

2012

David Otterbein
s1775189

Supervised by:
Prof. Dr. Ir. Marco Aiello
Dr. Herman Balsters



university of
 groningen

faculty of mathematics
and natural sciences

[ORM Business Tool]

A bachelor project in computing science. Development of a tool prototype that helps to create a ORM database on the basis of a BPMN model.

Table of Contents

List of Abbreviations and Symbols	2
1. Introduction.....	3
2. Project Description	4
3. Analysis	5
4. Concept	9
4.1 Parsing xpdI:	9
4.2 Creating the ORM:.....	10
5. Program code	12
The activity class:.....	12
The CLIToolView class:.....	12
The Creator class:	12
The Entity class:	13
The Fact class:.....	14
The Main class:	14
The Parser class:	14
The ToolController class:	15
The ToolView class:	16
The Transition class:	16
6. Testing	17
7. Conclusion	23
8. Future Work	24
9. Bibliography.....	25
10. Appendix.....	26
I. ORM Business Tool user guide	26

List of Abbreviations and Symbols

RUG	Rijksuniversiteit Groningen
ORM	Object Role Modeling
BPMN	Business Process Modeling Notation
BPM	Business Process Modeling
UI	User Interface
DB	Database
OLE	ORM Logic-based English
NORMA	Natural ORM Architect (for Visual Studio)
Javadoc	API (Application programming interface) documentation from Java source code
BPMC	Business Process Model Collaboration
XPDL	XML Process Definition Language
XML	Extensible Markup Language
W3C	World Wide Web Consortium
PHP	Hypertext Preprocessor formerly Personal Home Page Tools
JRE	Java Runtime Environment
I/O	Input/output
Refmode	Reference mode
UML	Unified Modelling Language

1. Introduction

The fastest way to finish work is to make you of computers and automation of the business process you might be going through right now. So you want to collect data from the related persons and store them in the process to get a good overview. To design such a working system you basically need to go through three different processes. First you need a business model (BPMN). From the business model you can derive a ORM database to handle all the data you encounter. Finally a user interface (UI) is needed so that the whole system can communicate with the user.

This means you have to work with three different tools. For example Bizagi (BPMN tool), Visual Studios with NORMA (ORM DB tool) and some kind of UI tool or program the interface yourself.

As a consequence you have to be an expert with those tools or outsource it to an expert. This is undesirable and the question is if there the possibility to create a tool which acts as an bridge between these designs steps. The focus of this project is to design a prototype of a tool which builds a bridge between the business process model and the object related database model. To make this transition more natural ORM logic based English (OLE)¹ can be used. This language is a hybrid between the syntax of ORM and English. It is accurate enough to describe and build a ORM database from it, but it feels more natural than for example the NORMA syntax. That means that the tool should parses a BPMN model and then, with the OLE input from the user, a ORM database model should be created.

¹ Dr. H Balsters. (2012, June). ORM Logic-based English (OLE) and the ORM ReDesigner tool: Fact-based Reengineering and Migration of Relational Databases.

2. Project Description

The following chapter gives an overview of how the project is structured and the course of development.

In this project a prototype will be developed to show that and how it would be possible to develop tool, as. The basic case of the prototype is the process of a bank transfer² with the related BPMN and ORM model. This case will be used to show that the prototype fulfil the requirements and as an example to illustrate the code or algorithms.

The main functionality can be described in one sentence:

A tool which converts a BPMN model to a ORM model by parsing the OLE user input.

The project is divided into three phases: The research, the implementation and the documentation phase.

In the research phase Bizagi (the BPMN tool), NORMA/Visual Studio (the ORM tool) and other related requirements or software will be investigated.

During the implementation phase the code of the prototype is written. This phase is subdivided into the general phase and the refinement phase. In the general phase the basic code will be written which fulfils the requirements from the research phase and in the refinement phase extra requirements which may arise will be covered. Also the code will be refactored so it satisfies the criteria of model-view-controlling. After the implementation is finished the prototype will be tested again to control the quality of it.

The aim of the documentation phase is to write the report and the Javadoc. In the report the notes which has been taken during the other phases will be written out and crucial parts of the source code will be explained.

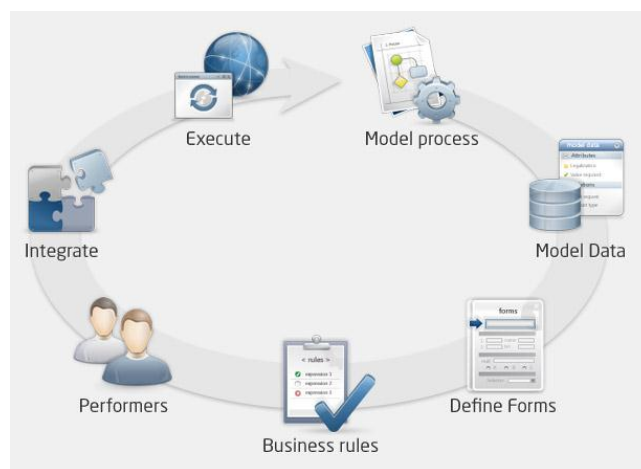
² Dr. H Balsters. (2012 Jan). Case for Lecture on Integration of Process- and Data modelling.

3. Analysis

This chapter summarises the analysis of the software and articles which were studied before the prototype has been programmed.

Bizagi:

BPMN is the most commonly used language to describe business processes, this is also why it has been chosen to represent processes in this project. There is also a variety of tools for BPMN modelling. Bizagi is a company which is specialised on business process software. They have two main products, the “BPMN Process Modeler” and the “BPM Suite”³. The “BPM Suite” is a whole environment for business processes in which it is not only possible to design business processes but also to model the data, build the forms, integrate and execute the model.



Source: <http://www.bizagi.com/images/specialImgs/solutions-graphic.jpg>

The “BPMN Process Modeler” is a freeware tool to model and document business processes. This tool was chosen because it has a good UI and is very reliable. After the process had been modelled in the program it can be saved to a bpm or a bpmc file, a Bizag Diagram Model or a Bizagi Collaboration Model. It also can be exported as an image, visio-, xpdI model or just the attributes of the model as a xml file. The xpdI and the xml file are both plaintext and have the extensible markup language format or rather an extension of it. Because they are pre-defined by the W3C they have a steady format and therefore can be parsed to read out certain attributes and values. This is why the xpdI format has been chosen to be parsed by the program.

ORM:

ORM has been chosen as the language for databases and database models because it is very good in describing the semantic relationship between the entities of a database. This is a big advantage over the more common UML, which has no such semantic description layer. Because we are working with business processes the semantic connection is very important and should not be lost if the model is migrated into a database.

³ Bizagi. (2012 May). Bizagi BPM Overview. Available: <http://www.bizagi.com/modeler/>

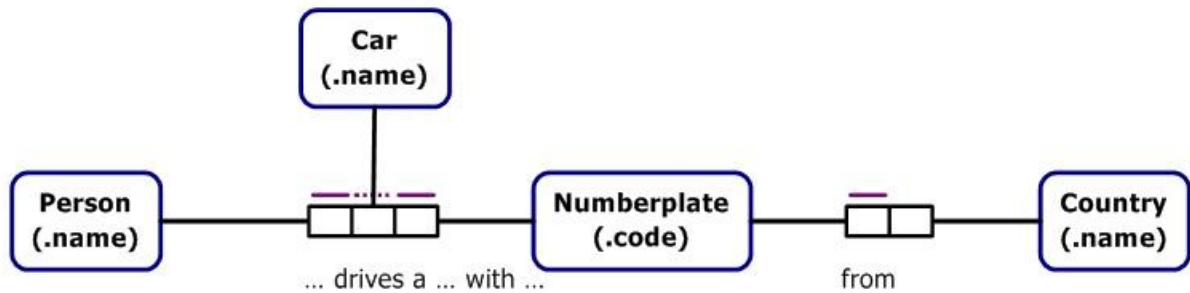
Visual Studio and NORMA:

NORMA (Natural ORM Architect for Visual Studio) is a plugin for Microsoft Visual Studio developed by ORM foundation. It has a ORM Fact Editor where, if entered in the right format, text is transformed to a ORM database model.

