Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

Satu Itäniemi

MASTER THESIS

An REA-Based Embedded DSL for Accounting Transactions

Submitted on 1 June 2016

Supervisor: Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander University Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

_____

Nürnberg, 1 June 2016

License

_____

Nürnberg, 1 June 2016

Abstract

Domain-specific programming languages (DLSs) are widely used and are popular for mainly two reasons: improving productivity for developers and improving communication between developers and domain experts. Also, domain experts with little or no programming knowledge can program using DSLs with little or no external support. To achieve a successful representation of a domain, the involvement of both the domain expert and the developer is important. However, this thesis uses an ontology based approach to designing and implementing a small, embedded DSL that allows the user to record basic accounting transactions and extract financial information. The ontology used is the REA (Resources- Events-Agents) accounting ontology. REA is a well-established conceptual framework in the field of accounting and is often used to analyze economic processes and functions.

Keywords: eDSL, REA Model, Accounting, Accounting Transactions

Contents

1 Introduction

1.1 Original Thesis Goals

The original thesis goals included an embedded DSL in the Grace language, which was then to be run on top of the Sweble engine. Grace is a relatively new object-oriented programming language targeted at new programmers (Noble et al., 2013). Sweble is a wiki-like domain expert programming system that serves as a host to different DSLs, providing them with an object model, transactions, and persistence (Dohrn, Riehle, 2011).

By using the DSL, the user should be able to create new REA entities that map to the REA ontology and follow the core axioms of the ontology. In addition to creating REA entities, the user can execute and record new transactions, which are then used as the basis for extracting financial information.

1.2 Changes to Thesis Goals

Due to time issues and complexities of a new language that is still under development, it was decided that Sweble and Grace would be dropped, and the thesis implemented in another host language, C#.

2 REA eDSL

In contrast to general purpose languages (GPLs), a domain-specific programming language is suitable for a narrow purpose and domain. DSLs are often highly constrained and declarative, and can be either textual or graphic. A textual DSL can be either external or internal (embedded) – an embedded DSL is represented within the syntax of a host language, normally a general-purpose language. The constructs of the host language are then used to implement the DSL. An external DSL is usually written in a separate language. A script in an external DSL is commonly parsed by a code in the host application using text parsing techniques. Using the approach of the internal DSL is often faster but introduces limitations as the DSL has to follow the rules of the host language (Fowler, 2010).

When developing a domain-specific language based on REA, most related work is on REA related topics. However, according to the best of my knowledge, there is currently no existing research on textual DSLs utilizing the REA model. Most of the research is in fact in modeling extensions and improvements to the core model to support businesses in management decision making, enterprise value chains and AIS (Accounting Information System) system development (Geerts, McCarthy, 1999; Geerts, McCarthy 2002; Verdaasdonk, 2013).

In addition to modeling extensions, there is some work on REA based visual domain specific languages (Mayrhofer et al., 2012 Sonnenberg et al., 2011) that allow the user to conceptualize their business models and possible AIS, and allow for code generation from the created models (Sedbrook, 2012).

The ontology is also the basis for open-EDI standards that support cooperation between trading partners, and there have been efforts to formalize it within an OWL (Web Ontology Language) representation (Gailly, Poels, 2007, 2008). Some of the REA concepts have also been used in other electronic business interchange formats, such as ebXML and UN/CEFACT (Geerts, McCarthy, 2001).

The objective of this thesis is to apply the core concepts from the REA model in designing and implementing a small, embedded DSL in C# that is intended for the use of an accountant with some programming experience.

The motivation for choosing REA is that being a well-established accounting ontology, it provides an excellent foundation for accounting domain specific use, especially in the case of this thesis where the communication with a domain expert is missing. Even though the REA model is applied, the goal is to respect the traditional and accepted accounting principles. Using the DSL, the user can record new accounting transactions without having to use double-entry bookkeeping and extract basic financial information.

However, neither the underlying model nor the DSL is a complete REA representation as that kind of work is out of scope, but the work can be extended to cover larger business models and different types of transactions.

The thesis structure is as follows: Firstly, the core REA model is described in how it relates to the DSL, and the key elements and rules are identified. Basic accounting information and how it relates to REA is also briefly covered. In the third chapter, the architecture and design are covered, and in the fourth chapter, the DSL operations are outlined, and examples are given on how to create new transactions and retrieve financial information. In the fifth chapter an evaluation and discussion of results is given.

2.1 The REA Model

The fundamental concepts of the REA ontology are Resources, Events, and Agents, and together with Commitment and Contracts, they form the "core" REA model. In short, the idea of the model is to represent a generic trading pattern: *economic events* involve *economic resources* and are driven by *economic agents*. These are connected by *stockflow*, *participation* and *duality* relationships as shown in Figure 1. For the remainder of the thesis, the word "economic" will be dropped for easier readability. Furthermore, the REA model is divided into two groups: the operational level and the policy level. The operational level is the basic trading pattern of resource, event, agent, and deals with "what has happened". The policy level has concepts for dealing with "what could, should, or should not happen" – semantic abstractions like scheduling, grouping, contracts and others are included. There are several patterns to deal with different situations (Hruby, 2006), but they are focused on solving specific economic areas, which is why it is not applicable to address all concepts here. The thesis focuses on the core trading pattern, together with commitment and contract.

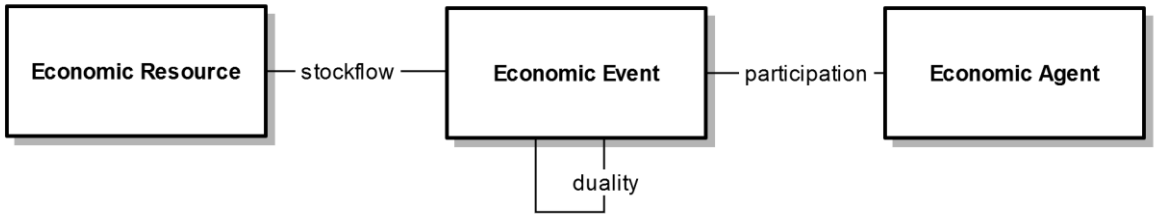


Figure 1: REA Core Model

An event can either increment or decrement a resource, for example receiving money from a customer increments the quantity of money on hand, and giving away a product decrements the quantity of product on hand. These events are paired in a *duality* relation and form an *exchange*, or transfer. The exchange is a fundamental concept of economic activities in the REA model (Hruby, 2006). This means that

To better illustrate this, we give an example of Steve´s Bike Shop in a sales process. This process of selling a bike to a customer is essentially just an exchange of money for a bike; Steve´s Bike gives a bike to the customer and receives cash in return – this represents an *outflow* of the bike and an *inflow* of cash. This is the give and take nature of the REA model. In this exchange, the customer and Steve´s Bike Shop are *agents*, while cash and bike are the *resources*. The *events* are sale and payment. In the sale event, Steve´s Bike gives a bike to the customer (a transfer of ownership); this is a transaction of Steve´s bike providing the bike, and the customer receiving it. Looking from the point of view of Steve´s Bike, the sale is a decrement event, because it decreases the value of resources that the shop owns – the number of bikes in the inventory, and therefore the value of the inventory decreases. The payment (or cash receipt) event is similarly a transaction of customer providing money, and Steve´s Bike is receiving it. Again, from the point of view of the bike shop, the payment event is an increment event, because it increments the amount resources the shop has – in this case, cash.

The terms are always dependent on the model viewpoint, and depend on the agent, e.g. here we are only interested in the standpoint of the company, not the customer – for the customer the sale event would be an increment, and the payment event would be a decrement.

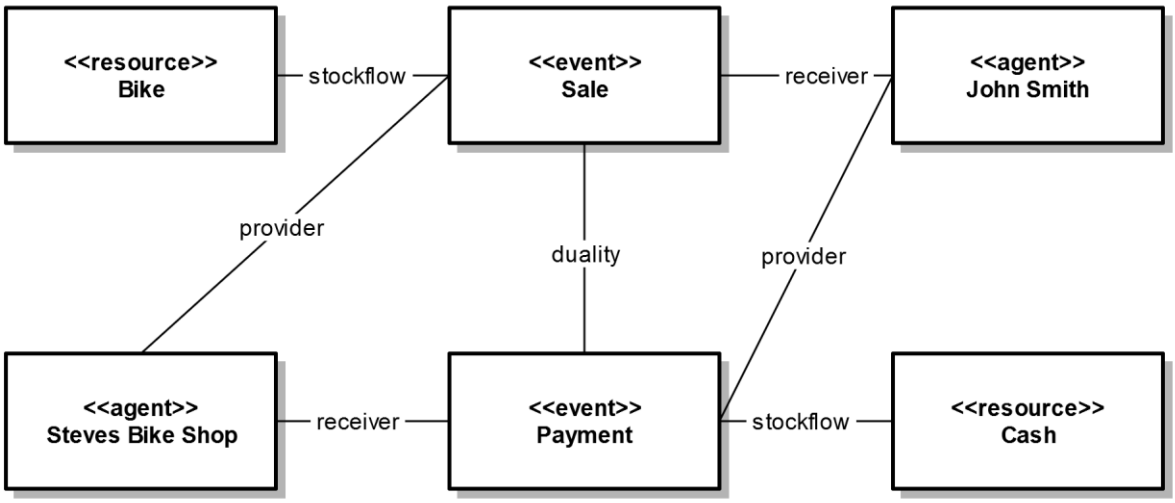


Figure 2: REA Exchange (Steve´s Bike sales process)

This basic pattern of economic exchange has been extended by the concept of *commitment (*Geerts, McCarthy, 2000a): a promise to fulfill a future event, e.g. line items on a sales order represents a commitment to sell goods. Structurally similar to events, increment and decrement commitments can be *reciprocal* like events are dual. The relationship between commitment and an event is called *fulfillment*, and commitment is connected to a resource through a *reserve* relationship, e.g. a promise to sell certain goods in the future. Agents are connected to events and commitments through the *participation* relationship (Hruby, 2006).

Commitments make up a *contract*, which is essentially a collection of increment and decrement commitments. A contract can specify what should happen if the commitments are not fulfilled, e.g. a sales order is a contract that contains commitments to sell goods, and receive payments. The terms of the sales order could specify additional commitments as penalties if the goods or the payments have not been received as promised (Hruby, 2006).

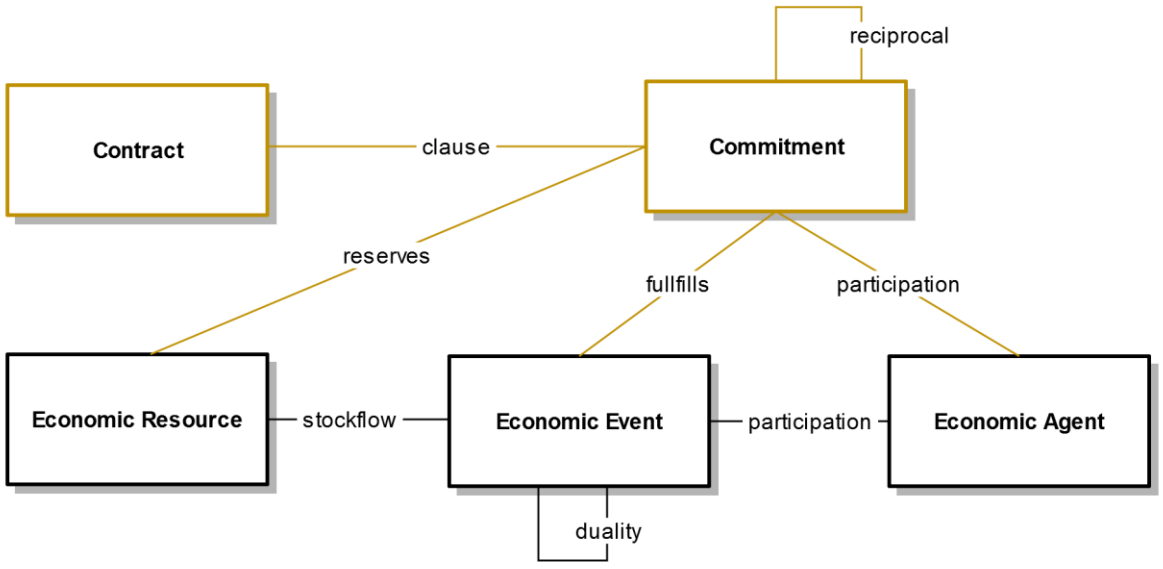


Figure 3: Extended REA Model

As an example of how the commitment works, I give an example with a more detailed sales process with an order, specifying a commitment of Steve´s Bike to sell a specific bike to a customer, and in return, a commitment of the customer to pay a specified amount of cash for the bike. The sales order is a contract between the agents Steve´s Bike and customer.

The sales line and payment line are commitments to perform events at a specified time in the future. The sales line is a commitment to carry out the event sale, and the order line is a commitment to perform the event payment. Realistically speaking, the sales order would contain terms to handle cases where the commitments are not fulfilled, for example when payment arrives late, or there is something wrong with the bike. The relationships are explained in detail in *2.1.1 REA Key Elements and Axioms.*

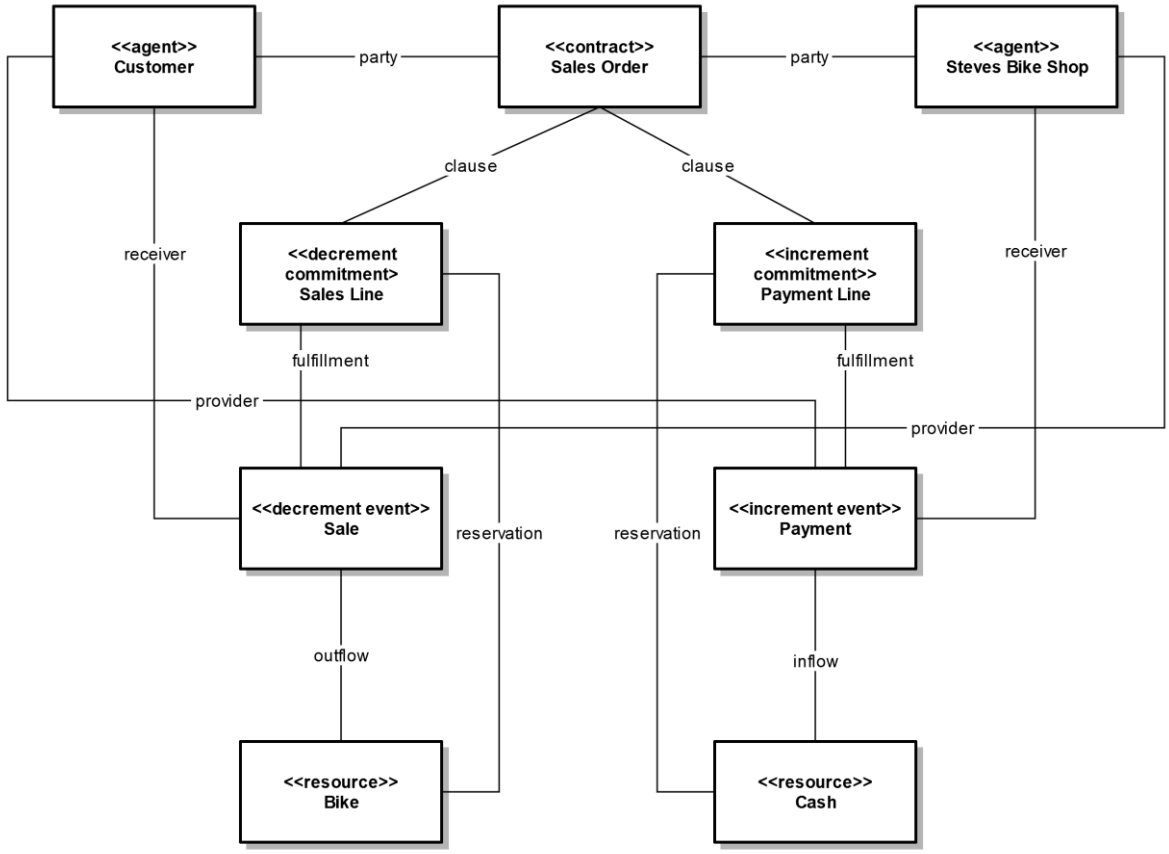


Figure 4: Sales process with Commitment (Steve´s Bike)

In addition to the *exchange*, REA also covers a *conversion* process in which a company uses or consumes resources to produce or modify resources. However; the conversion process is not addressed in this work. The data that is utilized in the exchange and conversion is the business data of a company, and the traditional accounting concepts of debit, credit, ledgers, receivables, balances and journals are derived from the underlying REA model describing the exchanges (and conversions). In contrast, in traditional accounting, this information would be derived using *double-entry bookkeeping* (Hruby, 2006).

