger(System.Reflection.MethodBase.GetCurrentMethod().DeclaringType);
    private static readonly log4net.ILog logwarn = log4net.LogManager.GetLogger("logWarn");
```

Fig. 8 - C# Logger

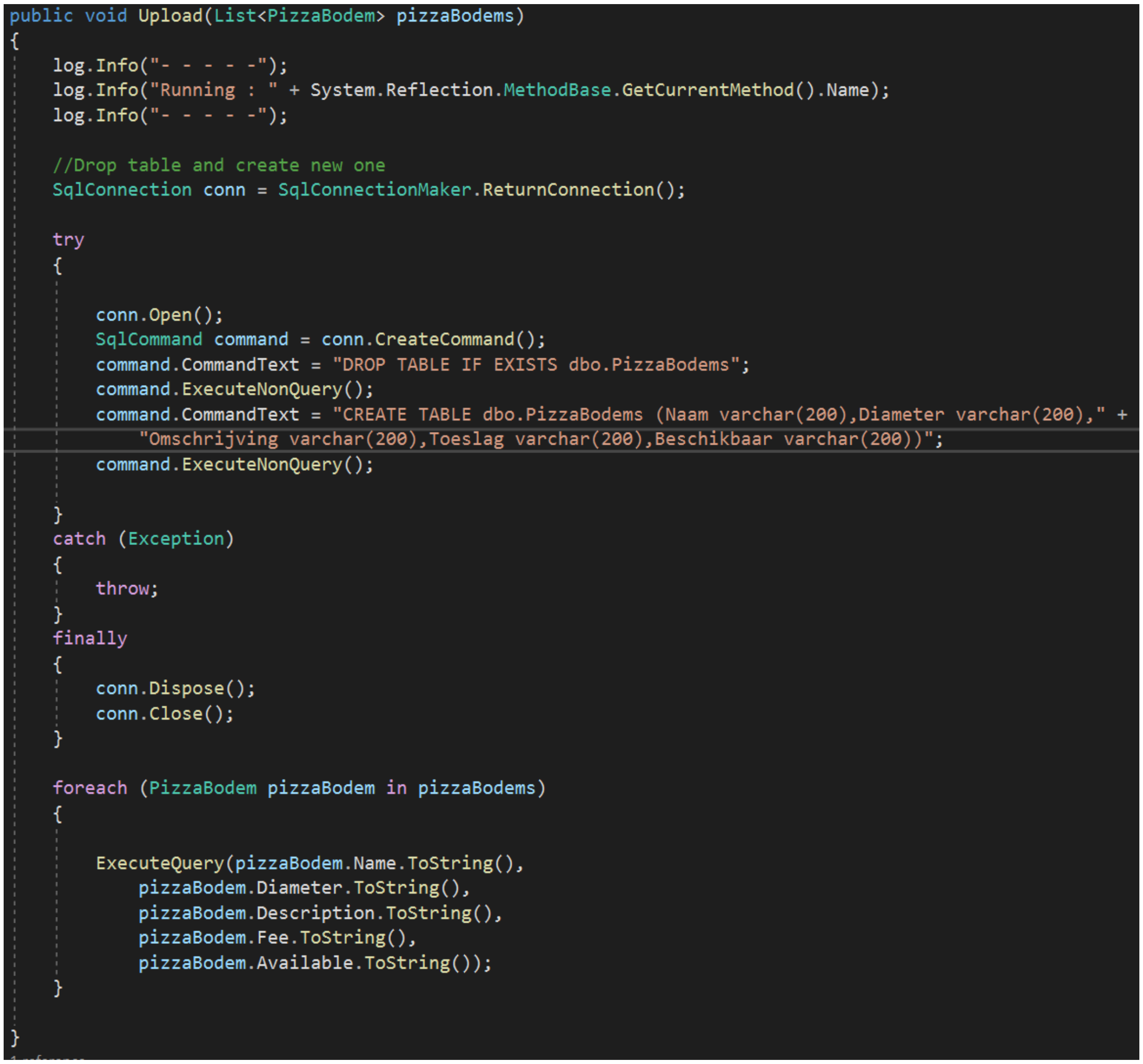
Progress of each process is being logged by a logger. That way we can keep track of errors during the import process. All errors are being saved into a file.
We decided to use *log4net* as our logging library as it is open source and offers all the functionality we could want. It has a reputation of being fast and reliable while remaining very flexible. There's a variety of methods of logging errors and events in your software and it can be configured at runtime. It is based on Java library *log4j*.



```
public List<PizzaBodem> ReadFile()
{
    log.Info("- - - - -");
    log.Info("Running : " + System.Reflection.MethodBase.GetCurrentMethod().Name);
    log.Info("- - - - -");

    ExcelPackage xlPackage = new ExcelPackage(new FileInfo(fileName));
    ExcelPackage.LicenseContext = LicenseContext.NonCommercial;

    string tempName = "";
    string tempDiameter = "";
    string tempDescription = "";
    string tempFee = "";
    Boolean tempAvailable;
    List<PizzaBodem> pizzaBodems = new List<PizzaBodem>();

    var myWorksheet = xlPackage.Workbook.Worksheets.First(); //select sheet here
    var totalRows = myWorksheet.Dimension.End.Row;
    var totalColumns = myWorksheet.Dimension.End.Column;

    for (int rowNum = 2; rowNum <= totalRows; rowNum++) //select starting row here
    {
        tempName = myWorksheet.GetValue(rowNum, 1).ToString();
        tempDiameter = myWorksheet.GetValue(rowNum, 2).ToString();
        tempDescription = myWorksheet.GetValue(rowNum, 3).ToString();
        tempFee = myWorksheet.GetValue(rowNum, 4).ToString();
        tempFee = new string(tempFee.Where(c => (Char.IsDigit(c)||c =='.'||c==',')).ToArray());
        tempFee = tempFee.Replace(",", ".");
        if (myWorksheet.GetValue(rowNum, 5).ToString() == "Ja")
        {
            tempAvailable = true;
        } else
        {
            tempAvailable = false;
        }

        pizzaBodems.Add(new PizzaBodem(tempName, tempDiameter, tempDescription,
            decimal.Parse(tempFee, CultureInfo.InvariantCulture), tempAvailable));
        log.Info("Succesfully added line:" + rowNum);
        tempName = "";
        tempDiameter = "";
        tempDescription = "";
        tempFee = "";
        tempAvailable = false;