ORM Logic-based English (OLE):

OLE is a language which is precise enough to describe a ORM database. With the OLE-method an ORM database can be represented in OLE.

If we take the following ORM model:



A population would be, *Carl* drives a *Ford* with *AA-12-34* from *USA*. To represent the model in OLE three steps must be performed⁴.

Step 1: List the elementary facts for the target model:

1. Person drives a Car with Numberplate
2. Numberplate from Country

Step 2: For every elementary fact, list all entities and values, along with reference modes and basic types:

1. Person is Entity (and is referred to by name (of type varchar))
2. Car is Entity (and is referred to by name (of type varchar))
3. Numberplate is Entity (and is referred to by code (of type varchar))
4. Country is Entity (and is referred to by name (of type varchar))

Step 3: For every elementary fact, list all uniqueness constraints:

1. **For each** Person and Numberplate , **there is exactly one** Car, **where** Person drives a Car with Numberplate.
2. **For each** Numberplate , **there is exactly one** Country, **where** Numberplate from Country.

⁴ Dr. H Balsters. (2012, June). ORM Logic-based English (OLE) and the ORM ReDesigner tool: Fact-based Reengineering and Migration of Relational Databases.

In ORM, the model as text would be:

Step 1: Facts:

1. **Person**(.name) drives a **Car**(.name) with **Numberplate**(.code)
2. **Numberplate**(.code) from **Country**(.name)

Step 2: Verbalization:

Person is an entity type.

Reference Scheme: Person has Person name.

Reference Mode: .name.

Fact Types:

Person has Person name.

Person drives a Car with Numberplate.

Car is an entity type.

Reference Scheme: Car has Car name.

Reference Mode: .name.

Fact Types:

Car has Car name.

Person drives a Car with Numberplate.

Numberplate is an entity type.

Reference Scheme: Numberplate has Numberplate code.

Reference Mode: .code.

Fact Types:

Numberplate has Numberplate code.

Person drives a Car with Numberplate.

Numberplate from Country.

Person drives a Car with Numberplate.

For each Person **and** Numberplate,

that Person drives a **at most one** Car with that Numberplate.

This association with Person, Numberplate **provides the preferred identification scheme for** PersonDrivesACarWithNumberplate.

Country is an entity type.

Reference Scheme: Country has Country name.

Reference Mode: .name.

Fact Types:

Country has Country name.

Numberplate from Country.

Numberplate from Country.

Each Numberplate from at most one Country.

It is possible that more than one Numberplate from the same Country.

Obviously purpose of the ORM representation is to provide a natural, easy to read visual representation and a very detailed text representation. Compared to the OLE representation, OLE is much more natural to read and has almost as much as details. This is why OLE has been chosen as an input language for the tool and then it is automatically converted to the NORMA fact editor format. This way the tool is more accessible for non ORM experts compared to the NORMA fact editor.

PHP tool:

Earlier this year a project about data normalization and migration in OLE reached its final state. Basically the project was about migrating a database to a ORM database and how far this process could be automatized⁵. First it was planned to rewrite the PHP code to a tool which transfers BPMN to ORM notation, but because of various reasons this idea could not be followed. Mostly because there were fewer parallels between the project than expected.

Java:

After it was clear that the PHP code from the project mentioned above would not be used, Java has been chosen as the programming language for the tool. The reason for doing so was the tool should run on every platform and with java it does run on every machine with a java runtime environment (JRE). Also Java has a very big library of reliable packages which for example can be used to parse an XML document.

Because it is a prototype the program does not have a UI and all the user communication is handled text based via a console. To make a future extension of the tool easier to implement and the code more readable, the source code follows the model-view-controlling method. This way only the view part of the program has to be changed to create a graphical user interface.

⁵ J.J. Pastoor. (2012, Jan). Databasemigratie en -normalisatie in ORM Logic-based English.

4. Concept

The chapter guides explains the design of the prototype. The design has been written before the code has been written and only explains theoretically how the program should work. The content of the design are the two main parts of the tool: parsing an xpdI file and creating the NORMA facts.

4.1 Parsing xpdI:

The xpdI file is the output file of the BPM model from the user and has been generated with the BPMN Process Modeler. It should be parsed as follows:

Each process element in the xpdI is named like this:

```
<Activity Id="b676c793-7a8c-40f8-af72-6d0920097c8e" Name="Noot">
```

The order of the activities in the xpdI file is the order of how they have been added to the model. Unfortunately it is not based on which activity comes first by traversing the flow of the model. To restore the actual order the activities must be traversed, starting by the first activity we encounter and then getting the activities which are connected to the first one by the transition. This is why it is crucial that start node has been added first to the model. To find the transitions of an activity the Id needs to be stored as a key, so that it can be compared to the Ids in the <Transitions> section. Furthermore the Ids are not used because the long key is not really readable for the users. To traverse the model the tool should read all the activities and transition from the xpdI file and stores them.

Then the Id of the start element (first element that appears with a Activity Id) should be searched for in the transition list in the “From” parameter:

```
<Transition Id="d90aa8b8-7682-47d5-a196-154ffd34061d" From="e6f415b2-5583-427c-af58-bd0d44b1d147" To="b676c793-7a8c-40f8-af72-6d0920097c8e">
```

The Id of the transition is not important, but the “To” gives us the Id of the second activity in the process flow. Then this Id needs to be found in the activity Ids so name can be read. With this method we can fill in the list of all the activities which it connect to.

This way we get an unordered list of activity objects. For example a “activitiesItPointsTo” list which contains the Id(s) of the activity(/ies) this activity points to. For the traversal of the activities, the activity also needs a property “visit” which indicates with a Boolean whether the activity has been visited or not.

Now the tool should have a function to order the activity list. The first activity from the list is set as the start activity. Now while all nodes has not been visited yet, it checks the current node if it has been visited or not. If it has not been visited the node will be visited and added to the end of the list of ordered activities. Next it is checked whether the node points to another activity via a path that has not been traversed yet. Then there are two cases:

The node still connects to another node / other nodes:

The next node which the node connects to is loaded as the current node. If the current node has not been visited yet nothing happens and the node will be visited in the next call of the while loop. If however the node has been visited the tool check if this node has any non-traversed paths left. If this is not the case the “The node has no other nodes that it connects to:” part is executed. If there are paths which not has been traversed yet the tool traverses this path and sets the found node as current node. Now if the current node still has non-traversed paths it is part of a loop and still needs to be traversed.

The node has no other nodes that it connects to:

If the current node does not have any more non-traversed connectors left the first unvisited node is searched by calling a function like for example the “getFirstUnvisitedActivity” function. This function should return the first unvisited activity by first checking if there are any non-traversed paths, from all the activities which already been ordered, left which point to a unvisited activity. If there is a non-traversed path from a activity in the ordered list it is traversed and the activity it points to, which has not been visited, is returned.