2.1.1 REA Key Elements and Axioms

Basic operational level with the relevant policy level concepts contains the following (Meliš et al., 2014), (Hruby, 2006):

Basic operational level concepts:

A resource is a thing or an object providing utility for an agent and is something that a company wants to monitor, plan and control. Products, services, money, labor and raw materials are examples.

An event represents either an increment or a decrement in the value of a resource that a company holds. Some events happen instantaneously, like sales of goods. Other events happen over time, such as rentals, using service and acquiring labor.

An agent is an individual or a company that has control over resources and is capable of transferring ownership of resources to another agent. Examples are customers, employees, vendors, and companies.

Relevant policy level concepts:

A commitment is a promise or obligation to perform some event at a specified time in the future. Commitment should include a scheduled date and an expected resource value.

A contract is a collection of commitments and terms.

Typification is a collection containing entities that are alike, with same characteristics defined by the type, forming “is a –kind- of” bond.

Relationships and domain rules

In addition to the basic entities, REA model includes relationships and domain rules (axioms) that should be followed; normally these relationships are applied as multiplicity constraints (Meliš et al., 2014). The relevant relationships and rules are described in Table 1.

Even though there are several possible relationships and axioms, there are three core rules that should be fulfilled whenever a commitment or an event happens. These are the *duality*, *stockflow* (inflow and outflow) and the *participate* relationship (ownership of a resource; an agent is only allowed to transfer ownership of resources that are under their control) axioms.

Relationship	Relation	Axiom
Stockflow (inflow/outflow)	Connects **resource** and **event (increment, decrement)**	There must be one event for every resource. There must be at least one resource for every event.
Participate(provider/ receiver)	Connects **agent** and **event**	An event must have two agents, one receiver, and one provider. An agent must be connected to an increment event by the receive relationship and to a decrement event by the provider relationship.
Duality	Connects **increment** and **decrement** event	Every increment event must be paired with at least one decrement event. Every decrement event must be paired with at least one increment event.
Fulfillment	Connects **commitment** and **event**	Every increment commitment must be connected with an increment event. Every decrement commitment must be connected with a decrement event.
Reciprocity	Connects **increment** and **decrement commitment**	Every decrement commitment must be connected with at least one increment commitment. Every increment commitment must be connected with at least one decrement commitment.
Clause	Connects **commitment** and **contract**	Every contract must contain decrement and increment commitment. Every commitment must be declared by some contract.
Party	Connects **contract** and **agent**	Each contract must have two parties(agents).
Reservation	Connects **commitment** and **resource**	Each commitment must be connected to a resource Each commitment must be connected to agents through the participate relationship (provide, receive).

Table 1: REA Relationships and Domain rules

2.1.2 Traditional Accounting and Motivation for REA

In 1982, McCarthy described in his paper "*A Generalized Framework for Accounting Systems in a Shared Data Environment*" issues with traditional accounting methods and proposed REA as a model to overcome these. The first issue that McCarthy described was that the data that is available in AIS is only monetary, which wasn´t helpful for analyzing results or making financial decisions (McCarthy, 1982). What this means in practice is that it was not possible to track where the money was coming from; e.g. an increase in total sales could be the result of several factors such as sales volume, price increases and so on. With traditional accounting methods, it is only possible to see that there is an increase, but not the reason for it. REA model tracks the flow of resources, and data is registered on a deeper level. For example, when a company sells resources, all the details of the event are stored such as customer name, the date when the event took place, the value of the sales order, which resources, how many and so on, depending on the exact requirements. All of this information is then used to extract meaningful financial information such as income statements, balance sheets, sales, cash flow statements and so on.

Another issue McCarthy mentioned was that traditional AIS only store the result of an event (the monetary value of an inflow/outflow) – this is related to the first weakness (McCarthy, 1982). It should be possible to base decision making on more than just the monetary value of events. McCarthy suggested that the data be kept in as simple form as possible to be aggregated by the user later.

The increment and decrement structure works well for physical resources and cash as depicted in the REA model. However, when it comes to debt and equity, the model loses some of its intuitiveness; for example, in a debt financing transaction, we have two increment events because the loan stock of the company increases, and at the same time, the inflow of cash increases the cash stock. Also, equity is not a transaction with a counterpart and claim to an entity, but rather it represents the owner´s interest in that entity (Schweiger, 2015).

This scenario clearly violates the REA axioms as we covered earlier. One possible way to handle this situation would be to model the financial transaction as a contract, which includes future payments as commitments that make up the contract.

2.1.3 Accounting methods

Traditional accounting starts with the accounting equation " Assets = Liabilities + Owner's Equity/ Stockholder's Equity" (Averkamp, n.d). This equation states that the resources of a company are assets and the claims to those resources are liabilities and equity. The equity is the owner´s claim to the net value of the enterprise, calculated as the difference between assets and liabilities. The assets, liability and equity govern how the increases and decreases are accounted - increases are recorded on one side, and decreases on the other side using T-accounts (Horngren, 2012):

Assets	Liabilities and Owner´s Equity
Increase = Debit | Decrease = Credit	Decrease = Debit | Increase = Credit

Figure 5: Rules of Debit/Credit

It depends on the type of the account if it is increased or decreased by a debit or credit.

The most common reports any company needs to produce are an income statement, a balance sheet, a cash flow statement, a list of open invoices, and a VAT (value-added tax, similar to sales report). These form the core accounting functionality of most AIS and are also a legal requirement. The thesis covers the income statement and the balance sheet, but the work can be extended to include the other types of reports as well. For the income statement, single-step income statement format is used:

Net Income = (Revenues + Gains) – (Expenses +Losses)

An income statement shows the profitability of a company during a specified time range; it covers revenues, expenses, gains, and losses, but it does not show cash receipt or cash disbursements. An income statement usually covers a time range of starting date of the company to a specified end date (most often the statement is generated annually) should be noted that revenues are different from receipts, they occur when a sale is made or when they are earned; receipts occur when the cash is received or collected. Revenues are often recorded before receiving the cash. (Averkamp, n.d.)

```
            Steves Bike
          Income Statement
    For Period 01.01.2015 - 16.05.2016
**************************************************
    Revenues
    Sales                   100000
    Investments             0
    _____________________________________________
    Total Revenues          100000
    _____________________________________________
    Expenses
    Cost of Goods Sold      50000
    General                 50000
    _____________________________________________
    Total Expenses          100000
    _____________________________________________
    Net Income              0
```

Figure 6: Dummy Income Statement – Steve´s Bike

A balance sheet summarizes assets, liabilities and owner´s equity at a specified point in time. The balance sheet should always satisfy the accounting equation described earlier:
Assets =Liabilities +Owner´s equity.

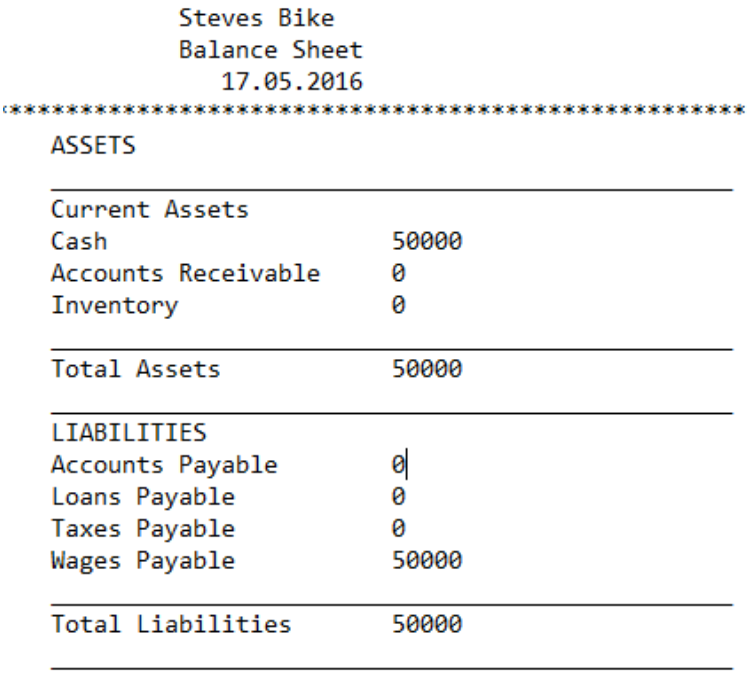


Figure 7: Dummy Balance Sheet –Steve's Bike

To generate these reports, events that affect them have to be captured – for example, events that transfer a resource (a physical thing or money) from one agent to another. Receiving a resource can be something that a company has purchased, for example, property or inventory or something that a company sells for profit, like a bike. There are other cases as well, for example, resources that are consumed as soon as they are received, such as a consultation. These are out of the scope of the thesis. In short, events that affect the income statement are r*evenues, the cost of goods sold* and *fixed costs*. For the balance sheet, the events are for example *cash, accounts receivable, fixed assets, finished goods, bank account, accounts payable, VAT payable* and *Owner´s equity*. Raw material events are left out as production is not covered.

For example, relevant events (and event pairs):

- Receive 2 Bikes from Vendor X
- Receive 10 Bikes from Vendor Y
- Deliver and receive payment for bike X for 2000€ incl. VAT
- Receive from A 10000€ into the cash account
- Order more inventory from Vendor X
- Deliver and invoice X for an order of 10 bikes, to be paid later

The details on how to generate these statements are given in Chapter 4, Implementation.

3 Architecture and design

The DSL uses a combination of techniques; fluent interfaces, method chaining and extension methods .The fluent interfaces are used with method chaining. In addition, the DSL allows the user to create new REA entities at runtime, and for this, a generic fluent factory is used. There are different builders for various parts of the DSL – the generic fluent factory, plus builders for the transactions and some native query operations. For the reports, there are extension methods.

The main goal was to follow the language of an REA transaction. In order to successfully record the transactions, the user should have a basic understanding of REA and the way transactions work in it; that there are always two agents, a resource and increment and decrement events, and possibly commitments. Before I go into more details on the design of the DSL, the construction of the model upon which the DSL is built on is described.

3.1 Domain Model

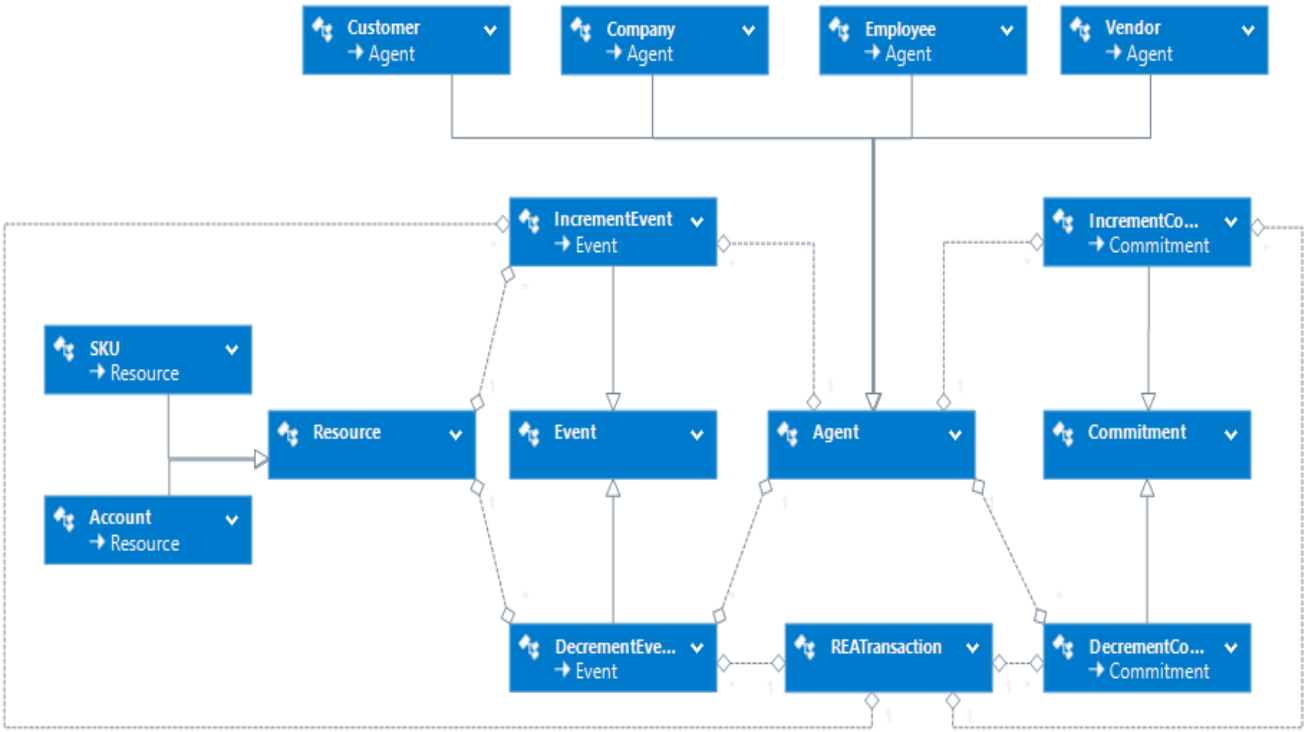


Figure 8: Domain Model (with concrete entities)

In the domain model above, we have the core REA entities as abstract classes, and some concrete entities relevant to the business case of Steve´s Bike. The *Customer*, *Company*, *Employee*, and *Vendor* are all agents participating in the increment and decrement commitments and events, while SKU represents an item that a company is interested in selling, and the account represents Cash. The REATransaction represents a complete transaction of event pairs. The properties vary based on business scenarios, but there are some that should always be present in an REA model. The *Agent* has an Id, a Name, and an Address; the *Resource* has an Id, Name, and Value, and *Event* has Id, Date, EventType and

Amount. The entities are associated with inheritance relationships and foreign and primary key constraints. As such, the model also represents the relational structure of the REA model, and should apply to different types of revenue creating business scenarios.

As mentioned earlier in the paper, manufacturing processes are not covered by this model. The model was created with the ADO.NET Entity Framework. The entity framework and the model and its creation is covered in more detail in the following sections.

3.2 Entity Framework

It is possible for the user to create new REA entities at runtime, but at this time, no database is generated. Additional functionality is required to map the generated entities to a database. However, to be able to demonstrate the REA model and the DSL, the data model was implemented using the data model in SQL Server 12 using ADO.NET Entity Framework. Simply put, the Entity Framework is an object-relational mapping library, and it provides a framework that lets developers work with relational data as domain-specific objects, which is why it is particularly useful for REA (which is often represented in a relational form). In the domain model above some of the foreign key constraints for REA are visible.