    }
    return pizzaBodems;
}
```

Fig. 9 - C# Filling item list

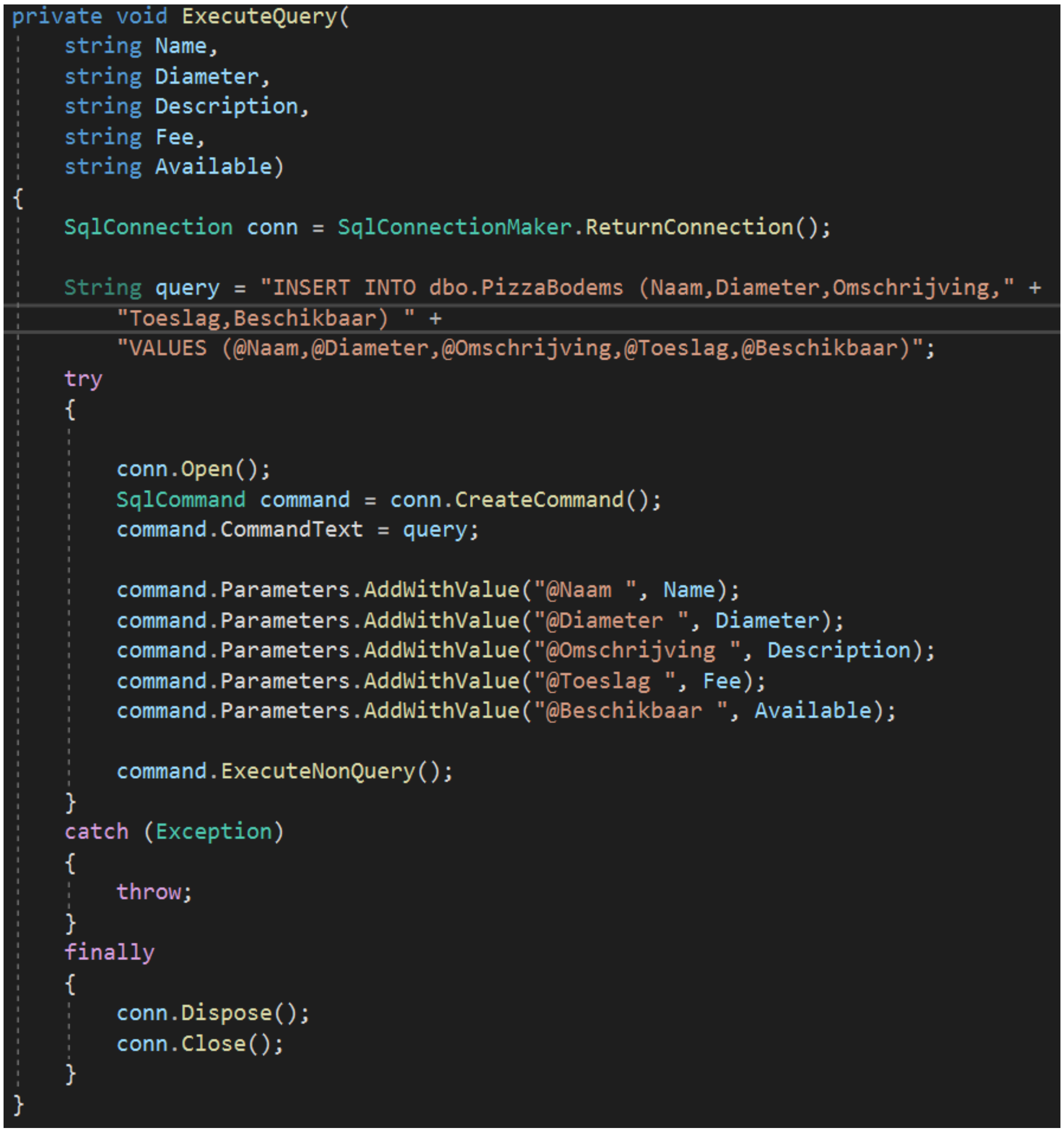
Items from original Mario files are instantiated as objects based on prepared classes and are loaded into a list. At this stage we loop through the rows and convert them to Strings, replace commas with period signs to prepare the data for conversion into numeric format later on and prepare the Boolean data for further conversion by setting them to *true/false* values following a more prevalent programming convention. On each iteration of the loop an object is being created and added to the list.
Above an Excel file (*.xlsx) is being loaded hence the use of *ExcelPackage* from the *EPPlus* library. This library has no external dependencies when working within the .NET environment and is free for non-commercial use. For other file types we use the following: *SystemIOStreamReader* for *.csv and *.txt files; *OleDBDataReader* via *OleDBConnection* to connect to MS Access *.mdb files. Both available as standard .NET environment offering basic functionality for the needs of the project.