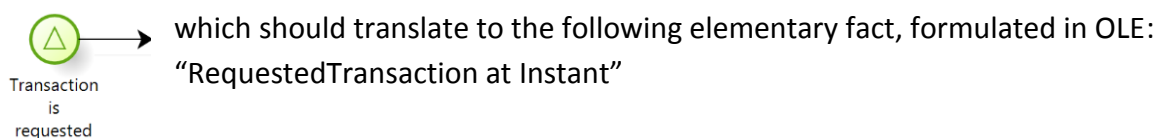
If no activity can be found in the ordered list, the first unvisited activity from the unordered “activities” is selected. This means it will return a random node because we don’t know which activity we will encounter first. But that only means that we could not encounter a activity from our current path so there is not a better way to select an appropriate activity. The function should return null if no activity can be found which means that the while loop can stop because all the activities has been visited, which means they are all in the ordered list.

Now we should have a list of activity names which we can present to the user, so that for each activity the elementary facts can be entered.

4.2 Creating the ORM:

Now there is an ordered list the user needs to enter all elementary facts which correspond to that activity in OLE.

The first activity which must be translated to a ORM DB model in our example is “Transaction is requested”.



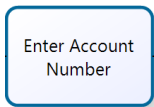
This sentence should be parsed and the user needs to input the reference mode of the entities:

RequestedTransaction is Entity and is referred to by: <nr>

Instant is Entity and is referred to by: <time>

This will yield the NORMA input:
RequestedTransaction(.nr) at Instant(.time)

The next step from our example is “Enter Account Number”. This activity has the following



elementary facts:
RequestedTransaction concerns Account
Account has Balance

This sentence gets parsed and the user needs to input the reference mode of the entities:

Account is Entity and is referred to by: <nr>

Balance is Entity and is referred to by: <EUR:>

This should yield the following NORMA input:

RequestedTransaction(.nr) concerns Account(.nr)

Account(.nr) has Balance(EUR:)

If we copy the three facts and enter them one by one into the NORMA fact editor we will get the following ORM model:

-

Without the external and uniqueness constraints. They cannot be entered into the fact editor and have to be added by hand.

5. Program code

The following chapter gives an overview of the code. The code is divided into classes and from each class the most important lines of code will be explained.

The activity class:

```
private String name, id;  
private LinkedList<String> activitiesItPointsTo;  
private Boolean visit;
```

An activity (object) has a “name” and an “id”. The id is the string which has been generated by the BPM tool. The list “activitiesItPointsTo” contains all the activities which can be reached from this activity. The Boolean “visit” indicates whether this activity has been visited or not.

The CLIToolView class:

This class handles all the input/output interaction with the user. It implements the “ToolView” view, which means that it implements the function which has been defined in the view. That means additionally to the I/O handling, it contains all the text which may be presented to the user while the program runs.

The Creator class:

```
private LinkedList<Activity> activities;  
private LinkedList<Fact> eleFacts;  
private LinkedList<Entity> modelEntities; //to avoid duplicates  
private boolean factsDefined = false;  
private boolean refmodesFilled = false;  
  
//All the refmodes that the program supports  
public static String refmodes = "# Code code ID id Id Name name nr Nr number  
                                time Title title";  
  
//All the refmodes that have to be represented in NORMA with a : after it  
and not a . before it  
public static String specialRefmodes = "AUD CE Celsius cm EUR Fahrenheit kg  
                                       km mile mm USD";
```

The list “activities” contains all the activities which has been passed from the “Parser” class. The list “eleFacts” will contain all the elementary facts which will be entered by the user. From this elementary facts every entity will be stored in “modelEntities” to avoid duplicates. Duplicates would be created if one entity is mentioned in multiple facts. Of course in all facts the same entity should be returned.

The Boolean “factsDefined” indicates whether all facts has been defined, in other words whether the user is finished with entering facts. The Boolean “refmodesFilled” indicates whether there are any entities left with a undefined reference mode. The last two strings contain all the reference modes which should be supported by the program. The check of the reference is an optional feature and has been added to prevent unreadable or useless reference modes.

The most complex function of the Creator is its main function where the ORM output is build.

```

/** Builds a elementary fact from the input string (creates Fact object)
 * @param input
 * @return the whether a elementary fact has been created or not
 */
public boolean buildFact(String input) {
    StringTokenizer st = new StringTokenizer(input);
    String current, sentence;
    LinkedList<Entity> entities = new LinkedList<Entity>();

    sentence = "";
    int entityCount = 0;

    //Builds up a fact until the scanned line is empty
    while (st.hasMoreTokens()) {
        current = st.nextToken();

        //if the first character of the word is upper case it is an Entity
        if(Character.isUpperCase(current.charAt(0))){
            //add the word to the sentence as an entity
            sentence = sentence + "!entitytoken! " + entityCount + " ";
            //and add to the fact's entity list
            Entity e = new Entity(current);
            entities.add(e);
            entityCount++;
        }else {
            //just add to the sentence
            sentence = sentence + current + " ";
        }
    }
    //everything is read in to create a fact, so we create it
    Fact fact = new Fact(sentence, entities);

    //If the fact has at least one and not more than 4 entities it is
    added to the list
    if(fact.getEntities().size() > 0 && fact.getEntities().size() < 4){
        eleFacts.add(fact);
        return true;
    }
    //The fact had too much or too little entities
    return false;
}

```

The comment in the code explains the functionality. If an entity has been recognized (a word which starts with a capital letter) it will be added to the sentence as !entitytoken! and then its number in the entity list of the fact. With the position in the list a reference mode can be added without altering the sentence itself. For example the elementary fact: Student has Mentor, becomes !entitytoken! 0 has !entitytoken! 1. So then if the reference mode has been added the full entity at position null can be filled into the !entitytoken! 0. Finally the sentence could look like this, Student(.nr) has Mentor(.code).

The Entity class:

```

private String name;
private String refmode;
private String externalIdentifier;

```

The entity class has, as shown above, a name, a reference mode and it could have an

external identifier. If the external identifier equals "" (empty) it is ignored and will not show up in the output. If however it has one it will be added to the output.

The Fact class:

```
private LinkedList<Entity> entities;  
private String sentence;
```

A fact has sentence, which is the user input of an elementary fact, where all the entities are replaced with token so they can be found in the list entities, for the reasons mentioned in the "Creator class".

The Main class:

The main class is the first class which is called in the java program. The view "CLIToolView" and the controller "ToolController" are constructed and the "run" function of the controller is called.

The Parser class:

```
private LinkedList<Activity> activities, ordered;  
private LinkedList<Transition> transitions;  
private Document document;
```

The parser has two lists with activities, one is unordered called "activities" and the other one is ordered called "ordered". All the transitions which are needed to sort the activities are stored in the list "transitions". The xpdL document which gets parsed is loaded into the Document "document" and gets parsed by the modified xml parser. The main function of the parser is the function "orderActivities". It, as the name suggests, orders the activities in a way a human would transverse the BPM. This function is needed because in the xpdL file the activities are ordered on the time which they have been added to the model. This order does not feel very natural and will make the creation of an ORM database unnecessarily complicated.

```
/**  
 * Orders all activities so that the start object is the first  
 * and the rest get traversed from there  
 */  
protected void orderActivities() {  
    Activity node, unvisited, tempnode = null;  
    Boolean allVisited = false;  
    node = activities.getFirst();  
  
    //While there are nodes unvisited:  
    while(!allVisited){  
        //If the has not been visited  
        if(!node.getVisit()){  
            //visit the node you have been on (from node) and add it  
            //to the ordered list  
            node.setVisit(true);  
            if(!node.getName().equals("")){  
                ordered.add(node);  
            }  
        }  
  
        //but still connects to another node--  
        if(node.getTo().size() > 0){  
            //get the connected node
```

```

tempnode = node;
node =
activities.get(findActivity(node.getTo().removeFirst()))
;

//if the connected node has been visited check if there
had been other to nodes in the from node
if(node.getVisit()){
    //try this one then
    if(tempnode.getTo().size() > 0){
        node =
        activities.get(findActivity(tempnode.getTo(
        ).removeFirst()));
    }
    //is it part of a nested loop?
    else if(node.getTo().size() > 0){
        //well yes it is so just continue in this
        loop!
    }
    //there is no other node we can connect to. Find
    the first unvisited
    else{
        unvisited =
        getFirstUnvisitedActivity();
        if(unvisited == null){
            allVisited = true;
        } else{
            node = unvisited;
        }
    }
}
} else {
    //--it does not so we take the first unvisited node we
    encounter. If none encountered we are done.
    unvisited = getFirstUnvisitedActivity();
    if(unvisited == null){
        allVisited = true;
    } else{
        node = unvisited;
    }
}
}
activities = ordered;
}