The Entity Framework supports the Entity Data Model (EDM) used for defining data at the conceptual level (Microsoft, 2011). The conceptual model, storage model, and mapping information are contained in a .edmx file. There are three main approaches to using the entity framework with the EF Designer and Code first. The EF designer can be used to define a model at a conceptual level (Model first), and the database is then generated based on the drawn model. Another option is to use the designer to map to an existing database (Database first). Finally, the code first approach works in a similar manner – the developer defines the model in code, and the database is then generated based on the code. Similarly, the code first approach can be used with an existing database.

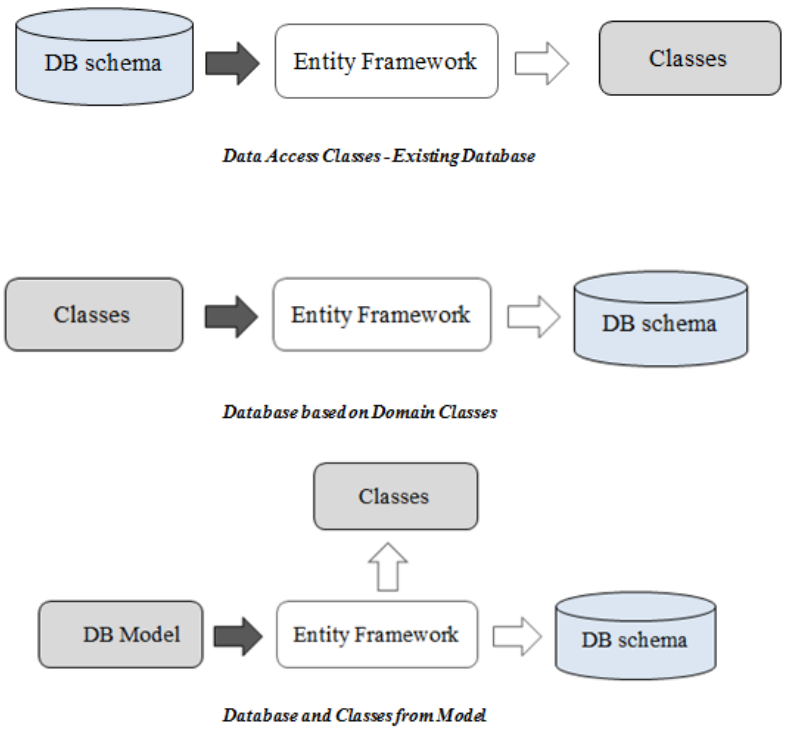


Figure 9: Entity Framework

For the REA model, I used the Database first – Existing Database approach as I had already built the database. Depending on the Visual Studio version, the user might have to upgrade to the latest version of the Entity Framework. The most recent version is available on NuGet:

- **Project -> Manage NuGet Packages**… (if this does not exist, install the latest version of NuGet)
- **Select Online tab**
- **Select the Entity Framework package**
- **Click Install**

There is an Entity Data Model wizard in Visual Studio that creates the .edmx file based on an existing data connection. Once the process is completed, the new model is added to the project and opened up in the Entity Framework Designer, and an App.config file containing the connection details for the database is created and added to the project. However, the visual designer only shows entities that are mapped to the database tables and views. The Model Browser gives a bigger picture, and it contains all the information about the EDM, the conceptual model, storage model, and mapping information.

Model browser:

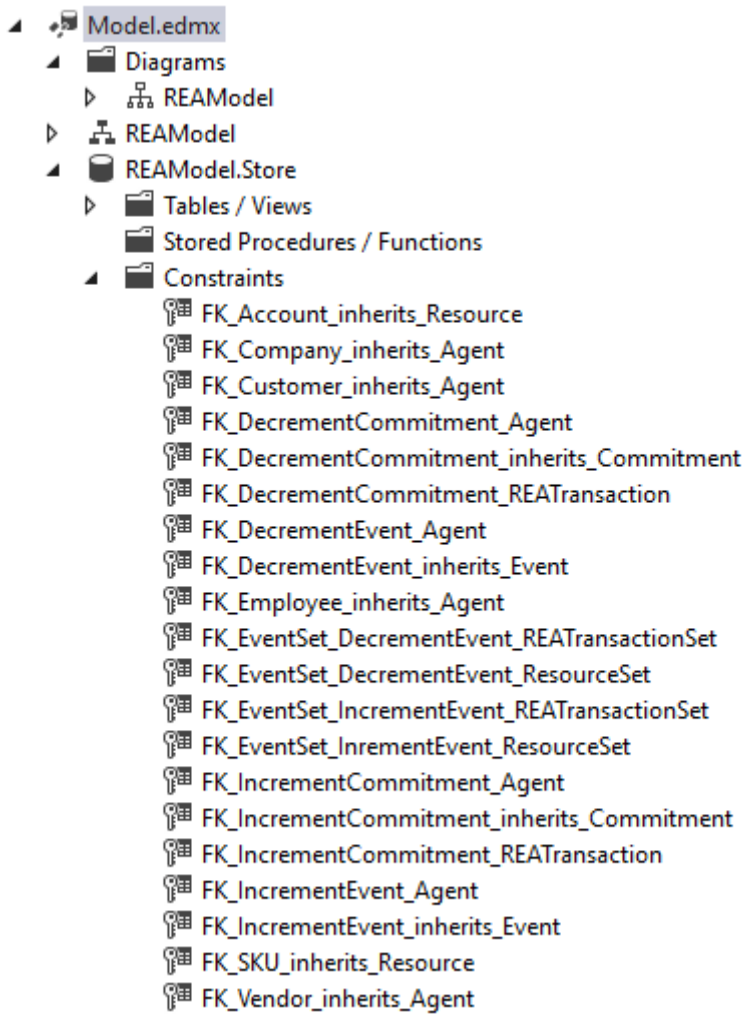


Figure 10: REA Data Storage

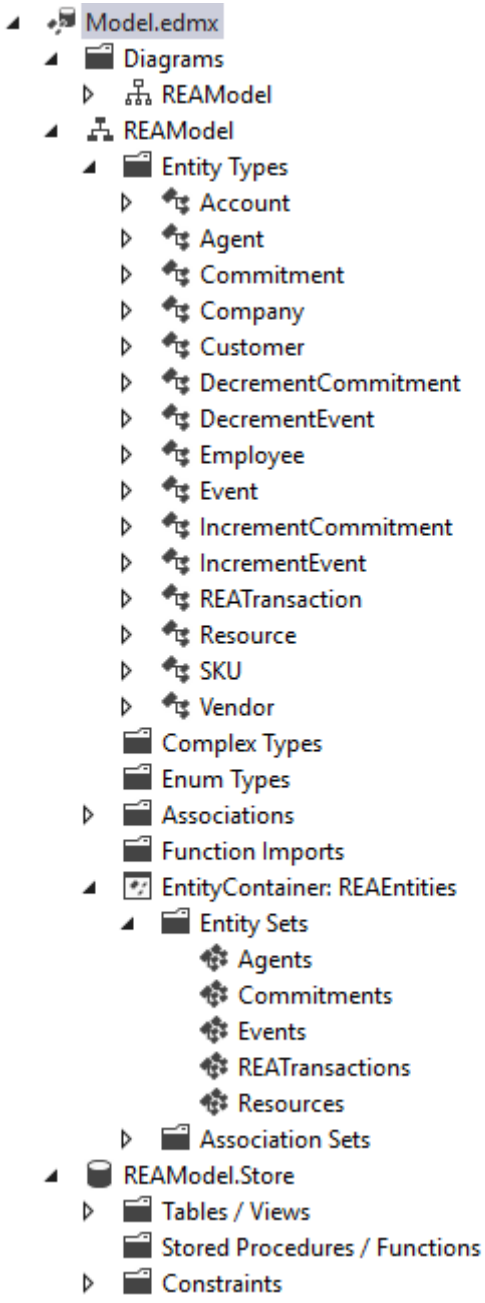


Figure 11: Entities of the REA model

The diagrams section contains the visual diagrams drawn or generated with the designer. The Entity Types lists all the classes that are mapped to the database tables: *Account, Agent, Commitment, Company, Customer, DecrementCommitment, DecrementEvent, Employee, Event, IncrementCommitment, IncrementEvent, REATransaction, Resource, SKU, and Vendor.* The complex types are classes that are generated by EDM and contain results of stored procedures and functions; at this point, there are none in the REA model. Enum types are all the entities which are used as Enum in the entity framework, of which there are none at this point. The associations is a list of the foreign key associations between entities. The Function Imports lists all functions mapped to stored procedures etc. The EntityContainer is a logical grouping of entity sets, association sets, and function imports. An entity set is a container for instances of an entity type, and instances of any type derived from that type. The relationship is like that of a row and table in a relational database: the entity type describes data structure and the entity set contains instances of the given structure. The entity set is not used to model data, but it provides a construct for a storage environment like SQL Server Db to group entity type instances so that they can be mapped to a data store (Microsoft, 2011).

The .Store shows all tables and views in the database, as well as stored procedures, functions, and constraints. The constraints for the REA model are visible in *Figure 11*.

The EDM also creates a class for the entities, in this case, called REAEntities. This class derives from System.Data.Entity.DbContext, and is referred to as the context class in the entity framework.

```
namespace REA_Dsl
{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Infrastructure;

    public partial class REAEntities : DbContext
    {
        public REAEntities()
            : base("name=REAEntities")
        {
        }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            throw new UnintentionalCodeFirstException();
        }

        public virtual DbSet<Agent> Agents { get; set; }
        public virtual DbSet<Commitment> Commitments { get; set; }
        public virtual DbSet<Event> Events { get; set; }
        public virtual DbSet<Resource> Resources { get; set; }
        public virtual DbSet<REATransaction> REATransactions { get; set; }
    }
}
```

This class acts as a bridge between the domain and the database and represents a combination of Unit of Work and Repository patterns – meaning that it can be used to query a database (LINQ to Entities) and group together changes that will be written back to the data store as a unit. (Microsoft, 2011).

The DbContext is typically used together with `DbSet<TEntity>`, the entity set for all entities mapped to the database, and here we have entity sets for Agents, Commitments, Events, Resources, and REATransactions. The: `base("name=REAEntities")` refers to the connection

string. The constructor loads the connection string from the configuration file and ensures that if it is not found an error will be thrown rather than creating a new database. The 'name =' syntax is important when using a connection string, or there might be databases all over the place. Mapping in the entity framework depends on the approach taken. The `OnModelCreating(DbModelBuilder modelBuilder)` method can be used to set up relationships. For example, if I had taken the Code first approach, I could map an entity type to a specific table:

```
modelBuilder.Entity<Customer>().ToTable("Customer");
```

Another key point to consider in the entity framework is *inheritance*. For the REA model to function on a more generic level, inheritance was important. The EF provides three ways to implement inheritance: Table – Per- Hierarchy (TPH), Table-Per-Type(TPT) and Table-Per-Concrete class. There are advantages and disadvantages to all three; in TPH all types in an inheritance hierarchy are mapped to a single table, and a specific column called discriminator is used to differentiate the types. In Code first, this is the default. In TPT, all types are mapped to individual tables, and tables that map to derived types also store a foreign key that joins the derived table with the base table. In TPC, all non-abstract types are mapped to individual tables. The inheritance for the REA model is done using the TPT.

The advantages of TPT are that the SQL schema is normalized, and to modify the base class or to add a new subclass only one class has to be changed or modified. The disadvantage is that complex queries will result in a lot of joins.

The process is quite straightforward, and can be done via code or the designer. Because I already had the model with the foreign keys, I did the mapping in the designer.

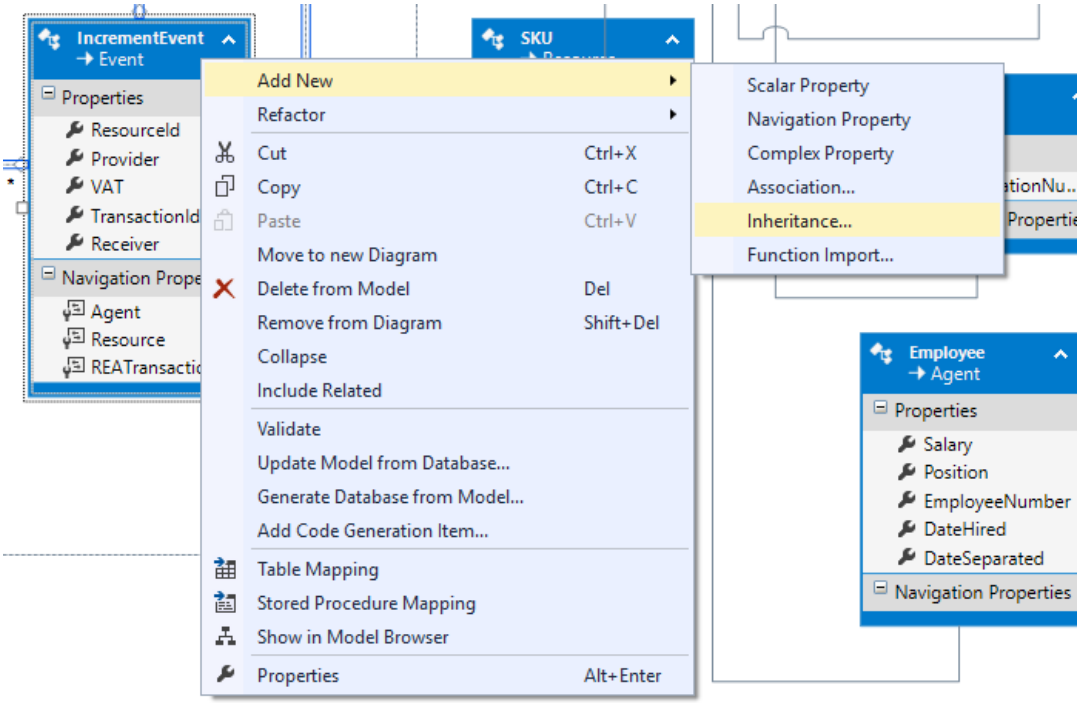


Figure 12: Map Inheritance 1a

Right-clicking on the entity that is to be mapped and selecting *Add New – Inheritance* brings up a selection window, where the user can choose the base entity and the derived entity. This process was repeated for all of the entities in the inheritance hierarchy. The base classes were marked as abstract. The columns were mapped to inherit their values from the base entities; if there are any properties that are also present in the base entity, they must be removed first.