```
public void Upload(List<PizzaBodem> pizzaBodems)
{
    log.Info("- - - - -");
    log.Info("Running : " + System.Reflection.MethodBase.GetCurrentMethod().Name);
    log.Info("- - - - -");

    //Drop table and create new one
    SqlConnection conn = SqlConnectionMaker.ReturnConnection();

    try
    {

        conn.Open();
        SqlCommand command = conn.CreateCommand();
        command.CommandText = "DROP TABLE IF EXISTS dbo.PizzaBodems";
        command.ExecuteNonQuery();
        command.CommandText = "CREATE TABLE dbo.PizzaBodems (Naam varchar(200),Diameter varchar(200)," +
            "Omschrijving varchar(200),Toeslag varchar(200),Beschikbaar varchar(200))";
        command.ExecuteNonQuery();

    }
    catch (Exception)
    {
        throw;
    }
    finally
    {
        conn.Dispose();
        conn.Close();
    }

    foreach (PizzaBodem pizzaBodem in pizzaBodems)
    {

        ExecuteQuery(pizzaBodem.Name.ToString(),
            pizzaBodem.Diameter.ToString(),
            pizzaBodem.Description.ToString(),
            pizzaBodem.Fee.ToString(),
            pizzaBodem.Available.ToString());
    }

}
```

Fig. 10 - C# Upload

After establishing a connection with the data base we execute queries to first drop the table if it exists and then create a new one. That way we wanted to guarantee data integrity. Connection and queries execution are contained within a *Try/Catch* block to catch exceptions and after execution we close the connection. A *for* loop executes the *INESRT* query filling the rows on each iteration of the loop.

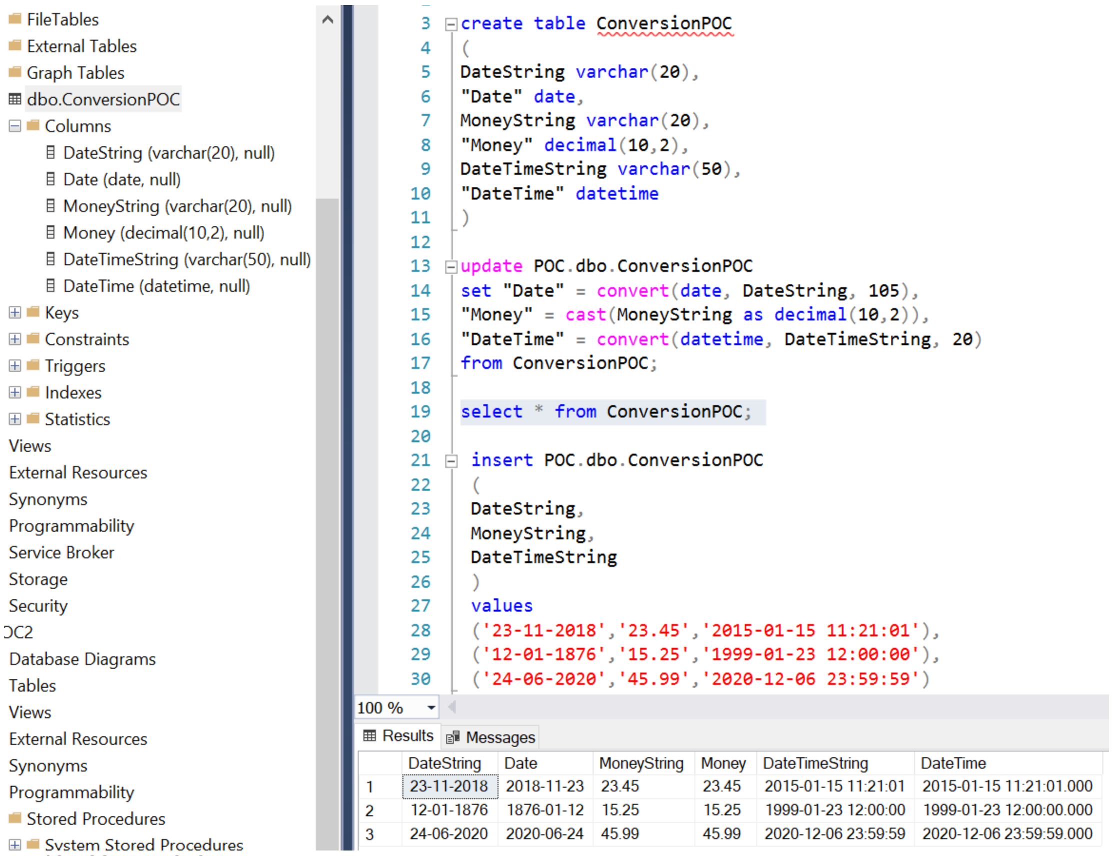


```
private void ExecuteQuery(
    string Name,
    string Diameter,
    string Description,
    string Fee,
    string Available)
{
    SqlConnection conn = SqlConnectionMaker.ReturnConnection();

    String query = "INSERT INTO dbo.PizzaBodems (Naam,Diameter,Omschrijving," +
        "Toeslag,Beschikbaar) " +
        "VALUES (@Naam,@Diameter,@Omschrijving,@Toeslag,@Beschikbaar)";
    try
    {

        conn.Open();
        SqlCommand command = conn.CreateCommand();
        command.CommandText = query;

        command.Parameters.AddWithValue("@Naam ", Name);
        command.Parameters.AddWithValue("@Diameter ", Diameter);
        command.Parameters.AddWithValue("@Omschrijving ", Description);
        command.Parameters.AddWithValue("@Toeslag ", Fee);
        command.Parameters.AddWithValue("@Beschikbaar ", Available);

        command.ExecuteNonQuery();
    }
    catch (Exception)
    {
        throw;
    }
    finally
    {
        conn.Dispose();
        conn.Close();
    }
}
```

Fig. 11 - C# Insert query execution

This is the *INSERT* query execution method. Actual values are being passed as arguments to the AddWithValue function.

2. SQL Stored Procedures



- FileTables
- External Tables
- Graph Tables
- dbo.ConversionPOC
  - Columns
    - DateString (varchar(20), null)
    - Date (date, null)
    - MoneyString (varchar(20), null)
    - Money (decimal(10,2), null)
    - DateTimeString (varchar(50), null)
    - DateTime (datetime, null)
  - Keys
  - Constraints
  - Triggers
  - Indexes
  - Statistics
- Views
- External Resources
- Synonyms
- Programmability
- Service Broker
- Storage
- Security
- C2
- Database Diagrams
- Tables
- Views
- External Resources
- Synonyms
- Programmability
- Stored Procedures
- System Stored Procedures