```

The comment in the code explains the general way how the function works. The loop detection is not that straight forward. The activity the tool currently works on is tested, whether there are any other nodes the activity connects to. Then then if there are any other activities which the new activity connects to, it is part of the loop, so we need to work on that node. If it isn't, it means that this is the last node we can encounter from here, so we need to find a new node via a new path. Before any activity gets traversed it is checked if the current is unvisited. If it is it is added to the ordered list. This way all activities will appear in an ordered manner in the list and only once.

The ToolController class:

The ToolController controls the view and the model. Because most functions and I/O calls are made here it gives a good overview of what happens, when and where.

The ToolView class:

The ToolView is the interface which is implemented by the CLIToolView. It also is an overview of all the I/O functions.

The Transition class:

```
private String id, from, to;
```

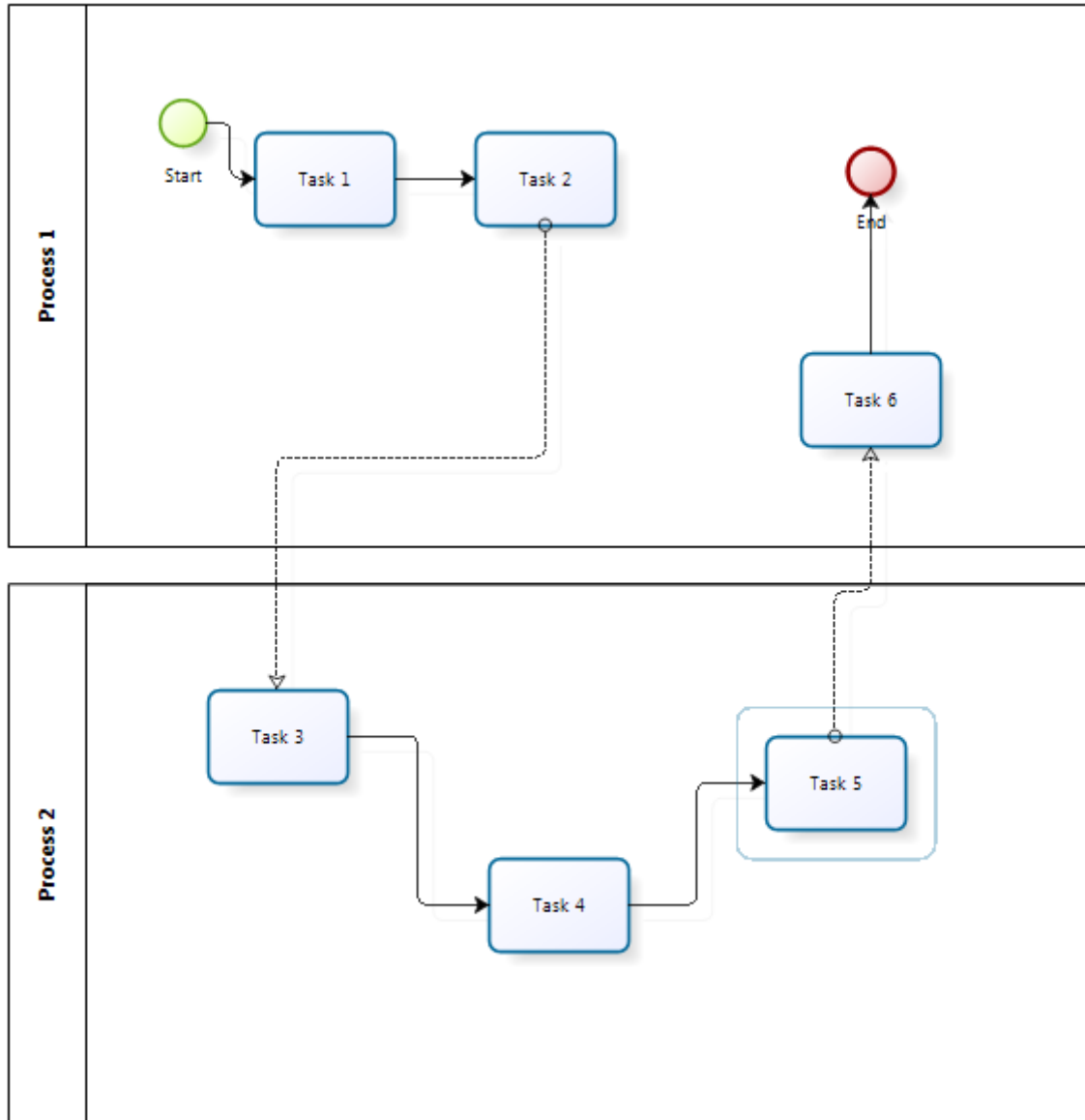
From a transition out of the XPDL file the Id is saved so that we can search for that transition. Also the from and to is needs, which are the Id's of the entities where the transition comes from and goes to.

6. Testing

The tool has continuously been tested while it was under development. There are few interesting smaller test cases which will be discussed in the following chapter.

First the capabilities of the parser must be tested. Testing if it can find all activities, also in special environments like multi pool, multi swim-lanes and sub-processes.

Consider the following multi pool process:



The output of the tool is:

```

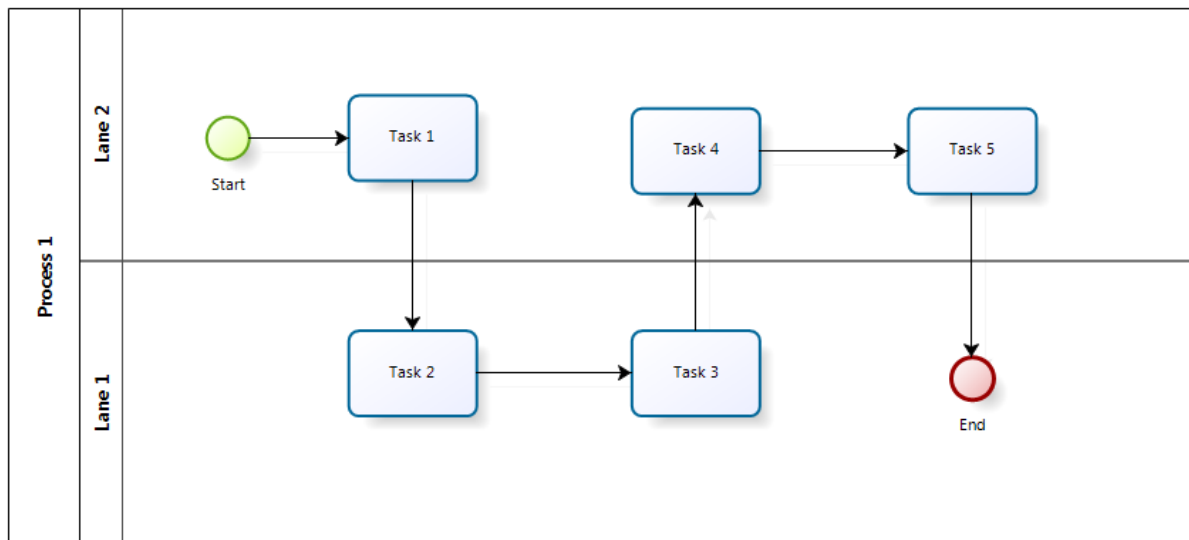
.-*!^*-.--*^WELCOME TO THE ORM BUSINESS DATABASE TOOL^*-.--*!^*-.
-----a text based prototype by David Otterbein-----

Please enter the full filename of the BPM .xpd1 file (e.g. <test.xpd1>)
multi-pool.xpd1
Amount of activities with a name: '8'
Activity 0: Start
Activity 1: Task 1
Activity 2: Task 2
Activity 3: Task 6
Activity 4: End
Activity 5: Task 3
Activity 6: Task 4
Activity 7: Task 5