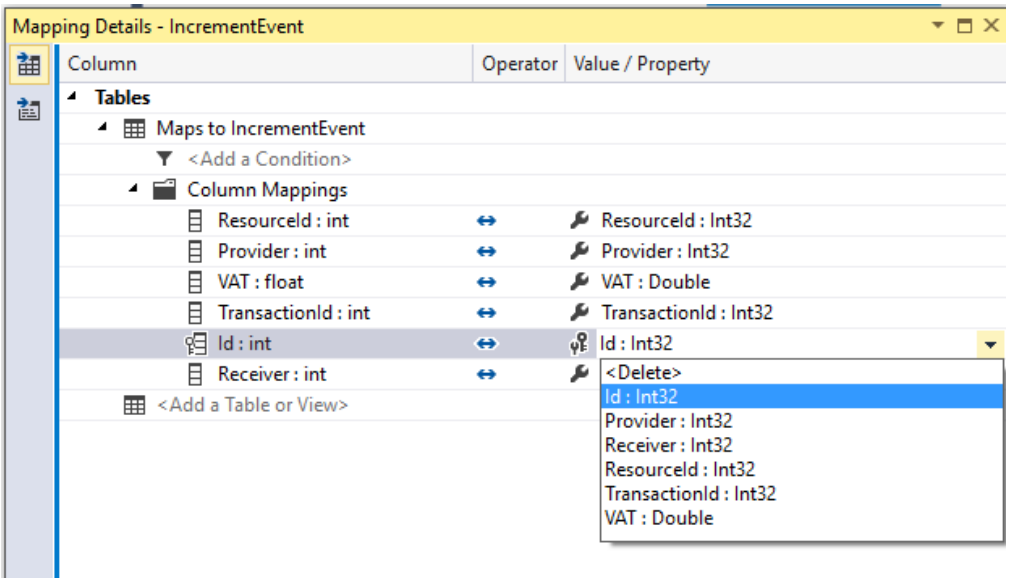


Figure 13: Map Inheritance 1b

Another advantage of TPT is polymorphic associations; in the database, a polymorphic association to a base class is represented as a foreign key referencing the table of that particular base class (e.g. IncrementEvent has a foreign key to Agent).

```
    public partial class IncrementCommitment : Commitment
    {
        public bool Fullfilled { get; set; }
        public int Provider { get; set; }
        public Nullable<double> VAT { get; set; }
        public int ResourceId { get; set; }
        public int TransactionId { get; set; }
        public Nullable<int> Receiver { get; set; }

        public virtual Agent Agent { get; set; }
        public virtual REATransaction REATransaction { get; set; }
    }
}
```

The `public virtual Agent Agent { get; set; }` represents the association. Using this it is possible to refer to instances of subclasses. For example, a query to the database:

```
ic = model.Commitments.OfType<IncrementCommitment>().SingleOrDefault(n => n.Id ==
id);
```

3.3 LINQ C#

Language-Integrated-Query (LINQ) was introduced in the.NET Framework version 3.5, and it allows developers to use query expressions to query external and in-memory data. LINQ defines a set of general purpose standard *query operators* that allow traversal, filter and projection in a direct but declarative way in any.NET based language. These standard query operators work on any **IEnumerable<T> (** an interface that tells us that we can enumerate over a sequence of `T` instances) based source. Also, it is possible to extend the LINQ standard operators with new domain-specific operators appropriate for a specific domain. For REA, I have not extended the operators as the standard set was enough at the moment, but it is a possibility in the future. LINQ has been extended to work with SQL data (LINQ to SQL), as well as Objects (LINQ to Objects) and the Entity Framework (LINQ to Entities)(Box et al., 2007).

LINQ is built on general purpose language features; lambdas, expression trees, extension methods and more. For example, a lambda expression is an anonymous function that is most often used to create delegates or expression tree types in LINQ. Expression trees are not executable code but are a tree-like data structure, where each node is an expression, e.g. a method call or binary operation like `x < y`. The code represented by expression trees can be run and compiled, which allows for example for the execution of LINQ queries in databases (Box et al., 2007). For example, this lambda expression: `Func<int, int, int> function = (a, b) => a + b;` has three parts; a declaration `Func<int, int, int> function`, an equals operator `=` , and the lambda expression `(a, b) => a + b;`. The lambda expression `(a, b) => a + b;` is actually equal to writing

```
public int function(int a, int b)
{
  return a + b;
}
```

For both cases, the method call would then be `int x = function (1, 2);`
The `Func` is a delegate type, and is declared by the **System** namespace -
`public delegate TResult Func<T1, T2, TResult>(T1 arg1, T2 arg2);`
This is a way of declaring a variable that references executable code.

The next point is to translate the code found in an expression into the expression tree. There is a fairly simple syntax to achieve this. The C# compiler generates expression trees from expression lambdas (or single-line lambdas). To do this, the developer should add a using statement to point to the Linq.Expressions namespace:
`using System.Linq.Expressions;` and then the expression can be constructed like so:
`Expression<Func<int, int, int>> expression = (a, b) => a + b;`

These expression trees are a key part of LINQ, especially in LINQ to SQL and LINQ to Entities. The reason why the conversion is necessary is that the query is not executed inside the C# program, but is translated into SQL and then executed on the database server.

For example, the following query to retrieve all decrement events from a certain time period:

```
var decrementTimeWindow = from de in d.Events.OfType<DecrementEvent>()
                                        join e in d.Events
                                        on de.Id equals e.Id
                                        where de.Date >= start && de.Date <= end
                                        select new
                                        {
                                            de
                                        };
```

Results in the following SQL query:

```
{SELECT
   [Extent1].[Id] AS [Id],
   '0X0X' AS [C1],
   [Extent1].[Date] AS [Date],
   [Extent1].[EventType] AS [EventType],
   [Extent1].[Amount] AS [Amount],
   [Extent2].[ResourceId] AS [ResourceId],
   [Extent2].[Receiver] AS [Receiver],
   [Extent2].[VAT] AS [VAT],
   [Extent2].[TransactionId] AS [TransactionId],
   [Extent2].[Provider] AS [Provider],
   CAST(NULL AS int) AS [C2],
   CAST(NULL AS int) AS [C3],
   CAST(NULL AS float) AS [C4],
   CAST(NULL AS int) AS [C5],
   CAST(NULL AS int) AS [C6]
   FROM   [dbo].[Event] AS [Extent1]
   INNER JOIN [dbo].[DecrementEvent] AS [Extent2] ON [Extent1].[Id] = [Extent2].[Id]
   INNER JOIN  (SELECT [UnionAll1].[Id] AS [Id1]
      FROM   (SELECT
         [Extent3].[Id] AS [Id]
         FROM [dbo].[DecrementEvent] AS [Extent3]
      UNION ALL
         SELECT
         [Extent4].[Id] AS [Id]
         FROM [dbo].[IncrementEvent] AS [Extent4]) AS [UnionAll1]
      INNER JOIN [dbo].[Event] AS [Extent5] ON [UnionAll1].[Id] = [Extent5].[Id] ) AS [Join2] ON
[Extent1].[Id] = [Join2].[Id1]
   WHERE ([Extent1].[Date] >= @p__linq__0) AND ([Extent1].[Date] <= @p__linq__1)}
```

3.3.1 LINQ to Entities

Using LINQ to Entities, queries are written against the Entity Framework conceptual model. The queries return objects that can be used by LINQ and EF.

I´ll briefly cover the relevant topics of eager loading, projection queries, and relationships with some example queries. Projection queries are used to select specific properties or expressions rather than the entity being queried. Properties from relevant entities can also be chosen. Eager loading is simply requesting that related data be returned with the query results, and navigating relationships means moving from an entity to its related data.

For example, the following projection query

```
var employees = from c in model.Agents
                            where c.AgentType == "Employee"
                            select new { c.Name, c.Address, c.AgentType };
```

returns :

```
[0] { Name = "Steve Jones", Address = "Mush Road 189", AgentType = "Employee" }
[1] { Name = "Missy Jones", Address = "Mush Road 189", AgentType = "Employee" }
[2] { Name = "Matty Jones", Address = "Mush Road 189", AgentType = "Employee" }
```

And, if we want to return related data (eager loading), we can use the Include method. The argument is a string that is the name of the related navigation property that we want to be returned:

```
var x = from c in model.Events.OfType<IncrementEvent>().Include("Resource")
                    where c.Amount < 100
                    select c;
```

If we only want to return a single entity, we can use the SingleOrDefault :

```
var comm = model.Events.SingleOrDefault(c => c.Id == 34);
```

One of the advantages of EDM is that the relationships are built into the model; this means that the developer does not have to construct complex joins to access related data. The queries are fairly simple when accessing a single entity, but there are also collections; for example, the decrement and increment events in the REA model. Projection can also be applied to collections, and the query

```
var ie., = from c in model.REATransactions
                    select new { c.Id,
                         Events = from e in c.IncrementEvents
                         select new {e.Agent, e.Amount, e.Date}};
```

returns an anonymous type with the Id, and a collection of events. An anonymous type lets the compiler work with types that have not been previously defined.

```
[0] { Id = 5, Events = {System.Collections.Generic.List<<>f__AnonymousTypeb<REA_Dsl.Agent,double,System.DateTime>>} }
[1] { Id = 6, Events = {System.Collections.Generic.List<<>f__AnonymousTypeb<REA_Dsl.Agent,double,System.DateTime>>} }
[2] { Id = 7, Events = {System.Collections.Generic.List<<>f__AnonymousTypeb<REA_Dsl.Agent,double,System.DateTime>>} }
[3] { Id = 8, Events = {System.Collections.Generic.List<<>f__AnonymousTypeb<REA_Dsl.Agent,double,System.DateTime>>} }
[4] { Id = 9, Events = {System.Collections.Generic.List<<>f__AnonymousTypeb<REA_Dsl.Agent,double,System.DateTime>>} }
```

3.4 Fluent Interfaces and Expression Builders

Once the domain model is done, it´s time to decide how to connect the DSL to code that implements the behavior.

An expression builder is an object, or a family of objects, that provides a fluent interface over a normal API. The intent for a fluent interface is to be readable, and an expression builder provides a fluent interface on top of a regular API. This isolates interface and fluent interface. Basically, the builder is like a translation layer that translates the fluent interface into the API (Fowler, 2010). The fluent interfaces for the REA DSL use method chaining to create a more readable syntax. The simple trick to a fluent interface is simply to make an object return itself from methods. One of the important things is to guide the user to perform certain things in a certain order.

3.5 Fluent Factory

In addition to the fluent interfaces, I wanted to make it possible for the user to create new REA entities – the model should be able to work with a number of business cases. Furthermore, the idea of having to create an interface and a factory for an unknown number of entities is not a good one. Therefore, I decided to use generics with lambda expressions, in a generic fluent factory. The factory is a creational design pattern, and does what its name implies ; it uses a specific object to create other objects. The generic factory has a method to add value to property, and to return the current instance of the entity that was set up by the factory.

Generics in C# are used for writing code for a class or a method without specifying the data type(s) the class or method works with, and the data type is then specified when for example an instance of that generic class is created.

Using the DSL and the fluent factory, the user creates a new derived class, provides a name for the class and defines the wanted properties and their types. The class is then evaluated and added to the solution. However, there is currently no functionality to map these newly created entities to the database – this can, however, be done fairly simply with the Entity Framework, or the functionality could be added in the future. The fluent factory can also be used to add new data to the database and to work with the REA entities and properties.

Creating a new class

```
var supplier = GenericFluentFactory<Agent>
.IsNew()
.WithClassName(x=>"Supplier")
.AddNewProperty(x=>"DiscountPercentage",  typeof(double))
.Create();
```

Creating a new instance of the entity

```
var c = GenericFluentFactory<Company>
.Is(new Company())
.Set(x => x.Name, "Steve´s Bike")
.Set(x => x.AgentType, "Company")
.Set(x => x.Address, "Bike Road 7")
.Set(x => x.RegistrationNumber,"B123456782")
.Create();
```

Below is an example of the supplier class generated by the DSL.

```
namespace REA_Dsl
{
    using System;

    public class Supplier : Agent
    {
        private double _DiscountPercentage;

        public double DiscountPercentage
        {
            get
            {
                return this._DiscountPercentage;
            }
            set
            {
                this._DiscountPercentage = value;
            }
        }
    }
}
```

The concrete class implementing the fluent factory also uses *Reflection* for examining and instantiating types, and to build new REA entities at runtime. Another use case for Reflection could be in mapping the generated classes to the database by using custom attributes.

3.6 Method Chaining

Method chaining is a common technique, and its most common form is with a fluent interface. The main idea is to invoke multiple method calls in a chain. For example, using a regular API, I would create my object, put it in a variable, and then use setters to manipulate the properties. If there are only a few items, it would probably just be done in a constructor, but in all likelihood, there are more than a few. With method chaining, a method that is a part of the method chain has to return an object to continue the chain, and here it returns itself:

```
public IReaFactory<T> AddNewProperty<T2>(Expression<Func<T, T2>> expression, ob-
ject value)
        {
            var propertyName = expression.Body.ToString();
            _dictionary.Add( propertyName,(Type)value);
            return this;

         }
```

However, method chaining violates many rules of common API design, such as naming conventions and command-query separation (separate methods that change state from those that don't). Also, usually multiple method calls would be written on a single line, but this doesn´t work well for method chaining, especially if there is a hierarchy (Fowler, 2010). The REA DSL uses method chaining with builders. In addition to the basic approach, the DSL uses multiple interfaces to force a fixed sequence of method chaining calls (progressive interfaces), and this is because the revenue and expense cycles in the REA model use similar language, but the behavior is different. In addition, there might a commitment specified to take place in the future, or an event might take place immediately. The DSL also lets the user build simple, native SQL statements that are executed against the database. The order of method calls is more important here.

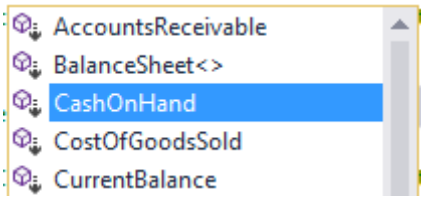
3.7 Extension Methods

An extension method lets the developer add methods to existing types without having to create a derived type or modify the original. They are a special kind of static method but are called as though they were instance methods on the extended type. For example, the LINQ standard query operators that add query functionality are actually extension methods, and they are brought to scope by using the `using System.Linq;`

The first parameter specifies which type the method operates on, and is preceded by the `this` modifier (Microsoft, n.d):

```
public static bool isNumeric(string s)
{
          float output;
          return float.TryParse(s, out output);
}
```

In Visual Studio, an extension method is displayed in the IntelliSense with downward arrow on it:



In the REA DSL, the extension methods are mostly used for reporting purposes, that is, querying the DB for financial information and building the income statement and the balance sheet. A fluent interface is used together with the extension methods to make the syntax more readable for the user, e.g.

```
Financial.Get().CashOnHand();
```

In addition to the reporting, there are some utility methods like the `public static bool isNumeric(string s).`

4 Implementation

4.1 REA Factory

The generic fluent factory contains an *orchestrator*, to avoid using syntax like new Factory (). The orchestrator is responsible for returning a ReaFactory of type T.

```
public static class GenericFluentFactory<T>
    {
        public static IReaFactory<T> Is (T entity)…

        public static IReaModel<T> IsNew()…

     }
```

The IReaFactory interface has methods to handle the creation of new entities, and a method to return the current instance of the entity created by the factory.

```
  public interface IReaFactory<T>
    {
        IReaFactory<T> Set(Expression<Func<T, object>> property, object value);
        IReaFactory<T> AddNewProperty<T2>(Expression<Func<T, T2>> expression, ob-
        ject type);
        T Create();
    }
```

The IReaModel<T> interface has two methods that are implemented by the GenericReaFactory. The methods handle the creation of a new REA entity based on user input.

```
    public interface IReaModel<T>
    {
        IReaFactory<T> WithClassName<T2>(Expression<Func<T, T2>> expression);

        IReaModel<T> CreateNewClass(IDictionary<string, Type> properties);


    }
```

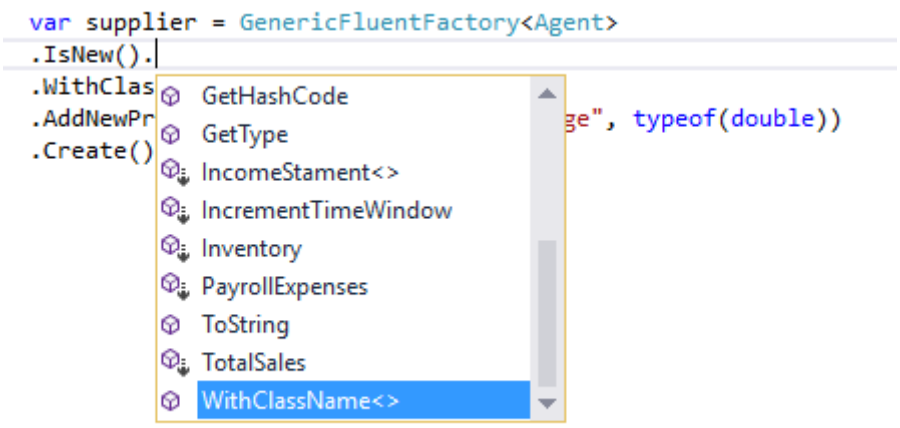
The concrete implementation class implements the methods exposed by the interfaces and includes a few other methods not visible to the DSL user.

```
public class GenericReaFactory<T> : IReaFactory<T>, IReaModel<T>
{

    T entity;
    private IDictionary<string, Type> _dictionary;
    private string className;
    private string uniqueName ="";

    public GenericReaFactory(T entity)...

    public GenericReaFactory()...

    public IReaFactory<T>Set(Expression<Func<T, object>> property, object value)...

    public IReaFactory<T> AddNewProperty<T2>(Expression<Func<T, T2>> expression, object value)...

    public IReaFactory <T> WithClassName<T2>(Expression<Func<T, T2>> expression)...

    public  T Create()...

    public IReaModel<T> CreateNewClass(IDictionary<string, Type> properties)...

    private static void AddFileToSolution(string fileName)...

    public int toInt(string s)...

    public DateTime toDate(string s)...

    public static REAEntities GetNewDataContext()...

    public void Insert()...

}
```