```sql
create table ConversionPOC
(
DateString varchar(20),
"Date" date,
MoneyString varchar(20),
"Money" decimal(10,2),
DateTimeString varchar(50),
"DateTime" datetime
)

update POC.dbo.ConversionPOC
set "Date" = convert(date, DateString, 105),
"Money" = cast(MoneyString as decimal(10,2)),
"DateTime" = convert(datetime, DateTimeString, 20)
from ConversionPOC;

select * from ConversionPOC;

 insert POC.dbo.ConversionPOC
 (
 DateString,
 MoneyString,
 DateTimeString
 )
 values
 ('23-11-2018','23.45','2015-01-15 11:21:01'),
 ('12-01-1876','15.25','1999-01-23 12:00:00'),
 ('24-06-2020','45.99','2020-12-06 23:59:59')
```

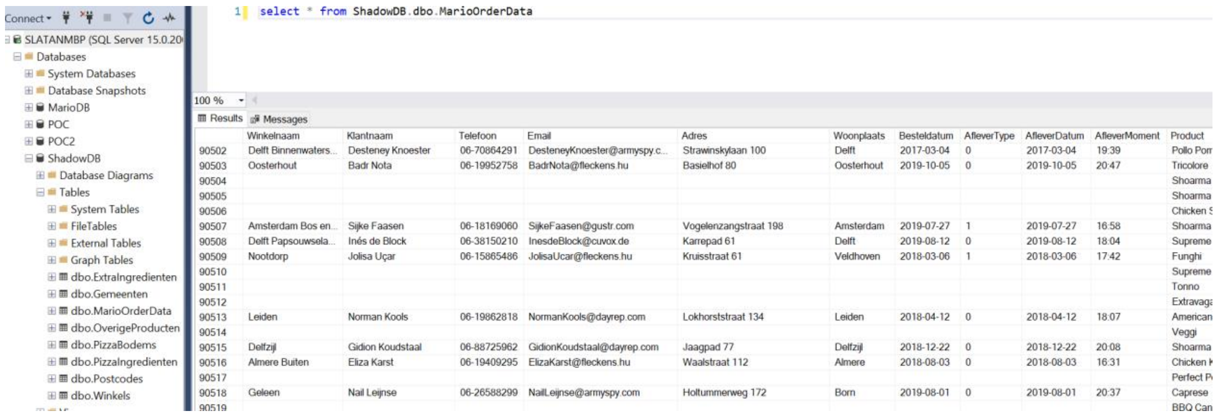
100 %

Results | Messages

	DateString	Date	MoneyString	Money	DateTimeString	DateTime
1	23-11-2018	2018-11-23	23.45	23.45	2015-01-15 11:21:01	2015-01-15 11:21:01.000
2	12-01-1876	1876-01-12	15.25	15.25	1999-01-23 12:00:00	1999-01-23 12:00:00.000
3	24-06-2020	2020-06-24	45.99	45.99	2020-12-06 23:59:59	2020-12-06 23:59:59.000

Fig. 12 - SQL proof of concept

As we've decided to do the data type conversion withing the SQL Server environment we start by creating proof of concepts snippets to test the conversion process. This proved to be straightforward as long as the supplied data is clean. That's why we make sure to clean the data during import into the shadow DB and to use proper formatting for date and time fields.



- SLATANMBP (SQL Server 15.0.20
  - Databases
    - System Databases
    - Database Snapshots
    - MarioDB
    - POC
    - POC2
    - ShadowDB
      - Database Diagrams
      - Tables
        - System Tables
        - FileTables
        - External Tables
        - Graph Tables
        - dbo.ExtraIngredienten
        - dbo.Gemeenten
        - dbo.MarioOrderData
        - dbo.OverigeProducten
        - dbo.PizzaBodems
        - dbo.PizzaIngredienten
        - dbo.Postcodes
        - dbo.Winkels