```

So the parser first discovers all activities in the first pool and then in the other pool. This is only desired if there is no direct connection between the pools, in a case like the example it would be desired if the parser first traverses the connection independently. However all the activities are discovered correctly.

With the following multi-lane input:



Will yield the following output:

```

.-*!^*-.--*^WELCOME TO THE ORM BUSINESS DATABASE TOOL^*-.--*!^*-.
-----a text based prototype by David Otterbein-----

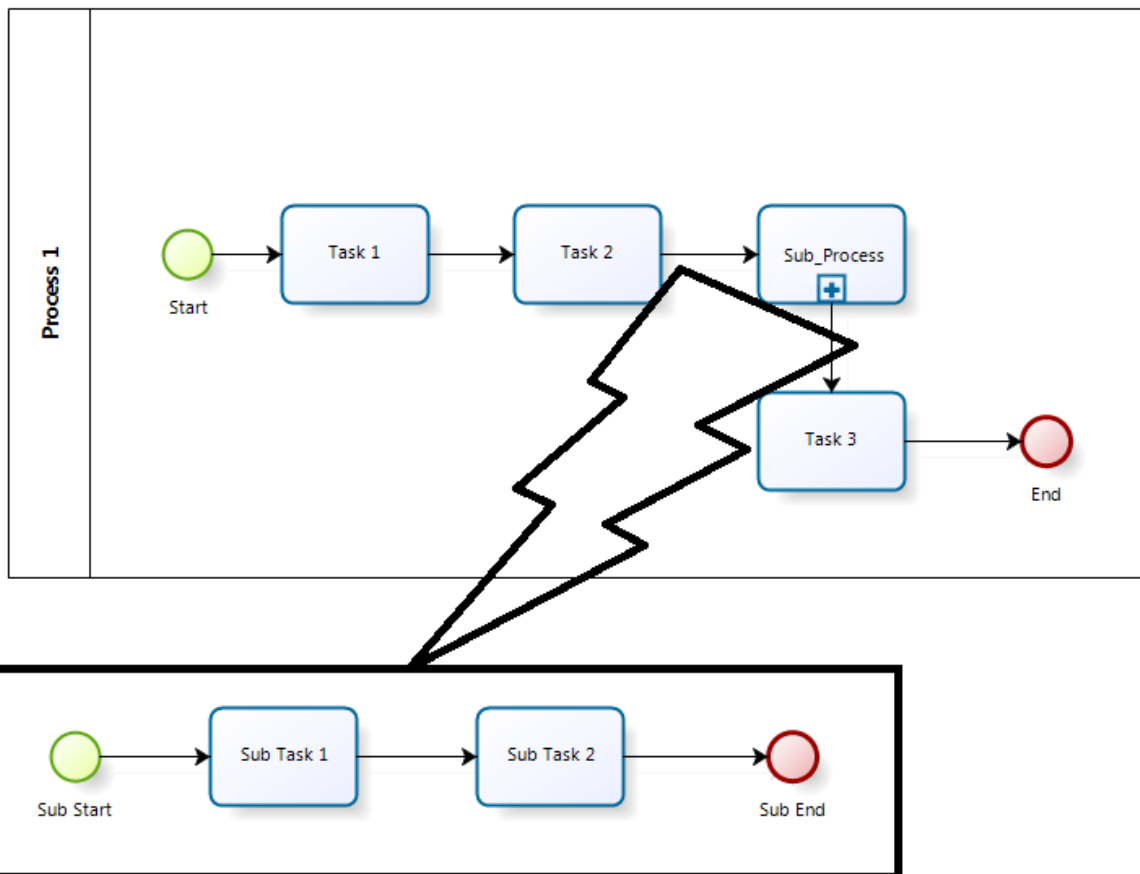
Please enter the full filename of the BPM .xpd1 file (e.g. <test.xpd1>)
multi-lane.xpd1
Amount of activities with a name: '7'
Activity 0: Start
Activity 1: Task 1
Activity 2: Task 2
Activity 3: Task 3
Activity 4: Task 4
Activity 5: Task 5
Activity 6: End

Enter <help> for help and commands
Enter all the elementary facts that correspond to the activity: Start

```

So the connections between the lanes are correctly read.

The last special input case which needs to be tested are sub-process diagrams. Consider the following diagram:



Will lead to the following output:

```

.-*^!^*-.--*^WELCOME TO THE ORM BUSINESS DATABASE TOOL^*-.--*^!^*-.
-----a text based prototype by David Otterbein-----

Please enter the full filename of the BPM .xpd1 file (e.g. <test.xpd1>)
sub-process.xpd1
Amount of activities with a name: '10'
Activity 0: Sub Start
Activity 1: Sub Task 1
Activity 2: Sub Task 2
Activity 3: Sub End
Activity 4: Start
Activity 5: Task 1
Activity 6: Task 2
Activity 7: Sub_Process
Activity 8: Task 3
Activity 9: End

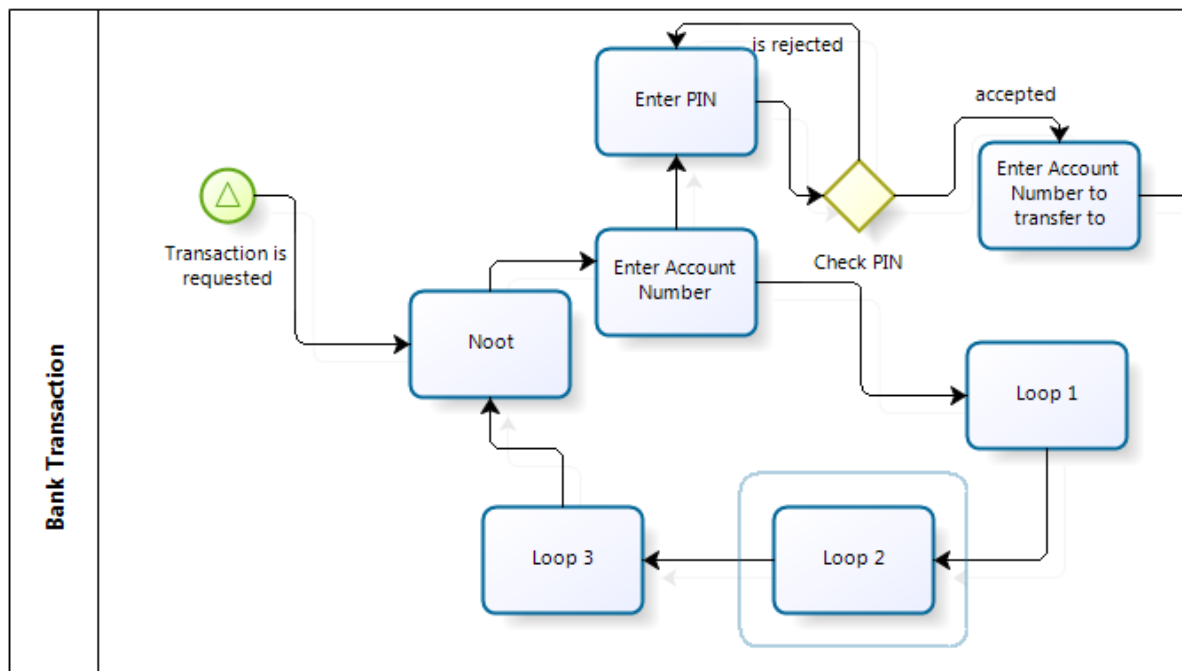
Enter <help> for help and commands
Enter all the elementary facts that correspond to the activity: Sub Start