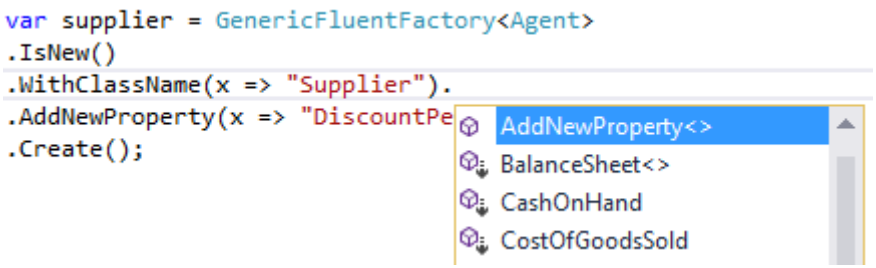
Figure 14: Concrete REA Factory

To create new REA entities, the user would first specify the type of entity he wants to create (in the example below, it is a supplier of type Agent). The IsNew() specifies that this is used to create a new concrete REA entity with properties. The IntelliSense shows that the next method in the sequence is the WithClassName<>.

```
var supplier = GenericFluentFactory<Agent>
.IsNew()
.WithClassName(x => "Supplier")
.AddNewProperty(x => "DiscountPercentage", typeof(double))
.Create();
```



Once the class name is given, the user can start to add the properties and their type. The IntelliSense again shows the available method(s). Once the properties are given, the chain finisher is the Create(); method.



The second usage for the factory is to create new instances of entities and to insert data into the database. For example, the user might want to populate the database with some data like employees. The actual transaction data will be added through transaction processing. This step would probably be only done in the beginning to add some initial data. There is currently no interface, but if one was created the user would probably add new entities through that; for example a new customer taking part in a transaction.

```
var employee = GenericFluentFactory<Employee>
              .Is(new Employee())
              .Set(x => x.Name, "Steve Jones")
              .Set(x => x.AgentType, "Employee")
              .Set(x => x.Address, "Mush Road 189")
              .Set(x => x.Salary, 30000)
              .Set(x => x.Position, "Owner")
              .Set(x => x.EmployeeNumber, "O1")
              .Set(x => x.DateHired, new DateTime(2015, 01, 01))
              .Set(x => x.DateSeparated, null)
             .Create();
```

The `var` implies that the compiler will determine the explicit type of the variable based on usage. In this case, `var` employee will be of type Employee, with a base type Agent. The process is very similar to that of creating a new class with properties. The only methods in the chain that the user uses here are Is(), Set(), and Create(); as the chain finisher.

4.2 REA Transaction

The REA Transaction is divided into different sections to handle scenarios involving commitment, expenses and revenue cycles. The interface ITransaction handles the “Give” side of a basic, revenue transaction. From the point of view of the company, in this example Steve´s Bike, the “Give” side of the transaction represents a decrement event – the company gives (sells) away a certain amount of resources, or in this case, bikes.

```
public interface ITransaction
    {
        ITransaction Give(int numberOfItems, string resource);
        ITransaction To(int agent );
        ITransaction From(int agent);
        ITransaction On(DateTime eventDate);
        ITransaction isFullfilled(bool fullfilled);
        ITransaction Record();
    }
```

For a regular transaction with no future commitment, the user would first specify the type of transaction. The other things that are required are the amount of resources, name of the resource, and the agents taking part in the event. The agents, in this case, are recognized by their id, as there might be several customers with the same name. The chain finisher is the Record(); which records and submits the changes to the database.

```
     Transaction.Type("Sale")
             .Give(1, "SuperSonic")
             .To(27)
             .From(3)
             .On(DateTime.Today)
             .Record();
```

The “Receive” side of the transaction is handled by the IDualityTransaction interface. This side of the transaction is very similar to the decrement side, except that it is possible to define a VAT percentage. The Receive side of the transaction represents an increment event from the point of view of the company, as the amount of resource under the control of the company increases; in this case, Cash.

```
  public interface IDualityTransaction
    {
        IDualityTransaction Receive(double resource, Currency currency);
        IDualityTransaction VAT(double vatPercentage);
        IDualityTransaction From(int agent);
        IDualityTransaction To(int agent);
        IDualityTransaction On(DateTime eventDate);
        IDualityTransaction isFullfilled(bool fullfilled);
        IDualityTransaction Record();
     }
```

Again, the user starts by defining a new transaction, and then records the amount that was received for the resources given in the decrement event, the agent(s) taking part in the transaction and the date when the event occurred, and again the chain finisher is Record(); which records the transaction in the database.

```
Transaction.Type("Payment")
             .Receive(2000, Currency.EUR)
             .To(3)
             .From(27)
             .On(DateTime.Today)
             .Record();
```

In addition to a transaction without a future commitment, it is possible to specify a commitment. The user can either record a new commitment or specify a transaction fulfilling a commitment recorded previously. In the ICommitmentTransaction, there is an overloaded method Commitment, one of which specifies a new commitment, and another uses an existing commitment.

```
public interface ICommitmentTransaction
    {
        ICommitmentTransaction Commitment(string commitmentType);
        ITransaction Give(int numberOfItems, string resource);
        ITransaction Reserve(int numberOfItems, string resource);
        IDualityTransaction Reserve(double amount, string resource);
        IDualityTransaction Receive(double resource, Currency currency);
        ICommitmentTransaction Due(DateTime date);
        ICommitmentTransaction Commitment(int commitmentId);
        ICommitmentTransaction isFullfilled(bool fullfilled);
    }
```

The concrete class in Figure 15implements all three interfaces, `ITransaction`, `IDualityTransaction`, and `ICommitmentTransaction`. The commitment is explained in more detail in the next section.

```
public class Transaction:ICommitmentTransaction,IDualityTransaction,ITransaction
{
    #region

    private class TableOptions...

    private List<TableOptions> tableOptionsList = new List<TableOptions>();

    private TableOptions currentTableOptions;

    private Transaction(string type)...

    private Transaction()...

    public static Transaction Type(string type)...

    public static Transaction Update()...

    public ICommitmentTransaction Commitment(string commitmentType)...

    public ICommitmentTransaction Due(DateTime due)...

    public ICommitmentTransaction Commitment(int id)...

    ICommitmentTransaction ICommitmentTransaction.isFullfilled(bool fullfilled)...

    public ITransaction Reserve(int numberOfItems, string resource)...

    public ITransaction Give(int numberOfItems, string resource)...

    ITransaction ITransaction.To(int agent)...

    ITransaction ITransaction.From(int agent)...

    ITransaction ITransaction.On(DateTime eventDate)...

    ITransaction ITransaction.isFullfilled(bool fullfilled)...

    public IDualityTransaction Reserve(double numberOfItems, string resource)...

    public IDualityTransaction Receive(double resourceValue, Currency currency)...

    IDualityTransaction IDualityTransaction.From(int agent)...

    IDualityTransaction IDualityTransaction.To(int agent)...

    IDualityTransaction IDualityTransaction.On(DateTime eventDate)...

    IDualityTransaction IDualityTransaction.isFullfilled(bool fullfilled)...

    IDualityTransaction IDualityTransaction.VAT(double vatPercentage)...

    ITransaction ITransaction.Record()...

    IDualityTransaction  IDualityTransaction.Record()...

    public static REAEntities GetNewDataContext()...
```

Figure 15: Concrete Transaction

4.2.1 REA Commitment

In addition to a transaction without a future commitment, it is possible to specify a commitment. The user can either record a new commitment or specify a transaction fulfilling a commitment recorded previously. In the ICommitmentTransaction, there is an overloaded method Commitment, one of which specifies a new commitment, and another uses an existing commitment.

```
public interface ICommitmentTransaction
    {
        ICommitmentTransaction Commitment(string commitmentType);
        ITransaction Give(int numberOfItems, string resource);
        ITransaction Reserve(int numberOfItems, string resource);
        IDualityTransaction Reserve(double amount, string resource);
        IDualityTransaction Receive(double resource, Currency currency);
        ICommitmentTransaction Due(DateTime date);
        ICommitmentTransaction Commitment(int commitmentId);
        ICommitmentTransaction isFullfilled(bool fullfilled);
    }
```

The language is very similar to that of recording a new transaction. The user specifies the transaction type, commitment type, due date and the resource and resource amount, and the agents involved, sets the isFulfilled to false and finishes the chain with the chain finisher Record(). One very important thing to note is that when updating a commitment, the user must give a commitment id of the commitment that they want to update. The commitment has not been fulfilled yet because the bike delivery has been set to take place 1.6.2016. Once the delivery takes place, the user updates the commitment and sets the isFullfilled to true, if everything went according to plan. Currently, there is no functionality for a penalty. The commitment has been marked as “Reserved: 50” in the database.

Once the bikes have been delivered, and the commitment has been fulfilled, the inventory is reduced by a number of bikes delivered. The updated commitment is then marked as updated, and because the bikes have now been delivered, a decrement event is recorded for that commitment. Additionally, the date when the update and decrement event occurred is recorded, along with the agents taking part in the transaction.

The REATransaction class keeps track of increment and decrement events, as well as increment and decrement commitments.

```
Transaction.Type("Sale")
                .Commitment("Sales Order")
                .Due(new DateTime(2016, 06, 01))
                .Reserve(50, "SuperSonic")
                .To(27)
                .From(3)
                .isFullfilled(false)
                .Record();


Transaction.Update()
                .Commitment(70)
                .Give(50, "SuperSonic")
                .On(DateTime.Today)
                .To(27)
                .From(3)
                .isFullfilled(true)
                .Record();
```

So far there are only the decrement commitments and events recorded; next, it´s time to handle the increment side. The increment commitment is from the customer's point of view; a promise to pay for the ordered items and the increment event takes place when the payment occurs. Again, the language is very similar, except that instead of recording a decrement commitment to deliver bikes, an increment commitment to receive payment is recorded. This way it is possible to track which items have not been paid for yet. The commitment will be marked with "Expected payment" until it has been marked as fulfilled and an increment event has been recorded for the commitment.

```
Transaction.Type("Payment")
                .Commitment("Cash Disbursement")
                .Due(new DateTime(2016, 06, 01))
                .Reserve(100000.0, "Cash")
                .To(3)
                .From(27)
                .isFullfilled(false)
                .Record();
```

It is also possible to record the VAT percentage. The amount of VAT is then calculated and recorded in the database.

```
Transaction.Update()
           .Commitment(57)
           .Receive(100000.0, Currency.EUR)
           .VAT(19)
           .On(DateTime.Today)
           .To(3)
           .From(27)
           .isFullfilled(true)
           .Record();
```

4.2.2 REA Expenditure

There is some functionality to handle the expense cycle; for example ordering new bikes for the inventory. The logic is similar to the commitment and transaction, and the user can also use the commitment here to order new inventory in the future, or use a "regular" transaction that takes place immediately. The ITransactionExpenditure is responsible for handling an expense transaction without a commitment, again representing a decrement event. The decrement event here would be for example a payment for a resource. The agents involved in such a transaction would be the company and a vendor.

```
public interface ITransactionExpenditure
 {
     ITransactionExpenditure Give(double amount, string resource);
     ITransactionExpenditure VAT(double vatPercentage);
     ITransactionExpenditure To(int agent);
     ITransactionExpenditure From(int agent);
     ITransactionExpenditure On(DateTime eventDate);
     ITransactionExpenditure isFullfilled(bool fullfilled);
     ITransactionExpenditure Record();
 }
```

Instead of using the `Transaction`, the user should start with the `ExpenditureTransaction.Type` to specify that this is a transaction involving the expense cycle. Apart from that, the language and flow are again very similar to the Transaction; the user specifies with "Give" the amount of resource, and the type of the resource, the VAT , the date when the event occurred, the agents involved, and whether the event was fulfilled.

```
ExpenditureTransaction.Type("Purchase")
                      .Give(50000, "Cash")
                      .VAT(19)
                      .On(new DateTime(2015, 01, 01))
                      .From(3)
                      .To(32)
                      .isFullfilled(true)
                      .Record();
```

The increment side is handled similarly by the IDualityExpenditureTransaction:

```
public interface IDualityExpenditureTransaction
  {

      IDualityExpenditureTransaction Receive(double numberOfItems,
      string resource);
      IDualityExpenditureTransaction VAT(double vatPercentage);
      IDualityExpenditureTransaction From(int agentId);
      IDualityExpenditureTransaction To(int agentId);
      IDualityExpenditureTransaction On(DateTime eventDate);
      IDualityExpenditureTransaction isFullfilled(bool fullfilled);
      IDualityExpenditureTransaction Record();
  }
}
```

To record the increment side of the transaction, the user again records the transaction type, resource and resource type, VAT, the date when the event took place, and the agents involved, and whether the event has been fulfilled. The finisher is once again Record();

```
ExpenditureTransaction.Type("Receive Inventory")
                    .Receive(25, "SuperSonic")
                    .VAT(19)
                    .On(new DateTime(2015, 01, 01))
                    .From(32)
                    .To(3)
                    .isFullfilled(true)
                    .Record();
```

In addition to regular expense transactions, it is possible to record commitments as well. However, there is no need to reserve resources here.

```
ICommitmentExpenditure Commitment(string commitmentType);
ITransactionExpenditure Give(double amount, string resource);
IDualityExpenditureTransaction Receive(double resourceValue,
string resource);
ICommitmentExpenditure Due(DateTime date);
ICommitmentExpenditure Commitment(int commitmentId);
ICommitmentExpenditure isFullfilled(bool fullfilled);
```

The expenditure transaction here is a decrement event for the company, and commitment is made to pay for the ordered resources in the future. Other than that, the language and flow follow the logic from previous examples.

```
ExpenditureTransaction.Type("Purchase")
              .Commitment("Cash Disbursement")
              .Due(new DateTime(2015, 01, 15))
              .Give(50000, "SuperSonic")
              .VAT(19)
              .From(3)
              .To(32)
              .isFullfilled(false)
              .Record();
```

And, update the commitment on the due date, and reduce the amount of resources (cash), causing a decrement event to take place in fulfilling the commitment to pay:

```
ExpenditureTransaction.Update()
              .Commitment(62)
              .Give(50000, "SuperSonic")
              .On(new DateTime(2015, 01, 15))
              .From(3)
              .To(32)
              .isFullfilled(true)
              .Record();
```

Again, it is not enough to just record the decrement side, we also must record a commitment to receive the bikes, and once the bikes have been received, the commitment will be marked as fulfilled and an increment event increasing the amount of resource will take place. The commitment could be handled in the same way as in the previous example as a commitment to receive the bikes.

```
ExpenditureTransaction.Type("Purchase")
            .Commitment("Receive Bikes")
            .Due(new DateTime(2015, 01, 15))
            .Receive(25, "SuperSonic")
            .From(32)
            .To(3)
            .isFullfilled(false)
            .Record();
```

And then when then commitment has been fulfilled, the increment event takes places and the commitment is updated:

```
    ExpenditureTransaction.Update()
                    .Commitment(68)
                    .Receive(25, "SuperSonic")
                    .VAT(19)
                    .On(new DateTime(2015, 01, 01))
                    .From(32)
                    .To(3)
                    .isFullfilled(true)
                   .Record();
```

4.3 REA Reporting – Extension methods

In addition to recording transactions, the user can generate simple financial reports; so far only two have been implemented, the income statement and the balance sheet. However, the user can retrieve other financial information, such as the value of the inventory, cost of goods sold, cash on hand, current balance, account receivable and sales. At the moment, the reporting functionality is quite limited, but the query mechanism that exists with LINQ and EF allows for more.