```sql
select * from ShadowDB.dbo.MarioOrderData
```

100 %

Results | Messages

	Winkelnaam	Klantnaam	Telefoon	Email	Adres	Woonplaats	Besteldatum	AfleverType	AfleverDatum	AfleverMoment	Product
90502	Delft Binnenwaters...	Desteney Knoester	06-70864291	DesteneyKnoester@armyspy.c...	Strawinskylaan 100	Delft	2017-03-04	0	2017-03-04	19:39	Pollo Por
90503	Oosterhout	Badr Nota	06-19952758	BadrNota@fleckens.hu	Basielhof 80	Oosterhout	2019-10-05	0	2019-10-05	20:47	Tricolore
90504											Shoarma
90505											Shoarma
90506											Chicken S
90507	Amsterdam Bos en...	Sijke Faasen	06-18169060	SijkeFaasen@gustr.com	Vogelenzangstraat 198	Amsterdam	2019-07-27	1	2019-07-27	16:58	Shoarma
90508	Delft Papsouwsela...	Inés de Block	06-38150210	InesdeBlock@cuvox.de	Karrepad 61	Delft	2019-08-12	0	2019-08-12	18:04	Supreme
90509	Nootdorp	Jolisa Uçar	06-15865486	JolisaUcar@fleckens.hu	Kruisstraat 61	Veldhoven	2018-03-06	1	2018-03-06	17:42	Funghi
90510											Supreme
90511											Tonno
90512											Extravaga
90513	Leiden	Norman Kools	06-19862818	NormanKools@dayrep.com	Lokhorststraat 134	Leiden	2018-04-12	0	2018-04-12	18:07	American
90514											Veggi
90515	Delfzijl	Gidion Koudstaal	06-88725962	GidionKoudstaal@dayrep.com	Jaagpad 77	Delfzijl	2018-12-22	0	2018-12-22	20:08	Shoarma
90516	Almere Buiten	Eliza Karst	06-19409295	ElizaKarst@fleckens.hu	Waalstraat 112	Almere	2018-08-03	0	2018-08-03	16:31	Chicken K
90517											Perfect P
90518	Geleen	Nail Leijnse	06-26588299	NailLeijnse@armyspy.com	Holtummerweg 172	Born	2019-08-01	0	2019-08-01	20:37	Caprese
90519											BBQ Can

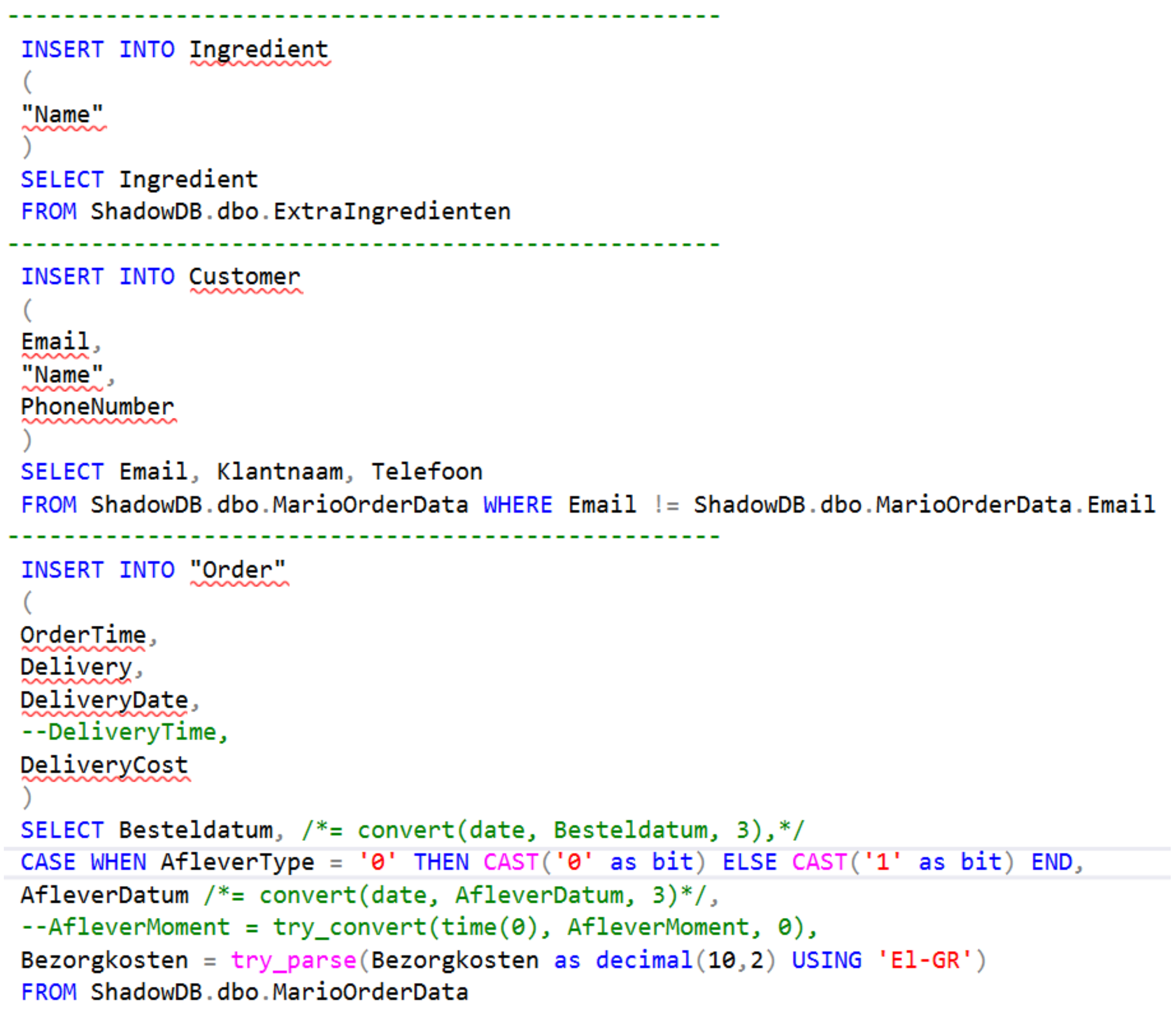
Fig. 13 - SQL ShadowDB

MarioOrderData as viewed in our ShadowDB. It basically mirrors the original file with some improvements to formatting and cleaning up of the unnecessary empty rows etc.