```

The sub-process is listed first, but it is separated as a whole. After the sub-process ended, the main process is listed. So we can conclude that all activities had been discovered and the process and sub-process are listed separated.

Another important function which has been tested is the sort or order function. The function "orderActivities" as described in the code section under "Parser" orders all the activities in such a way a human would order them.

For example consider the following input BPM:



Normally after the activity “Enter Account Number” we would enter the loop before we discover the model any further. Assert that during the design process the activity “Enter PIN” and the path after it has been added first. Also the designer forgot the activity “Noot” and added this activity later. Then a straight forward order from the parser would be:

```

.-*^!^*-...^WELCOME TO THE ORM BUSINESS DATABASE TOOL^*-...*^!^*-
-----a text based prototype by David Otterhein-----

Please enter the full filename of the BPM .xpd1 file (e.g. <test.xpd1>)
loop.xpd1
Amount of activities with a name: '14'
Activity 0: Transaction is requested
Activity 1: Enter Account Number
Activity 2: Enter PIN
Activity 3: Check PIN
Activity 4: Check Debit
Activity 5: Enter Account Number to transfer to
Activity 6: Calculate new source balance
Activity 7: Calculate new sink balance
Activity 8: Transaction completed
Activity 9: Enter Amount
Activity 10: Noot
Activity 11: Loop 1
Activity 12: Loop 2
Activity 13: Loop 3

Enter <help> for help and commands
Enter all the elementary facts that correspond to the activity: Transaction is requested

```

But this is undesired because the activity “Noot” should appear second in the model and the loop should not appear in the end. If the “orderActivities” function is activated the ordered list of activities give a more natural way of traversing the model. Then from the same model the following list is generated:

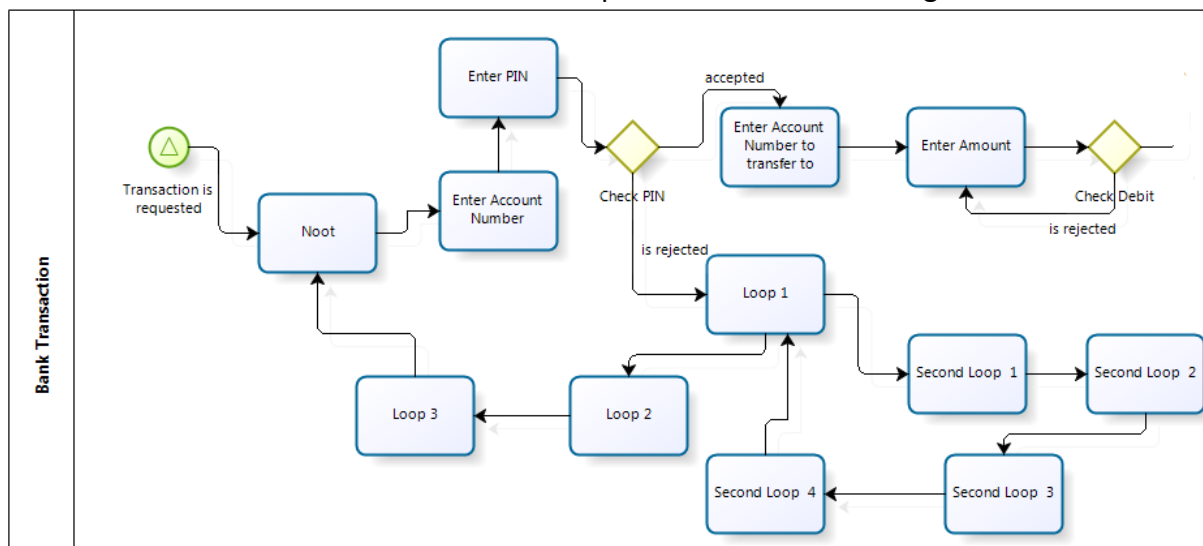
```

Welcome to the prototype of the BPMN to ORM tool
-----Business Databases tool-----

Please enter the full filename of the BPMN .xpd1 file (e.g. <test.xpd1>)
loop.xpd1
Amount of activities with a name: '14'
Activity 0: Transaction is requested
Activity 1: Noot
Activity 2: Enter Account Number
Activity 3: Loop 1
Activity 4: Loop 2
Activity 5: Loop 3
Activity 6: Enter PIN
Activity 7: Check PIN
Activity 8: Enter Account Number to transfer to
Activity 9: Enter Amount
Activity 10: Check Debit
Activity 11: Calculate new source balance
Activity 12: Calculate new sink balance
Activity 13: Transaction completed
Enter <help> for help and commands
Enter all the elementary facts that correspond to the activity: Transaction is requested

```

The order function also works with nested loops. Consider the following model:



In this model a second loop is nested into the first one. Without the order function the following activity list is generated:

```

.-*^!^*-.--*^WELCOME TO THE ORM BUSINESS DATABASE TOOL^*-.--*^!^*-.
-----a text based prototype by David Otterbein-----

Please enter the full filename of the BPMN .xpd1 file (e.g. <test.xpd1>)
loop-nested.xpd1
Amount of activities with a name: '18'
Activity 0: Transaction is requested
Activity 1: Enter Account Number
Activity 2: Enter PIN
Activity 3: Check PIN
Activity 4: Check Debit
Activity 5: Enter Account Number to transfer to
Activity 6: Calculate new source balance
Activity 7: Calculate new sink balance
Activity 8: Transaction completed
Activity 9: Enter Amount
Activity 10: Noot
Activity 11: Loop 1
Activity 12: Loop 2
Activity 13: Loop 3
Activity 14: Second Loop 1
Activity 15: Second Loop 2
Activity 16: Second Loop 3
Activity 17: Second Loop 4

Enter <help> for help and commands
Enter all the elementary facts that correspond to the activity: Transaction is requested

```

However with the function enabled the following list of activities is generated:

```

Welcome to the prototype of the BPMN to ORM tool
-----Business Databases tool-----

Please enter the full filename of the BPM .xpd1 file (e.g. <test.xpd1>)
loop-nested.xpd1
Be sure that the specified file is in the project folder!
C:\Users\David\Desktop\loop-nested.xpd1
Please enter the full filename of the BPM .xpd1 file (e.g. <test.xpd1>)
loop-nested.xpd1
Amount of activities with a name: '18'
Activity 0: Transaction is requested
Activity 1: Noot
Activity 2: Enter Account Number
Activity 3: Enter PIN
Activity 4: Check PIN
Activity 5: Loop 1
Activity 6: Second Loop 1
Activity 7: Second Loop 2
Activity 8: Second Loop 3
Activity 9: Second Loop 4
Activity 10: Loop 2
Activity 11: Loop 3
Activity 12: Enter Account Number to transfer to
Activity 13: Enter Amount
Activity 14: Check Debit
Activity 15: Calculate new source balance
Activity 16: Calculate new sink balance
Activity 17: Transaction completed
Enter <help> for help and commands
Enter all the elementary facts that correspond to the activity: Transaction is requested
_

```

Obviously the ordered list captures the flow of the model much more naturally.