These have been implemented by using extension methods, and an interface for better language flow.

```
public interface IFinancial{}
```

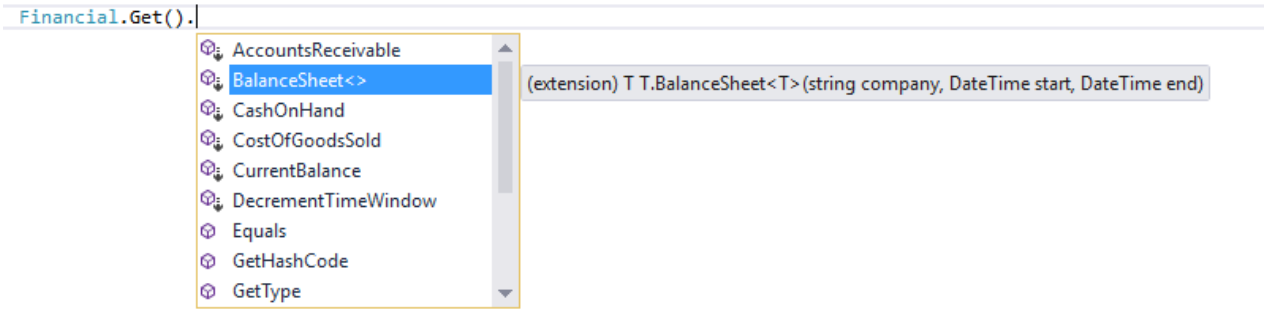
And the implementation:

```
public class Financial:IFinancial
    {
        private Financial(){}

        public static Financial Get()…
    }
```

The extension methods relevant for the reports:

```
public static class Extension
{
    public static IQueryable<T> Search<T>(this IQueryable<T> source,
        Expression<Func<T, string>> stringProperty, string searchTerm)...

    public static List<dynamic> DecrementTimeWindow(this object o, DateTime start, DateTime end)...

    public static List<dynamic> IncrementTimeWindow(this object o, DateTime start, DateTime end)...

    public static double TotalSales(this object o, DateTime start, DateTime end)...

    public static double AccountsReceivable(this object o, DateTime start, DateTime end)...

    public static double CostOfGoodsSold(this object o)...

    public static double CurrentBalance(this object o)...

    public static double CashOnHand(this object o)...

    public static double PayrollExpenses(this object o, DateTime start, DateTime end)...

    public static double Inventory(this object o)...

    public static REAEntities GetNewDataContext()...

    public static T IncomeStament<T>(this T o, string company, DateTime start, DateTime end)...

    public static T BalanceSheet<T>(this T o, string company, DateTime start, DateTime end)...

     public static string GetTableName(this DbContext ctx, Type entityType)...
```

As mentioned in 3.7 Extension Methods, the user can see the available extension methods in Visual Studio IntelliSense marked with a downward arrow:



For example, to get the cost of good sold, the user would select(or type) the method CostOfGoodsSold; `Financial.Get().CostOfGoodsSold();`
This same patter works for all of the other extension methods used for reporting.

To get the income statement and balance sheet for a specific time period, the user would speficy the company that the statement is for, and the time range :

```
DateTime start = new DateTime(2015, 01, 01);
DateTime end = new DateTime(2016, 05, 16);

Financial.Get().IncomeStatement ("Steves Bike", start, end);
Financial.Get().BalanceSheet("Steves Bike", start, end);
```

The dates could also be written straight in, like so:

```
Financial.Get().IncomeStatement ("Steves Bike", new DateTime(2015, 01, 01), new
DateTime(2016, 05, 16));
```

4.3.1 REA Query

There is also a possibility to write very simple, native SQL queries against the database. However, as the inheritance model used can result in very complicated joins; this is not as easy or efficient than using LINQ to query the database. The queries are also implemented with a builder and method chaining; progressive interfaces were used to constrict the sequence of allowed methods.

```
public interface IQueryBuilder
    {
        IQueryBuilder Select(params string[] columns);
        IQueryJoinBuilder Join(string tableName);
        IQueryBuilder As(string alias);
        IQueryBuilder Where(string filter);
        IList<T> Execute<T>();
    }
  public interface IQueryJoinBuilder
    {
        IQueryJoinBuilder As(string alias);
        IQueryBuilder On(string joinCondition);
    }
    public interface IQueryRequired
    {
        IQueryBuilder From(string tableName);

    }
```

```
public interface IQueryRequired
    {
        IQueryBuilder From(string tableName);

    }
```

And the implementation:

```
public class Query : IQueryBuilder, IQueryJoinBuilder, IQueryRequired
{

    #region
    private class TableOptions
    {
        public string[] Columns { get; set; }
        public string TableName { get; set; }
        public string Alias { get; set; }
        public string JoinCondition { get; set; }
    }

    private List<TableOptions> tableOptionsList = new List<TableOptions>();
    private TableOptions currentTableOptions;
    private string filter;

    public IQueryRequired Create()...

    public IQueryBuilder Select(params string[] columns)...

    public IQueryBuilder From(string tableName)...

    public IQueryJoinBuilder Join(string tableName)...

     IQueryJoinBuilder IQueryJoinBuilder.As(string alias)...

     IQueryBuilder IQueryBuilder.As(string alias)...

    public IQueryBuilder On(string joinCondition)...

    public IQueryBuilder Where(string filter)...

    public IList<T> Execute<T>()...
```

Figure 16: QueryBuilder

The *From* is always mandatory; the others are optional depending on the query the user wants to build. Because of the inheritance model, the user cannot instantiate a query for the core REA entities; they are abstract. For example,

```
var query = new Query().Create();
var a  = query
         .From("Agent")
         .Execute<Agent>();
```

Would fail, and return an empty list instead.

Examples:

```
var customerQ = query
.From("Agent").As("a")
.Join("Customer").As("c").On("c.Id =a.Id")
.Where("a.AgentType = 'Customer'")
.Execute<Customer>();

var ic = query
.From("Commitment").As("c")
.Join("IncrementCommitment").As("ic").On("c.Id =ic.Id")
.Where("c.Id = ic.Id")
.Execute<IncrementCommitment>();

var ie = query .From("Event").As("e")
.Join("DecrementEvent").As("de").On("e.Id =de.Id")
.Where("de.Receiver = 27")
.Execute<DecrementEvent>();

 var rt = query
 .From("REATransaction")
 Execute<REATransaction>();
```

5 Discussion and Evaluation

The goal of this thesis was to explore how the REA model could be applied in recording simple accounting transactions without using double-entry bookkeeping, or a tool such as Excel or a full blown Accounting Information System. In section 2.1 , the REA model was described in how it relates to the thesis, meaning that the model used is not a complete utilization of the possibilities REA model offers. There are many different patterns and use cases for REA, some of which are more suitable for building an AIS or an ERP system.

Another issue to consider here is a domain with some level of complexity, especially when it comes to the legalities of recording business transactions, and the trustworthiness of the information that is captured. The thesis was written without the consult of a domain expert, and so I believe there might be some issues when it comes to the accounting side of things. And so, as it stands, the REA eDSL would require more validation and constraints to ensure readiness for any production use; however, as a proof –of-concept the tool is working, as can be seen by the transactions being successfully recorded and used to generate simple financial statements and accounting information.

In addition, the domain model is not tied to any specific business case, but could be used for a number of businesses whose main goal is to sell products for profit. However, I would like to see the DSL extended to allow for more complicated business scenarios, as well as handling of contracts, claims and debt. On another note, the query builder included is very limited, but offers some use to domain experts with limited knowledge of SQL. However, it would probably be rewritten to extend LINQ instead, and possibly be made dynamic in nature.

Another point in evaluating the DSL is how useful and easy to learn it would be for the domain expert. Because the DSL is heavily based on the REA model, and takes all of its language from the model, the user is expected to understand the basics of the model, especially the domain rules. Since the concept of REA, especially in AIS development is well-established in the field of accounting, I don´t believe this to be a very large issue for the domain expert. To this end, I tried to keep the language fairly similar throughout the DSL, meaning that even though the behavior is different, the language still feels familiar and easy to learn after learning it for the basic transaction. The REA Model exposes quite a large set of terms that relate to the integrity of the model, and the domain rules, and in my opinion using all of the terms would have made the eDSL more complicated and lengthy to use. Therefore the basic nature of the REA transaction (Give –Receive) was used as a basis to differentiate between an increment and decrement, and the word Commitment was used to determine whether a commitment would take place. The resource flows were then handled based on what it was they were incrementing or decrementing, and agents were linked to each event and commitment. At the moment, the domain rules are mostly enforced by the inheritance model and the constraints set in the database, which leaves some room for the user to erroneously enter data. As mentioned earlier, more validation and error checking is needed.

Even though the goal was to record accounting transactions, it is also important that the underlying model is correct and supports the transactions and query operations needed for the financial information. The intention is to use the DSL to drive the generation of a proper structure that can be used for any business case; however as mentioned earlier this is not currently possible in the DSL, and the domain expert would have to manually map the generated entity to the existing model, and then update using the entity framework. This is not a very complicated process, but it is possible that the relationship mapping would have to be redone, and some data loss may occur. Therefore, if another entity is needed, or an existing one altered, it is suggested that this be done before populating the database with any data.

As a final note, I would not say that the REA DSL is complete, but it does achieve the set requirements of recording transactions, creating entities and extracting financial information that can be used to determine the current status and profitability of a business.