```sql
ALTER PROCEDURE spImportIntoMarioDB
AS
BEGIN
SET IDENTITY_INSERT Municipality ON
--SET IDENTITY_INSERT Zipcode ON
------------------------------------------------------
        INSERT INTO dbo.Municipality
        (
        MunicipalityCode,
        "Name"
        )
        SELECT Kode, Naam
        FROM ShadowDB.dbo.Gemeenten
------------------------------------------------------
        INSERT INTO dbo.Zipcode
        (
        Zipcode,
        SeriesIndex,
        BreakpointFrom,
        BreakpointTo,
        City,
        Streetname,
        MunicipalityCode
        )
        SELECT Postcode, Reeksind, Breekpunt_van, Breekpunt_tem, Woonplaats,Straatnaam,Gemeentecode
        FROM ShadowDB.dbo.Postcodes
------------------------------------------------------
        INSERT INTO dbo.VoucherType
        (
        "Name"
        )
        SELECT GebruikteCoupon
        FROM ShadowDB.dbo.MarioOrderData
------------------------------------------------------
        INSERT INTO dbo.ProductCategory
        (
        "Name"
        )
        SELECT Categorie
        FROM ShadowDB.dbo.PizzaIngredienten
------------------------------------------------------
```

Fig. 14 - SQL 1st draft INSERT SP

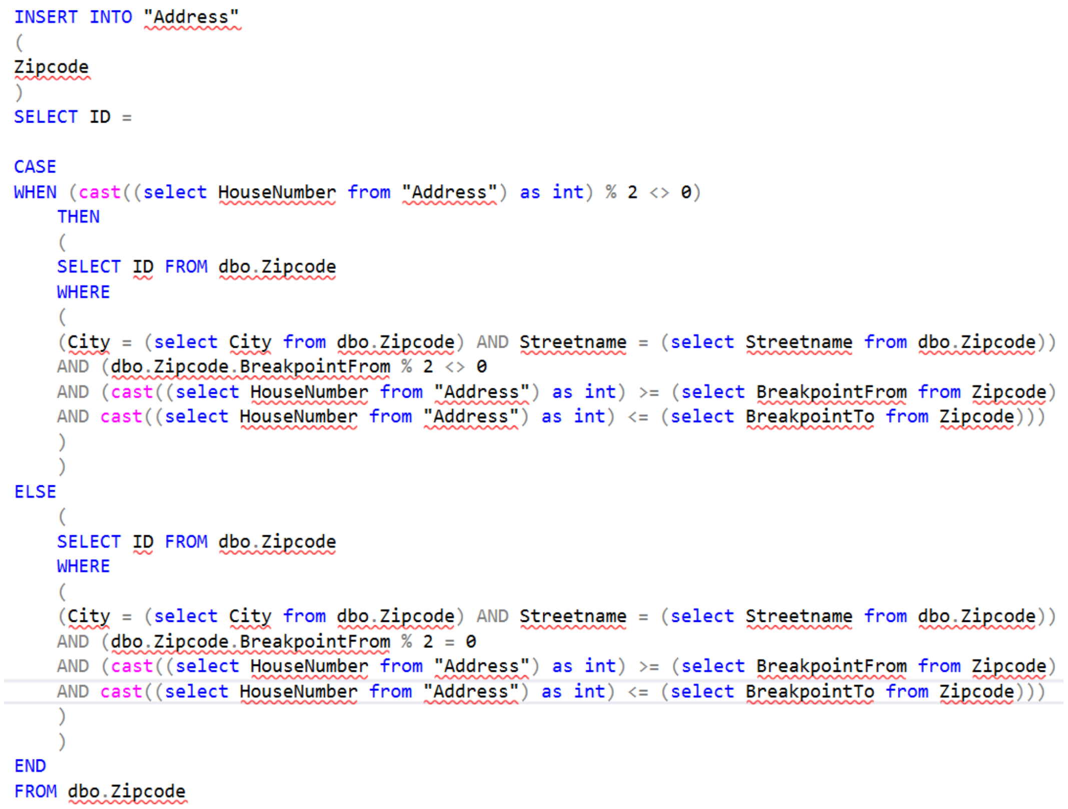
As our data are placed in a data base already, we start to put together our INSERT stored procedures. On the supplied screenshot pictured a very 1st simple draft that copies the data between the two data bases. This version of the SP is far from finished however since it doesn't properly redistribute the data into its target locations.



```sql
--------------------------------------------------
INSERT INTO Ingredient
(
"Name"
)
SELECT Ingredient
FROM ShadowDB.dbo.ExtraIngredienten
--------------------------------------------------
INSERT INTO Customer
(
Email,
"Name",
PhoneNumber
)
SELECT Email, Klantnaam, Telefoon
FROM ShadowDB.dbo.MarioOrderData WHERE Email != ShadowDB.dbo.MarioOrderData.Email
--------------------------------------------------
INSERT INTO "Order"
(
OrderTime,
Delivery,
DeliveryDate,
--DeliveryTime,
DeliveryCost
)
SELECT Besteldatum, /*= convert(date, Besteldatum, 3),*/
CASE WHEN AfleverType = '0' THEN CAST('0' as bit) ELSE CAST('1' as bit) END,
AfleverDatum /*= convert(date, AfleverDatum, 3)*/,
--AfleverMoment = try_convert(time(0), AfleverMoment, 0),
Bezorgkosten = try_parse(Bezorgkosten as decimal(10,2) USING 'El-GR')
FROM ShadowDB.dbo.MarioOrderData
```

Fig. 15 - SQL further SP development

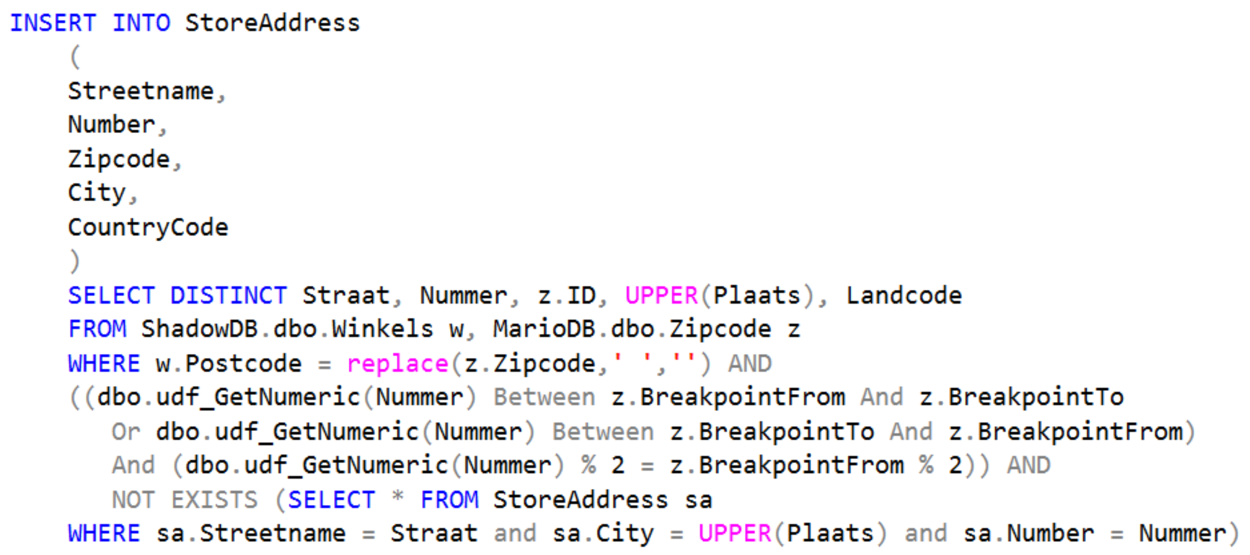
A further improvement on the SP which now converts the data to the required types. Converting to Boolean type can be seen in the *CASE* statement. Some issues we've had is proper date/time string formatting due to the limited possibilities of formatting within SQL Server. We managed to solve it by pre-formatting in the C# software, so the string matches the required format.