Finally we can conclude that the basic functionality, that the prototype needs to provide, is working as intended.

7. Conclusion

The question was if it is possible to aid a BPMN user to build a corresponding ORM database for a business process model. This help should be provided by a tool which “converts a BPMN model to a ORM model by parsing the OLE user input”. After the tool worked properly on the basis of the “transfer case”, it has been tested elaborately. With the outcome of these tests the tool has been optimized in such a way, that it works with basic BPMN models and has enough functionality to aid non-ORM-experts to build a database.

It can be concluded that the tool-prototype proves the concept of aiding a user to build a ORM database with a tool to be possible.

8. Future Work

Extending the parser:

The parser has only been programmed to do what he needs to do. It could be extended so it can automatically searches the start signal (event), so it does not need to be added first to the model anymore. Also optional features like indicating which lane or pool an activity is in could be added.

Graphical User Interface:

The user interface is text-based. A graphical user interface would be much more appealing for most of the users and may make the tool even more accessible.

Group Testing:

Unfortunately there was not enough time left to test the tool in a group (5 to 20 persons). Based on the group testing small things may be redesigned or features could be added.

9. Bibliography

- 1: Dr. H Balsters. (2012, June). ORM Logic-based English (OLE) and the ORM ReDesigner tool: Fact-based Reengineering and Migration of Relational Databases.
- 2: Dr. H Balsters. (2012 Jan). Case for Lecture on Integration of Process- and Data modelling: Money Transfer from one Account to another.
- 3: Bizagi. (2012 May). Bizagi BPM Overview. Available: <http://www.bizagi.com/modeler/>
- 4: J.J. Pastoor. (2012, Jan). Databasemigratie en -normalisatie in ORM Logic-based English.

10. Appendix

I. ORM Business Tool user guide

Introduction

The ORM Business Tool is programmed in Java. The prototype of the tool was built as an bachelor project by David Otterbein for the RUG.

Purpose

The business database tool guides users through the process of making an ORM database based of a business model (BPMN) with ORM Logic-based English (OLE) as the user input format . The user enters the elementary facts from each step of the business model. This way the database model is extended with each step the user takes and finally put together to one big model. The OLE input is the basis on which the NORMA input structure is generated, so that the user can copy the NORMA input to easily create a DDL with the NORMA tool.


Target audience

This tool is developed for educational, research and business use. It should enable non ORM specialists to create a ORM DB by using ORM Logic-based English to describe the database. Because the tool is just a prototype, the given results (output of the tool) need to be further checked by hand.

How to start

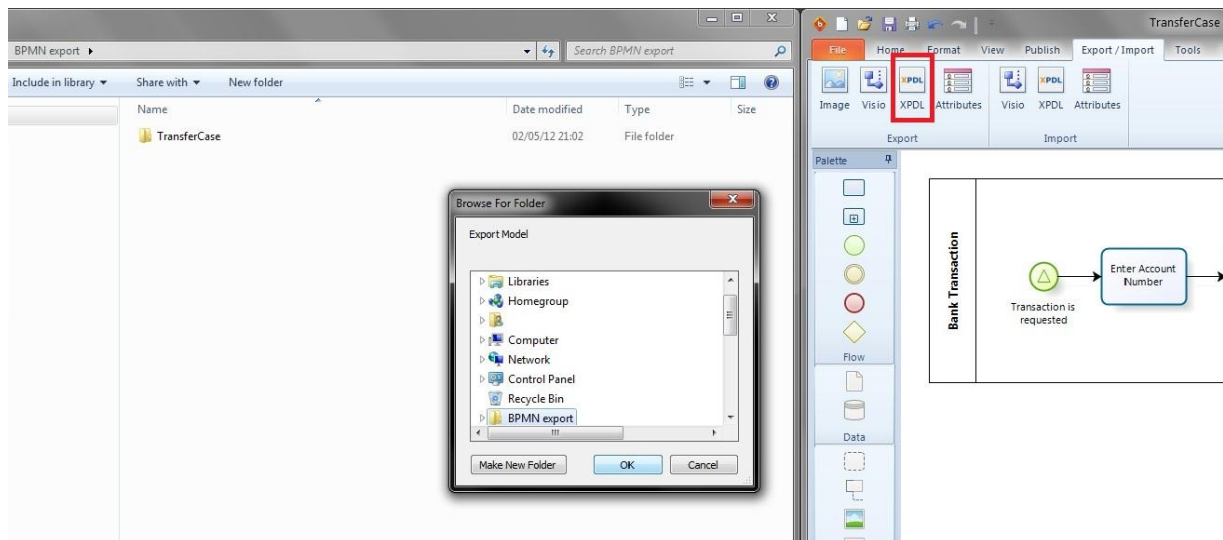
The tool is written in java. The prototype needs to be executed via command-line/console with the command `<java -jar BPMNtoORM.jar>`. Windows user can also execute the `BPMNtoORM.exe`. The .xpdI files, that you want to work with, need to be in the same folder as the jar/exe file. The program can be closed by simply closing the console or pressing `<ctrl+c>`.

Under Windows the console can be resized by right clicking the upper bar of the window. These are the recommended settings:



The image shows a screenshot of a Windows console window with three sections of settings:

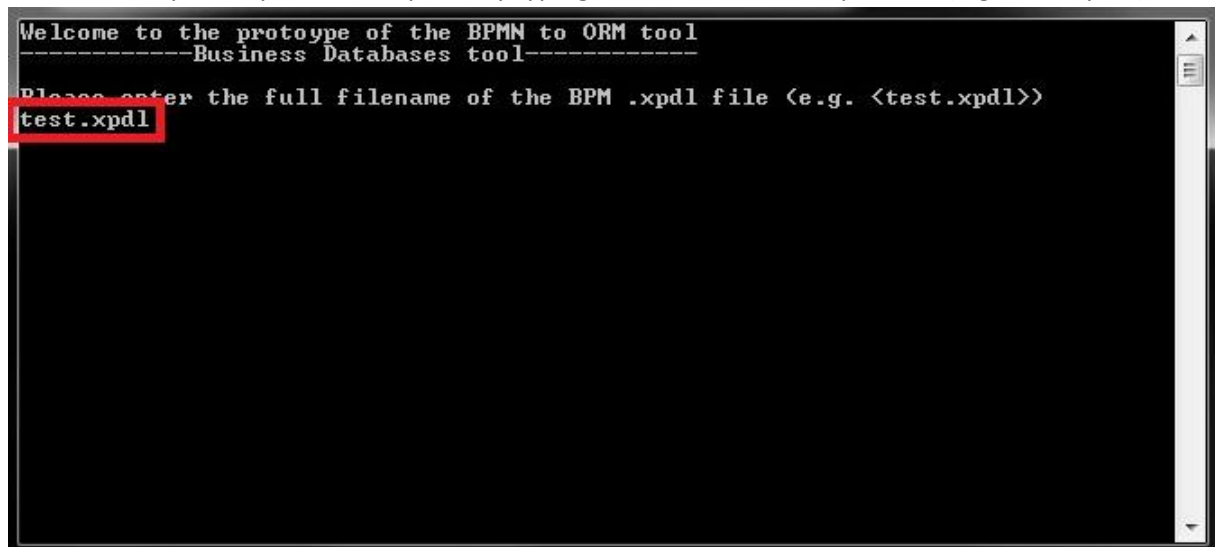
- Screen Buffer Size**
 - Width: 120
 - Height: 480
- Window Size**
 - Width: 120
 - Height: 50
- Window Position**
 - Left: 125
 - Top: 125
 - ☒ Let system position window



The xpd file can be exported like this. The file in the folder has to be renamed.