Appendix A REA Database Schema (EF)

```
- --------------------------------------------------
-- Entity Designer DDL Script for SQL Server 2005, 2008, 2012 and Azure
-- --------------------------------------------------
-- Date Created: 05/30/2016 09:59:33
-- Generated from EDMX file: C:\Users\Sa\documents\visual studio
2013\Projects\ReaDSL_\REA_Dsl\Model.edmx
-- --------------------------------------------------

SET QUOTED_IDENTIFIER OFF;
GO
USE [REA];
GO
IF SCHEMA_ID(N'dbo') IS NULL EXECUTE(N'CREATE SCHEMA [dbo]');
GO

-- --------------------------------------------------
-- Dropping existing FOREIGN KEY constraints
-- --------------------------------------------------

IF OBJECT_ID(N'[dbo].[FK_Account_inherits_Resource]', 'F') IS NOT NULL
    ALTER TABLE [dbo].[Account] DROP CONSTRAINT [FK_Account_inherits_Resource];
GO
IF OBJECT_ID(N'[dbo].[FK_Company_inherits_Agent]', 'F') IS NOT NULL
    ALTER TABLE [dbo].[Company] DROP CONSTRAINT [FK_Company_inherits_Agent];
GO
IF OBJECT_ID(N'[dbo].[FK_Customer_inherits_Agent]', 'F') IS NOT NULL
    ALTER TABLE [dbo].[Customer] DROP CONSTRAINT [FK_Customer_inherits_Agent];
GO
IF OBJECT_ID(N'[dbo].[FK_DecrementCommitment_Agent]', 'F') IS NOT NULL
    ALTER TABLE [dbo].[DecrementCommitment] DROP CONSTRAINT
[FK_DecrementCommitment_Agent];
GO
IF OBJECT_ID(N'[dbo].[FK_DecrementCommitment_inherits_Commitment]', 'F') IS NOT NULL
    ALTER TABLE [dbo].[DecrementCommitment] DROP CONSTRAINT
[FK_DecrementCommitment_inherits_Commitment];
GO
IF OBJECT_ID(N'[dbo].[FK_DecrementCommitment_REATransaction]', 'F') IS NOT NULL
    ALTER TABLE [dbo].[DecrementCommitment] DROP CONSTRAINT
[FK_DecrementCommitment_REATransaction];
GO
IF OBJECT_ID(N'[dbo].[FK_DecrementEvent_Agent]', 'F') IS NOT NULL
    ALTER TABLE [dbo].[DecrementEvent] DROP CONSTRAINT [FK_DecrementEvent_Agent];
GO
IF OBJECT_ID(N'[dbo].[FK_DecrementEvent_inherits_Event]', 'F') IS NOT NULL
    ALTER TABLE [dbo].[DecrementEvent] DROP CONSTRAINT
[FK_DecrementEvent_inherits_Event];
GO
IF OBJECT_ID(N'[dbo].[FK_Employee_inherits_Agent]', 'F') IS NOT NULL
    ALTER TABLE [dbo].[Employee] DROP CONSTRAINT [FK_Employee_inherits_Agent];
GO
IF OBJECT_ID(N'[dbo].[FK_EventSet_DecrementEvent_REATransactionSet]', 'F') IS NOT NULL
    ALTER TABLE [dbo].[DecrementEvent] DROP CONSTRAINT
[FK_EventSet_DecrementEvent_REATransactionSet];
GO
IF OBJECT_ID(N'[dbo].[FK_EventSet_DecrementEvent_ResourceSet]', 'F') IS NOT NULL
    ALTER TABLE [dbo].[DecrementEvent] DROP CONSTRAINT
[FK_EventSet_DecrementEvent_ResourceSet];
GO
IF OBJECT_ID(N'[dbo].[FK_EventSet_IncrementEvent_REATransactionSet]', 'F') IS NOT NULL
    ALTER TABLE [dbo].[IncrementEvent] DROP CONSTRAINT
[FK_EventSet_IncrementEvent_REATransactionSet];
```

```
GO
IF OBJECT_ID(N'[dbo].[FK_EventSet_InrementEvent_ResourceSet]', 'F') IS NOT NULL
    ALTER TABLE [dbo].[IncrementEvent] DROP CONSTRAINT
[FK_EventSet_InrementEvent_ResourceSet];
GO
IF OBJECT_ID(N'[dbo].[FK_IncrementCommitment_Agent]', 'F') IS NOT NULL
    ALTER TABLE [dbo].[IncrementCommitment] DROP CONSTRAINT
[FK_IncrementCommitment_Agent];
GO
IF OBJECT_ID(N'[dbo].[FK_IncrementCommitment_inherits_Commitment]', 'F') IS NOT NULL
    ALTER TABLE [dbo].[IncrementCommitment] DROP CONSTRAINT
[FK_IncrementCommitment_inherits_Commitment];
GO
IF OBJECT_ID(N'[dbo].[FK_IncrementCommitment_REATransaction]', 'F') IS NOT NULL
    ALTER TABLE [dbo].[IncrementCommitment] DROP CONSTRAINT
[FK_IncrementCommitment_REATransaction];
GO
IF OBJECT_ID(N'[dbo].[FK_IncrementEvent_Agent]', 'F') IS NOT NULL
    ALTER TABLE [dbo].[IncrementEvent] DROP CONSTRAINT [FK_IncrementEvent_Agent];
GO
IF OBJECT_ID(N'[dbo].[FK_IncrementEvent_inherits_Event]', 'F') IS NOT NULL
    ALTER TABLE [dbo].[IncrementEvent] DROP CONSTRAINT
[FK_IncrementEvent_inherits_Event];
GO
IF OBJECT_ID(N'[dbo].[FK_SKU_inherits_Resource]', 'F') IS NOT NULL
    ALTER TABLE [dbo].[SKU] DROP CONSTRAINT [FK_SKU_inherits_Resource];
GO
IF OBJECT_ID(N'[dbo].[FK_Vendor_inherits_Agent]', 'F') IS NOT NULL
    ALTER TABLE [dbo].[Vendor] DROP CONSTRAINT [FK_Vendor_inherits_Agent];
GO

-- ------------------------------------------------
-- Dropping existing tables
-- ------------------------------------------------

IF OBJECT_ID(N'[dbo].[Account]', 'U') IS NOT NULL
    DROP TABLE [dbo].[Account];
GO
IF OBJECT_ID(N'[dbo].[Agent]', 'U') IS NOT NULL
    DROP TABLE [dbo].[Agent];
GO
IF OBJECT_ID(N'[dbo].[Commitment]', 'U') IS NOT NULL
    DROP TABLE [dbo].[Commitment];
GO
IF OBJECT_ID(N'[dbo].[Company]', 'U') IS NOT NULL
    DROP TABLE [dbo].[Company];
GO
IF OBJECT_ID(N'[dbo].[Customer]', 'U') IS NOT NULL
    DROP TABLE [dbo].[Customer];
GO
IF OBJECT_ID(N'[dbo].[DecrementCommitment]', 'U') IS NOT NULL
    DROP TABLE [dbo].[DecrementCommitment];
GO
IF OBJECT_ID(N'[dbo].[DecrementEvent]', 'U') IS NOT NULL
    DROP TABLE [dbo].[DecrementEvent];
GO
IF OBJECT_ID(N'[dbo].[Employee]', 'U') IS NOT NULL
    DROP TABLE [dbo].[Employee];
GO
IF OBJECT_ID(N'[dbo].[Event]', 'U') IS NOT NULL
    DROP TABLE [dbo].[Event];
GO
IF OBJECT_ID(N'[dbo].[IncrementCommitment]', 'U') IS NOT NULL
    DROP TABLE [dbo].[IncrementCommitment];
```

```
GO
IF OBJECT_ID(N'[dbo].[IncrementEvent]', 'U') IS NOT NULL
    DROP TABLE [dbo].[IncrementEvent];
GO
IF OBJECT_ID(N'[dbo].[REATransaction]', 'U') IS NOT NULL
    DROP TABLE [dbo].[REATransaction];
GO
IF OBJECT_ID(N'[dbo].[Resource]', 'U') IS NOT NULL
    DROP TABLE [dbo].[Resource];
GO
IF OBJECT_ID(N'[dbo].[SKU]', 'U') IS NOT NULL
    DROP TABLE [dbo].[SKU];
GO
IF OBJECT_ID(N'[dbo].[Vendor]', 'U') IS NOT NULL
    DROP TABLE [dbo].[Vendor];
GO

-- ------------------------------------------------
-- Creating all tables
-- ------------------------------------------------

-- Creating table 'Resources'
CREATE TABLE [dbo].[Resources] (
    [Id] int IDENTITY(1,1) NOT NULL,
    [Name] nvarchar(max)  NOT NULL,
    [Value] float  NOT NULL,
    [ResourceType] nvarchar(max)  NOT NULL
);
GO

-- Creating table 'Resources_Account'
CREATE TABLE [dbo].[Resources_Account] (
    [AccountNumber] nvarchar(max)  NOT NULL,
    [InitialBalance] float  NOT NULL,
    [CurrentBalance] float  NOT NULL,
    [Description] nvarchar(max)  NOT NULL,
    [Id] int  NOT NULL
);
GO

-- Creating table 'Agents'
CREATE TABLE [dbo].[Agents] (
    [Id] int IDENTITY(1,1) NOT NULL,
    [Name] nvarchar(max)  NOT NULL,
    [Address] nvarchar(max)  NOT NULL,
    [AgentType] nvarchar(max)  NOT NULL
);
GO

-- Creating table 'Commitments'
CREATE TABLE [dbo].[Commitments] (
    [Id] int IDENTITY(1,1) NOT NULL,
    [Date] datetime  NOT NULL,
    [ExpectedAmount] float  NULL,
    [DueDate] datetime  NOT NULL,
    [Amount] float  NULL,
    [CommitmentType] nvarchar(max)  NOT NULL,
    [Description] varchar(max)  NULL
);
GO

-- Creating table 'Agents_Customer'
CREATE TABLE [dbo].[Agents_Customer] (
    [CustomerNumber] int IDENTITY(1,1) NOT NULL,
```

```
    [Id] int  NOT NULL
);
GO

-- Creating table 'Events'
CREATE TABLE [dbo].[Events] (
    [Id] int IDENTITY(1,1) NOT NULL,
    [Date] datetime  NOT NULL,
    [EventType] nvarchar(max)  NOT NULL,
    [Amount] float  NOT NULL
);
GO

-- Creating table 'Events_DecrementEvent'
CREATE TABLE [dbo].[Events_DecrementEvent] (
    [ResourceId] int  NOT NULL,
    [Receiver] int  NOT NULL,
    [VAT] float  NULL,
    [TransactionId] int  NOT NULL,
    [Provider] int  NULL,
    [Id] int  NOT NULL
);
GO

-- Creating table 'Events_IncrementEvent'
CREATE TABLE [dbo].[Events_IncrementEvent] (
    [ResourceId] int  NOT NULL,
    [Provider] int  NOT NULL,
    [VAT] float  NULL,
    [TransactionId] int  NOT NULL,
    [Receiver] int  NULL,
    [Id] int  NOT NULL
);
GO

-- Creating table 'Agents_Vendor'
CREATE TABLE [dbo].[Agents_Vendor] (
    [VendorNumber] int IDENTITY(1,1) NOT NULL,
    [Id] int  NOT NULL
);
GO

-- Creating table 'REATransactions'
CREATE TABLE [dbo].[REATransactions] (
    [Id] int  NOT NULL,
    [Description] nvarchar(max)  NOT NULL
);
GO

-- Creating table 'Commitments_DecrementCommitment'
CREATE TABLE [dbo].[Commitments_DecrementCommitment] (
    [Fullfilled] bit  NOT NULL,
    [Receiver] int  NOT NULL,
    [VAT] float  NULL,
    [ResourceId] int  NOT NULL,
    [TransactionId] int  NOT NULL,
    [Provider] int  NULL,
    [Id] int  NOT NULL
);
GO

-- Creating table 'Commitments_IncrementCommitment'
CREATE TABLE [dbo].[Commitments_IncrementCommitment] (
    [Fullfilled] bit  NOT NULL,
```

```
    [Provider] int  NOT NULL,
    [VAT] float  NULL,
    [ResourceId] int  NOT NULL,
    [TransactionId] int  NOT NULL,
    [Receiver] int  NULL,
    [Id] int  NOT NULL
);
GO

-- Creating table 'Agents_Company'
CREATE TABLE [dbo].[Agents_Company] (
    [RegistrationNumber] varchar(max)  NULL,
    [Id] int  NOT NULL
);
GO

-- Creating table 'Resources_SKU'
CREATE TABLE [dbo].[Resources_SKU] (
    [InitialQuantity] int  NULL,
    [QuantityOnHand] int  NULL,
    [UnitCost] float  NOT NULL,
    [Id] int  NOT NULL
);
GO

-- Creating table 'Agents_Employee'
CREATE TABLE [dbo].[Agents_Employee] (
    [Salary] float  NOT NULL,
    [Position] nvarchar(max)  NOT NULL,
    [EmployeeNumber] nvarchar(max)  NOT NULL,
    [DateHired] datetime  NOT NULL,
    [DateSeparated] datetime  NOT NULL,
    [Id] int  NOT NULL
);
GO

-- --------------------------------------------------
-- Creating all PRIMARY KEY constraints
-- --------------------------------------------------

-- Creating primary key on [Id] in table 'Resources'
ALTER TABLE [dbo].[Resources]
ADD CONSTRAINT [PK_Resources]
    PRIMARY KEY CLUSTERED ([Id] ASC);
GO

-- Creating primary key on [Id] in table 'Resources_Account'
ALTER TABLE [dbo].[Resources_Account]
ADD CONSTRAINT [PK_Resources_Account]
    PRIMARY KEY CLUSTERED ([Id] ASC);
GO

-- Creating primary key on [Id] in table 'Agents'
ALTER TABLE [dbo].[Agents]
ADD CONSTRAINT [PK_Agents]
    PRIMARY KEY CLUSTERED ([Id] ASC);
GO

-- Creating primary key on [Id] in table 'Commitments'
ALTER TABLE [dbo].[Commitments]
ADD CONSTRAINT [PK_Commitments]
    PRIMARY KEY CLUSTERED ([Id] ASC);
GO
```

```
-- Creating primary key on [Id] in table 'Agents_Customer'
ALTER TABLE [dbo].[Agents_Customer]
ADD CONSTRAINT [PK_Agents_Customer]
    PRIMARY KEY CLUSTERED ([Id] ASC);
GO

-- Creating primary key on [Id] in table 'Events'
ALTER TABLE [dbo].[Events]
ADD CONSTRAINT [PK_Events]
    PRIMARY KEY CLUSTERED ([Id] ASC);
GO

-- Creating primary key on [Id] in table 'Events_DecrementEvent'
ALTER TABLE [dbo].[Events_DecrementEvent]
ADD CONSTRAINT [PK_Events_DecrementEvent]
    PRIMARY KEY CLUSTERED ([Id] ASC);
GO

-- Creating primary key on [Id] in table 'Events_IncrementEvent'
ALTER TABLE [dbo].[Events_IncrementEvent]
ADD CONSTRAINT [PK_Events_IncrementEvent]
    PRIMARY KEY CLUSTERED ([Id] ASC);
GO

-- Creating primary key on [Id] in table 'Agents_Vendor'
ALTER TABLE [dbo].[Agents_Vendor]
ADD CONSTRAINT [PK_Agents_Vendor]
    PRIMARY KEY CLUSTERED ([Id] ASC);
GO

-- Creating primary key on [Id] in table 'REATransactions'
ALTER TABLE [dbo].[REATransactions]
ADD CONSTRAINT [PK_REATransactions]
    PRIMARY KEY CLUSTERED ([Id] ASC);
GO

-- Creating primary key on [Id] in table 'Commitments_DecrementCommitment'
ALTER TABLE [dbo].[Commitments_DecrementCommitment]
ADD CONSTRAINT [PK_Commitments_DecrementCommitment]
    PRIMARY KEY CLUSTERED ([Id] ASC);
GO

-- Creating primary key on [Id] in table 'Commitments_IncrementCommitment'
ALTER TABLE [dbo].[Commitments_IncrementCommitment]
ADD CONSTRAINT [PK_Commitments_IncrementCommitment]
    PRIMARY KEY CLUSTERED ([Id] ASC);
GO

-- Creating primary key on [Id] in table 'Agents_Company'
ALTER TABLE [dbo].[Agents_Company]
ADD CONSTRAINT [PK_Agents_Company]
    PRIMARY KEY CLUSTERED ([Id] ASC);
GO

-- Creating primary key on [Id] in table 'Resources_SKU'
ALTER TABLE [dbo].[Resources_SKU]
ADD CONSTRAINT [PK_Resources_SKU]
    PRIMARY KEY CLUSTERED ([Id] ASC);
GO

-- Creating primary key on [Id] in table 'Agents_Employee'
ALTER TABLE [dbo].[Agents_Employee]
ADD CONSTRAINT [PK_Agents_Employee]
    PRIMARY KEY CLUSTERED ([Id] ASC);
```

```
GO

-- --------------------------------------------------
-- Creating all FOREIGN KEY constraints
-- --------------------------------------------------

-- Creating foreign key on [Receiver] in table 'Events_DecrementEvent'
ALTER TABLE [dbo].[Events_DecrementEvent]
ADD CONSTRAINT [FK_DecrementEvent_Agent]
    FOREIGN KEY ([Receiver])
    REFERENCES [dbo].[Agents]
        ([Id])
    ON DELETE NO ACTION ON UPDATE NO ACTION;
GO

-- Creating non-clustered index for FOREIGN KEY 'FK_DecrementEvent_Agent'
CREATE INDEX [IX_FK_DecrementEvent_Agent]
ON [dbo].[Events_DecrementEvent]
    ([Receiver]);
GO

-- Creating foreign key on [Provider] in table 'Events_IncrementEvent'
ALTER TABLE [dbo].[Events_IncrementEvent]
ADD CONSTRAINT [FK_IncrementEvent_Agent]
    FOREIGN KEY ([Provider])
    REFERENCES [dbo].[Agents]
        ([Id])
    ON DELETE NO ACTION ON UPDATE NO ACTION;
GO

-- Creating non-clustered index for FOREIGN KEY 'FK_IncrementEvent_Agent'
CREATE INDEX [IX_FK_IncrementEvent_Agent]
ON [dbo].[Events_IncrementEvent]
    ([Provider]);
GO

-- Creating foreign key on [ResourceId] in table 'Events_DecrementEvent'
ALTER TABLE [dbo].[Events_DecrementEvent]
ADD CONSTRAINT [FK_EventSet_DecrementEvent_ResourceSet]
    FOREIGN KEY ([ResourceId])
    REFERENCES [dbo].[Resources]
        ([Id])
    ON DELETE NO ACTION ON UPDATE NO ACTION;
GO

-- Creating non-clustered index for FOREIGN KEY
'FK_EventSet_DecrementEvent_ResourceSet'
CREATE INDEX [IX_FK_EventSet_DecrementEvent_ResourceSet]
ON [dbo].[Events_DecrementEvent]
    ([ResourceId]);
GO

-- Creating foreign key on [ResourceId] in table 'Events_IncrementEvent'
ALTER TABLE [dbo].[Events_IncrementEvent]
ADD CONSTRAINT [FK_EventSet_InrementEvent_ResourceSet]
    FOREIGN KEY ([ResourceId])
    REFERENCES [dbo].[Resources]
        ([Id])
    ON DELETE NO ACTION ON UPDATE NO ACTION;
GO

-- Creating non-clustered index for FOREIGN KEY
'FK_EventSet_InrementEvent_ResourceSet'
CREATE INDEX [IX_FK_EventSet_InrementEvent_ResourceSet]
```

```
ON [dbo].[Events_IncrementEvent]
    ([ResourceId]);
GO

-- Creating foreign key on [Receiver] in table 'Commitments_DecrementCommitment'
ALTER TABLE [dbo].[Commitments_DecrementCommitment]
ADD CONSTRAINT [FK_DecrementCommitment_Agent]
    FOREIGN KEY ([Receiver])
    REFERENCES [dbo].[Agents]
        ([Id])
    ON DELETE NO ACTION ON UPDATE NO ACTION;
GO

-- Creating non-clustered index for FOREIGN KEY 'FK_DecrementCommitment_Agent'
CREATE INDEX [IX_FK_DecrementCommitment_Agent]
ON [dbo].[Commitments_DecrementCommitment]
    ([Receiver]);
GO

-- Creating foreign key on [TransactionId] in table 'Events_DecrementEvent'
ALTER TABLE [dbo].[Events_DecrementEvent]
ADD CONSTRAINT [FK_EventSet_DecrementEvent_REATransactionSet]
    FOREIGN KEY ([TransactionId])
    REFERENCES [dbo].[REATransactions]
        ([Id])
    ON DELETE NO ACTION ON UPDATE NO ACTION;
GO

-- Creating non-clustered index for FOREIGN KEY
'FK_EventSet_DecrementEvent_REATransactionSet'
CREATE INDEX [IX_FK_EventSet_DecrementEvent_REATransactionSet]
ON [dbo].[Events_DecrementEvent]
    ([TransactionId]);
GO

-- Creating foreign key on [TransactionId] in table 'Events_IncrementEvent'
ALTER TABLE [dbo].[Events_IncrementEvent]
ADD CONSTRAINT [FK_EventSet_IncrementEvent_REATransactionSet]
    FOREIGN KEY ([TransactionId])
    REFERENCES [dbo].[REATransactions]
        ([Id])
    ON DELETE NO ACTION ON UPDATE NO ACTION;
GO

-- Creating non-clustered index for FOREIGN KEY
'FK_EventSet_IncrementEvent_REATransactionSet'
CREATE INDEX [IX_FK_EventSet_IncrementEvent_REATransactionSet]
ON [dbo].[Events_IncrementEvent]
    ([TransactionId]);
GO

-- Creating foreign key on [Provider] in table 'Commitments_IncrementCommitment'
ALTER TABLE [dbo].[Commitments_IncrementCommitment]
ADD CONSTRAINT [FK_IncrementCommitment_Agent]
    FOREIGN KEY ([Provider])
    REFERENCES [dbo].[Agents]
        ([Id])
    ON DELETE NO ACTION ON UPDATE NO ACTION;
GO

-- Creating non-clustered index for FOREIGN KEY 'FK_IncrementCommitment_Agent'
CREATE INDEX [IX_FK_IncrementCommitment_Agent]
ON [dbo].[Commitments_IncrementCommitment]
    ([Provider]);
```

```
GO

-- Creating foreign key on [TransactionId] in table 'Commitments_DecrementCommitment'
ALTER TABLE [dbo].[Commitments_DecrementCommitment]
ADD CONSTRAINT [FK_DecrementCommitment_REATransaction]
    FOREIGN KEY ([TransactionId])
    REFERENCES [dbo].[REATransactions]
        ([Id])
    ON DELETE NO ACTION ON UPDATE NO ACTION;
GO

-- Creating non-clustered index for FOREIGN KEY
'FK_DecrementCommitment_REATransaction'
CREATE INDEX [IX_FK_DecrementCommitment_REATransaction]
ON [dbo].[Commitments_DecrementCommitment]
    ([TransactionId]);
GO

-- Creating foreign key on [TransactionId] in table 'Commitments_IncrementCommitment'
ALTER TABLE [dbo].[Commitments_IncrementCommitment]
ADD CONSTRAINT [FK_IncrementCommitment_REATransaction]
    FOREIGN KEY ([TransactionId])
    REFERENCES [dbo].[REATransactions]
        ([Id])
    ON DELETE NO ACTION ON UPDATE NO ACTION;
GO

-- Creating non-clustered index for FOREIGN KEY
'FK_IncrementCommitment_REATransaction'
CREATE INDEX [IX_FK_IncrementCommitment_REATransaction]
ON [dbo].[Commitments_IncrementCommitment]
    ([TransactionId]);
GO

-- Creating foreign key on [Id] in table 'Resources_Account'
ALTER TABLE [dbo].[Resources_Account]
ADD CONSTRAINT [FK_Account_inherits_Resource]
    FOREIGN KEY ([Id])
    REFERENCES [dbo].[Resources]
        ([Id])
    ON DELETE CASCADE ON UPDATE NO ACTION;
GO

-- Creating foreign key on [Id] in table 'Agents_Customer'
ALTER TABLE [dbo].[Agents_Customer]
ADD CONSTRAINT [FK_Customer_inherits_Agent]
    FOREIGN KEY ([Id])
    REFERENCES [dbo].[Agents]
        ([Id])
    ON DELETE CASCADE ON UPDATE NO ACTION;
GO

-- Creating foreign key on [Id] in table 'Events_DecrementEvent'
ALTER TABLE [dbo].[Events_DecrementEvent]
ADD CONSTRAINT [FK_DecrementEvent_inherits_Event]
    FOREIGN KEY ([Id])
    REFERENCES [dbo].[Events]
        ([Id])
    ON DELETE CASCADE ON UPDATE NO ACTION;
GO

-- Creating foreign key on [Id] in table 'Events_IncrementEvent'
ALTER TABLE [dbo].[Events_IncrementEvent]
ADD CONSTRAINT [FK_IncrementEvent_inherits_Event]
```

```
    FOREIGN KEY ([Id])
    REFERENCES [dbo].[Events]
        ([Id])
    ON DELETE CASCADE ON UPDATE NO ACTION;
GO

-- Creating foreign key on [Id] in table 'Agents_Vendor'
ALTER TABLE [dbo].[Agents_Vendor]
ADD CONSTRAINT [FK_Vendor_inherits_Agent]
    FOREIGN KEY ([Id])
    REFERENCES [dbo].[Agents]
        ([Id])
    ON DELETE CASCADE ON UPDATE NO ACTION;
GO

-- Creating foreign key on [Id] in table 'Commitments_DecrementCommitment'
ALTER TABLE [dbo].[Commitments_DecrementCommitment]
ADD CONSTRAINT [FK_DecrementCommitment_inherits_Commitment]
    FOREIGN KEY ([Id])
    REFERENCES [dbo].[Commitments]
        ([Id])
    ON DELETE CASCADE ON UPDATE NO ACTION;
GO

-- Creating foreign key on [Id] in table 'Commitments_IncrementCommitment'
ALTER TABLE [dbo].[Commitments_IncrementCommitment]
ADD CONSTRAINT [FK_IncrementCommitment_inherits_Commitment]
    FOREIGN KEY ([Id])
    REFERENCES [dbo].[Commitments]
        ([Id])
    ON DELETE CASCADE ON UPDATE NO ACTION;
GO

-- Creating foreign key on [Id] in table 'Agents_Company'
ALTER TABLE [dbo].[Agents_Company]
ADD CONSTRAINT [FK_Company_inherits_Agent]
    FOREIGN KEY ([Id])
    REFERENCES [dbo].[Agents]
        ([Id])
    ON DELETE CASCADE ON UPDATE NO ACTION;
GO

-- Creating foreign key on [Id] in table 'Resources_SKU'
ALTER TABLE [dbo].[Resources_SKU]
ADD CONSTRAINT [FK_SKU_inherits_Resource]
    FOREIGN KEY ([Id])
    REFERENCES [dbo].[Resources]
        ([Id])
    ON DELETE CASCADE ON UPDATE NO ACTION;
GO

-- Creating foreign key on [Id] in table 'Agents_Employee'
ALTER TABLE [dbo].[Agents_Employee]
ADD CONSTRAINT [FK_Employee_inherits_Agent]
    FOREIGN KEY ([Id])
    REFERENCES [dbo].[Agents]
        ([Id])
    ON DELETE CASCADE ON UPDATE NO ACTION;
GO

-- --------------------------------------------------
-- Script has ended
-- --------------------------------------------------
```