```
INSERT INTO "Address"
(
Zipcode
)
SELECT ID =

CASE
WHEN (cast((select HouseNumber from "Address") as int) % 2 <> 0)
    THEN
    (
    SELECT ID FROM dbo.Zipcode
    WHERE
    (
    (City = (select City from dbo.Zipcode) AND Streetname = (select Streetname from dbo.Zipcode))
    AND (dbo.Zipcode.BreakpointFrom % 2 <> 0
    AND (cast((select HouseNumber from "Address") as int) >= (select BreakpointFrom from Zipcode)
    AND cast((select HouseNumber from "Address") as int) <= (select BreakpointTo from Zipcode)))
    )
    )
ELSE
    (
    SELECT ID FROM dbo.Zipcode
    WHERE
    (
    (City = (select City from dbo.Zipcode) AND Streetname = (select Streetname from dbo.Zipcode))
    AND (dbo.Zipcode.BreakpointFrom % 2 = 0
    AND (cast((select HouseNumber from "Address") as int) >= (select BreakpointFrom from Zipcode)
    AND cast((select HouseNumber from "Address") as int) <= (select BreakpointTo from Zipcode)))
    )
    )
END
FROM dbo.Zipcode
```

Fig. 16 - SQL housenumber zipcode matching

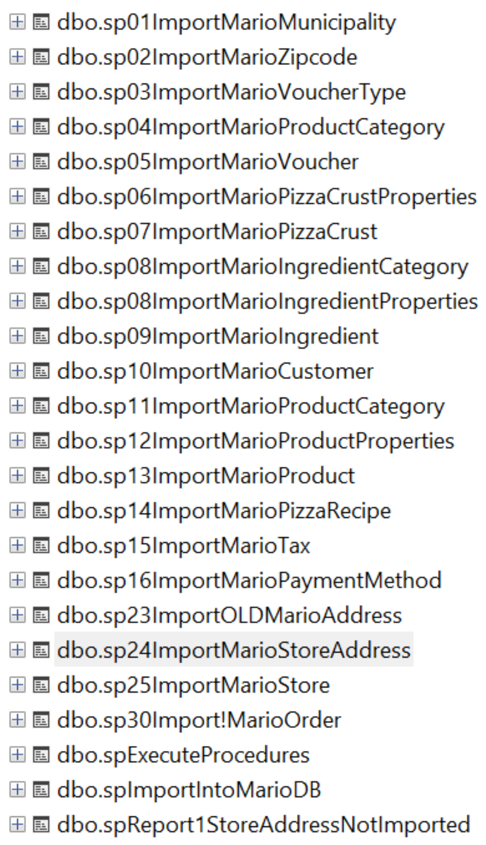
The zipcode table contains the street name and ranges of house numbers – separate for odd and separate for even numbers. An address has to be not only within a given range, but it also has be compared against the right range – odd or even. To couple the proper zipcode to given address we decided on using *MODULO* to identify whether house number is odd or even and then to compare whether it's within the given range. *MOD* is also used to identify whether a row concerns even or odd ranges. This early draft doesn't work properly.