Select source

Choose the .xpd file you want to parse by typing the name into the input field (e.g. <test.xpd>)



Interpret source

Then the xpd file gets interpreted and the activities are ordered in such a way that the designer most likely would have traversed the model. It is important that the start event/node has been added first to the model, because the reference point in the xpd file is the first node which has been added. Then naming of all activities is very important too, because otherwise they will not appear in the activity list.

Define elementary facts

Define the elementary facts, which correspond to the Activity. In other words which elementary facts are needed to describe the database which lies underneath the activity. All the elementary facts should be defined in OLE.

Type in <next> to get to the next activity and <help> for help.

Define entities

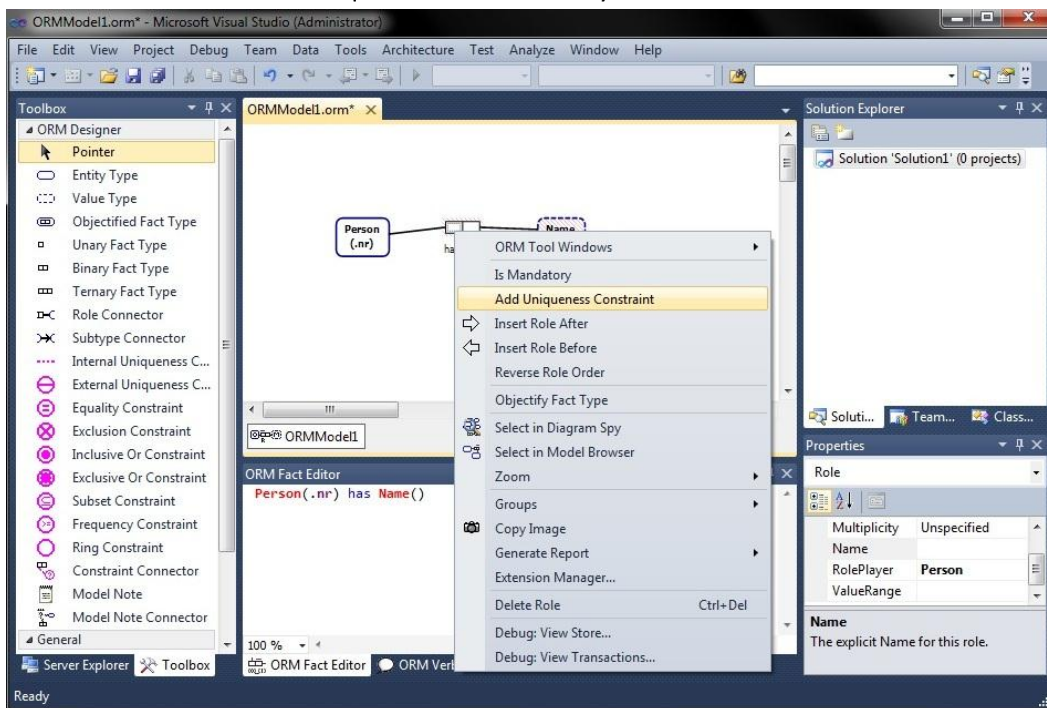
Define the reference type and reference mode for every entity. The entities are discovered from the elementary facts that has been entered. An external reference can be introduced by typing <external>. Then the reference needs to be formulated in OLE and will appear as a note at the end of the output. Type in <help> to view all supported reference modes.

ORM Fact editor

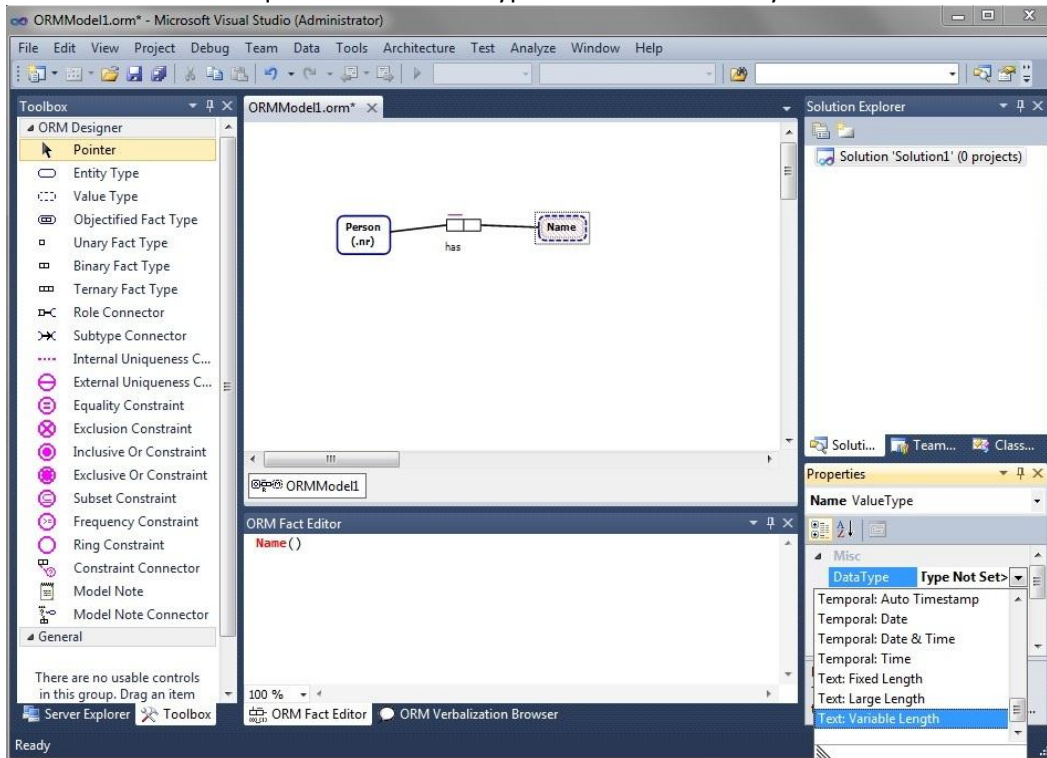
The output which is generated now has to be copied into the NORMA fact editor. This should be done one sentence at a time because the editor only parses the first sentence.

Define constraints

Unfortunately the NORMA fact editor does not support the text input of uniqueness constraints and mandatory constraints. They has to added by hand by right clicking on the desired side of the entity in the relation and add a uniqueness or mandatory constraint.



Some entities also require that the data type is entered manually.



External identifiers like they are mentioned in the notes of the output has to be added by adding an external uniqueness constraint. Double click on it so that a dotted line to the mouse pointer appears then double click on the first entity relation that should be us as an identifier. Likewise you can add more entity relations if needed. Then in the properties of the external uniqueness constraint the option “IsPreferredIdentifier” has to be set to true.