```
INSERT INTO StoreAddress
    (
    Streetname,
    Number,
    Zipcode,
    City,
    CountryCode
    )
    SELECT DISTINCT Straat, Nummer, z.ID, UPPER(Plaats), Landcode
    FROM ShadowDB.dbo.Winkels w, MarioDB.dbo.Zipcode z
    WHERE w.Postcode = replace(z.Zipcode,' ','') AND
    ((dbo.udf_GetNumeric(Nummer) Between z.BreakpointFrom And z.BreakpointTo
       Or dbo.udf_GetNumeric(Nummer) Between z.BreakpointTo And z.BreakpointFrom)
       And (dbo.udf_GetNumeric(Nummer) % 2 = z.BreakpointFrom % 2)) AND
       NOT EXISTS (SELECT * FROM StoreAddress sa
    WHERE sa.Streetname = Straat and sa.City = UPPER(Plaats) and sa.Number = Nummer)
```

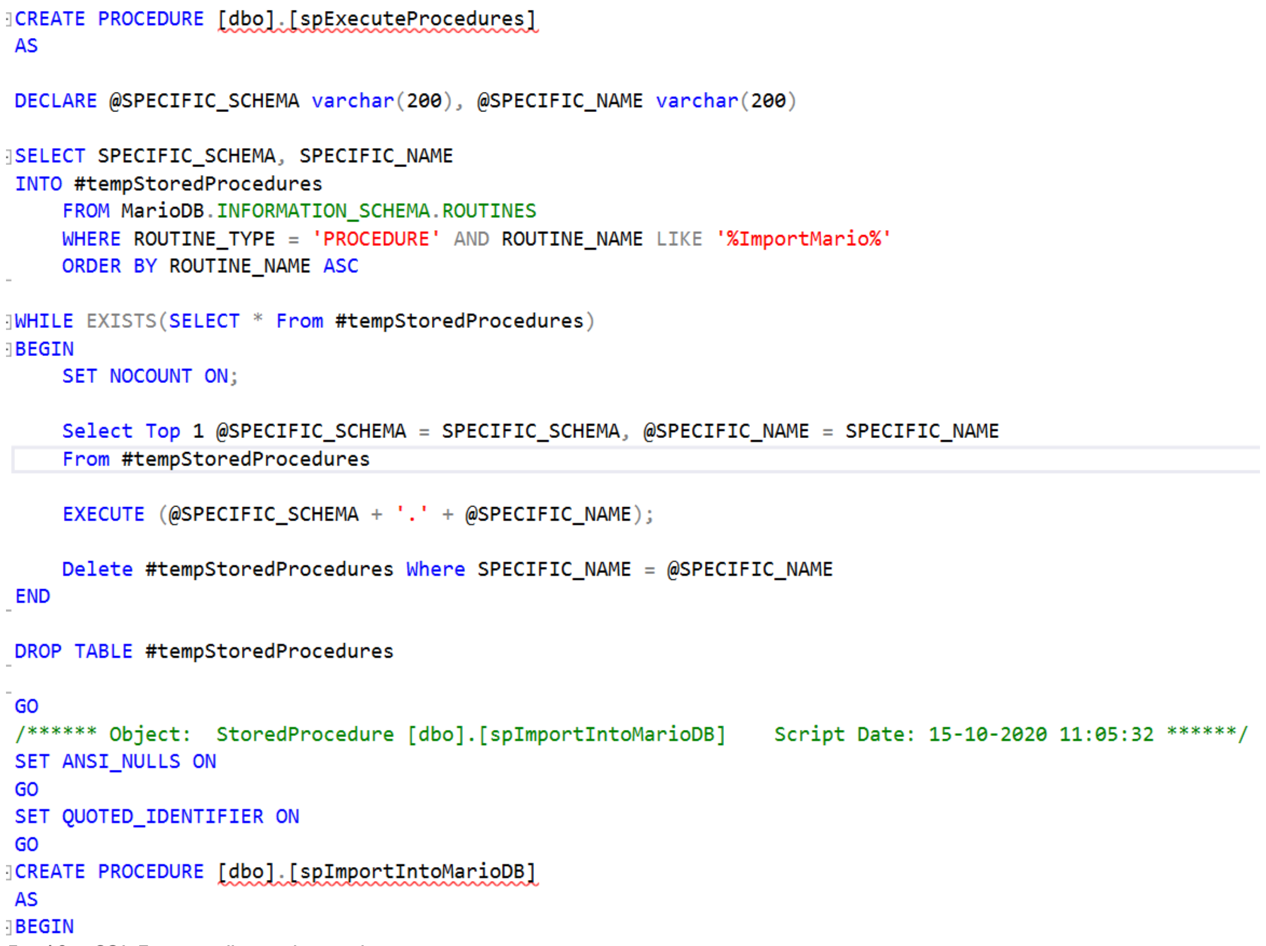
Fig. 17 - SQL housenumber zipcode matching 2

This is the working version of the SP from previous page. We've included a check whether given data exists in the DB already to avoid duplicates. Working on this SP it turned out not all zip codes are included within the supplied files making binding many of the addresses from supplied files impossible. This would have to be reported back to the customer and a solution would have to be pursued.



```
⊞ dbo.sp01ImportMarioMunicipality
⊞ dbo.sp02ImportMarioZipcode
⊞ dbo.sp03ImportMarioVoucherType
⊞ dbo.sp04ImportMarioProductCategory
⊞ dbo.sp05ImportMarioVoucher
⊞ dbo.sp06ImportMarioPizzaCrustProperties
⊞ dbo.sp07ImportMarioPizzaCrust
⊞ dbo.sp08ImportMarioIngredientCategory
⊞ dbo.sp08ImportMarioIngredientProperties
⊞ dbo.sp09ImportMarioIngredient
⊞ dbo.sp10ImportMarioCustomer
⊞ dbo.sp11ImportMarioProductCategory
⊞ dbo.sp12ImportMarioProductProperties
⊞ dbo.sp13ImportMarioProduct
⊞ dbo.sp14ImportMarioPizzaRecipe
⊞ dbo.sp15ImportMarioTax
⊞ dbo.sp16ImportMarioPaymentMethod
⊞ dbo.sp23ImportOLDMarioAddress
⊞ dbo.sp24ImportMarioStoreAddress
⊞ dbo.sp25ImportMarioStore
⊞ dbo.sp30Import!MarioOrder
⊞ dbo.spExecuteProcedures
⊞ dbo.spImportIntoMarioDB
⊞ dbo.spReport1StoreAddressNotImported
```

Fig. 18 - SQL all stored procedures used



```
CREATE PROCEDURE [dbo].[spExecuteProcedures]
AS

DECLARE @SPECIFIC_SCHEMA varchar(200), @SPECIFIC_NAME varchar(200)

SELECT SPECIFIC_SCHEMA, SPECIFIC_NAME
INTO #tempStoredProcedures
    FROM MarioDB.INFORMATION_SCHEMA.ROUTINES
    WHERE ROUTINE_TYPE = 'PROCEDURE' AND ROUTINE_NAME LIKE '%ImportMario%'
    ORDER BY ROUTINE_NAME ASC

WHILE EXISTS(SELECT * From #tempStoredProcedures)
BEGIN
    SET NOCOUNT ON;

    Select Top 1 @SPECIFIC_SCHEMA = SPECIFIC_SCHEMA, @SPECIFIC_NAME = SPECIFIC_NAME
    From #tempStoredProcedures

    EXECUTE (@SPECIFIC_SCHEMA + '.' + @SPECIFIC_NAME);

    Delete #tempStoredProcedures Where SPECIFIC_NAME = @SPECIFIC_NAME
END

DROP TABLE #tempStoredProcedures

GO
/****** Object:  StoredProcedure [dbo].[spImportIntoMarioDB]    Script Date: 15-10-2020 11:05:32 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[spImportIntoMarioDB]
AS
BEGIN
```

Fig. 19 – SQL Execute all stored procedures

We have contained all of our Stored Procedures within a script, which is a functionality within the SQL Server Management Studio. The creation of the data base itself, all PK/FK constraints are also handled by the script. This allows us to quickly set the whole data base back up again when the data gets too messy due to the trial and error during testing. After running the script we have all the SP's already created and it's just a matter of running them all in consecutive order. This is being handled by the pictured SP. It looks for specific naming scheme and executes the SP's till it reaches the last one, at which point the execution is terminated.