Appendix B Database Excerpts

The commitment table contains information about all of the commitments, both revenue and expense. The expense commitments are always marked with “Expense” in the commitment type; in the future this would be probably be handled by a property specifying whether a transaction is an expense transaction or a revenue transaction. In addition, we can see that the description changes to “Update” once the commitment has been fulfilled. For example, there is an open Sales Order for 50 bikes (resource):

Id	Date	ExpectedAm...	DueDate	Amount	CommitmentType	Description
76	15.01.2015	50000	15.01.2015	50000	DecrementCommitment - Expense Cash Disbursement	Updated
78	01.01.2015	25	15.01.2015	25	IncrementCommitment - Expense Receive Bikes	Updated
79	31.05.2016	50	01.06.2016	50	DecrementCommitment - Sales Order	Updated
80	31.05.2016	100000	01.06.2016	100000	IncrementCommitment - Cash Disbursement	Updated
81	01.01.0001	50	01.06.2016	*NULL*	DecrementCommitment - Sales Order	Reserved :50

Figure 17: Commitments

The DecrementCommitment table shows details about a particular commitment, the agents and resources involved, and whether VAT was taken, as well as whether or not the commitment has been fulfilled. The IncrementCommitment table shows the same information as the DecrementCommitment.

Fullfilled	Receiver	VAT	ResourceId	TransactionId	Id	Provider
True	32	*NULL*	1	35	76	3
True	27	*NULL*	7	45	79	3
False	27	*NULL*	7	50	81	3

Figure 18: Decrement Commitment

The REATransaction table shows increment and decrement events and commitments and how they are related. For example, we can see that there was simple revenue transaction with increment and decrement events, and similarly the sales order has a corresponding decrement event. If there was a commitment that had not been fulfilled, the corresponding event would not be shown either, as it would not have taken place yet.

Description	TransactionId
DecrementCommitment - Expense Cash Disbursem...	35
DecrementEvent	35
IncrementCommitment - Expense Receive Bikes	40
DecrementEvent	40
Revenue	42
Revenue	42
DecrementCommitment - Sales Order	45
DecrementEvent	45
IncrementCommitment - Cash Disbursement	48
IncrementEvent	48

Figure 19: REATransaction

The Events table holds information about the increment and decrement events. For example, for the two simple revenue transactions, we can see that 1 resource was sold for 2000; the „Fulfills" column is empty for these two transactions as there was no commitment. In addition, we can see that for example the decrement event that fulfills a commitment with Id 79, which is the Sales Order

Date	EventType	Amount	Fulfills
15.01.2015	DecrementEvent	50000	76
01.01.2015	DecrementEvent	25	78
31.05.2016	DecrementEvent - Sale	1	*NULL*
31.05.2016	IncrementEvent - Payment	2000	*NULL*
31.05.2016	DecrementEvent	50	79
31.05.2016	IncrementEvent	100000	80

Figure 20: Events

Once the bikes have been delivered, there is a payment that the company is expecting:

Id	Date	ExpectedAm...	DueDate	Amount	Commitment...	Description
76	15.01.2015	50000	15.01.2015	50000	DecrementCo...	Updated
78	01.01.2015	25	15.01.2015	25	IncrementCom...	Updated
79	31.05.2016	50	01.06.2016	50	DecrementCo...	Updated
80	31.05.2016	100000	01.06.2016	100000	IncrementCom...	Updated
81	31.05.2016	50	01.06.2016	50	DecrementCo...	Updated
82	01.01.0001	100000	01.06.2016	*NULL*	IncrementCom...	Expected Payment :100000

The payment has been completed, and the commitment updated:

Id	Date	ExpectedAm...	DueDate	Amount	CommitmentType	Description
76	15.01.2015	50000	15.01.2015	50000	DecrementCommitment - Expense Cash Disbursement	Updated
78	01.01.2015	25	15.01.2015	25	IncrementCommitment - Expense Receive Bikes	Updated
79	31.05.2016	50	01.06.2016	50	DecrementCommitment - Sales Order	Updated
80	31.05.2016	100000	01.06.2016	100000	IncrementCommitment - Cash Disbursement	Updated
81	31.05.2016	50	01.06.2016	50	DecrementCommitment - Sales Order	Updated
82	31.05.2016	100000	01.06.2016	100000	IncrementCommitment - Cash Disbursement	Updated

Similarly for the expense cycle (earlier in time):

Date	ExpectedAmo...	DueDate	Amount	CommitmentType	Description
15.01.2015	50000	15.01.2015	50000	DecrementCommitment - Expense Cash Disbursement	Updated
01.01.0001	25	15.01.2015	25	IncrementCommitment - Expense Receive Bikes	Purchase 25
NULL	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*

And we can see that the resources were affected as so:
(SKU)

AccountNum...	InitialBalance	CurrentBalance	Description	Id
DE1334554545R...	70000	20000	Operating Acco...	1

(Cash)

InitialQuantity	QuantityOnHa...	UnitCost	Id
50	75	1000	7

References

Averkamp, H. (n.d.). Accounting Topics Covered | AccountingCoach. Retrieved May 21, 2016, from http://www.accountingcoach.com/accounting-topics

Box Don, H. A. (2007). LINQ: .NET Language-Integrated Query. Abgerufen am 28. 5 2016 von Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/bb308959.aspx

Dohrn, H., Riehle, D. (2011). Design and implementation of the Sweble Wikitext parser. *Proceedings of the 7th International Symposium on Wikis and Open Collaboration - WikiSym '11*. Page 72-81.

Fowler, Martin. (2010). Domain Specific Languages (1st ed.). Addison-Wesley Professional.

Gailly, F., G. Poels.(2006a). Towards an operational REA ontology using web ontology languages.Proceedings of the 2nd International REA Technology Workshop, Santorini Island, Greece, IT University Technical Report Series, IT University of Copenhagen, Denmark.

Gailly, F., G. Poels. (2006b). Towards an OWL-formalization of the resource event agent business domain ontology. Formalization REA Business Domain Ontology. Ghent University, The Netherlands: Ghent University Department of Economics and Business Administration.

Geerts, G. L., W. E. McCarthy. (1999). An accounting object infrastructure for knowledge-based enterprise models. IEEE Intelligent Systems and Their Applications 14 (4): 89–94.

Geerts, G., Mccarthy, W. (2000a). An ontological analysis of the economic primitives of the extended-REA enterprise information architecture. *International Journal of Accounting Information Systems*, 1-16.

Geerts GL, McCarthy WE (2000b) Augmented Intensional Reasoning in Knowledge-Based Accounting Systems. Journal of Information Systems, Volume 14, No. 2, 2000, pp. 127-150.

Geerts, McCarthy (2001) REA Ontology Use in ebXML and the UN/CEFACT Modeling Methodology(N90). The First Semantic Web Working Symposium. Retrieved from https://files.ifi.uzh.ch/ddis/iswc_archive/iswc/ih/SWWS-2001/program/position/soi-mccarthy.pdf

Horngren, Ch., Harrison, W., Oliver, S.: (2012) Accounting, 9th edn. Pearson, Boston

Hruby, P. (2006).Model-Driven Design Using Business Patterns Springer, ISBN-13978-3-540-30154-7

McCarthy, WE. (1982) The REA Accounting Model: A Generalized Framework for Accounting Systems in a Shared Data Environment. The Accounting Review (July 1982) pp. 554-78

Mayrhofer, D., Huemer, C., Hofreiter, B., & Sonnenberg, C. (2011). REA-XML: An Unambiguous Language for REA Business Models. *2011 IEEE 8th International Conference on E-Business Engineering*. vol., no., pp.44,51, 19-21 Oct. 2011

Mayrhofer, D., Huemer, C. (2012). REA-DSL: Business Model Driven Data-Engineering. *2012 IEEE 14th International Conference on Commerce and Enterprise Computing*, vol., no., pp.9,16, 9-11 Sept. 2012

Meliš Z., Ševčík J., Žáček J., Huňka F. (2014).REA Key Elements Identification, Global Journal on Technology [Online].05, pp 196-201. Available from: www.awer-center.org/pitcs

Microsoft. (29. 08 2011). Getting Started (Entity Framework). Abgerufen am 27. 05 2016 von Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/bb386876%28v=vs.100%29.aspx

Microsoft (n.d). Extension Methods. Microsoft Developer Network: https://msdn.microsoft.com/en-us//library/bb383977.aspx abgerufen

Sedbrook, Tod, A. (2012) Modeling the REA Enterprise Ontology with a Domain Specific Language. Journal of Emerging Technologies in Accounting: December 2012, Vol. 9, No. 1, pp. 47-70.

Schwaiger, Walter S.A. (2015). The REA Accounting Model: Enhancing Understandability and Applicability.

Sonnenberg, C., Huemer, C., Hofreiter, B., Mayrhofer, D., Braccini, A. (2011). The REA- DSL: A Domain Specific Modeling Language for Business Models. In: Proceedings of the 23rd International Conference on Advanced Information Systems Engineering (CAiSE 2011), LNCS 6741, Springer, pp. 252-266

Bruce, K., Black, A., Homer,M., Noble, J., Ruskin, A., Yannow, R. (2013). Seeking Grace: a new object-oriented language for novices. Proceedings 44th SIGCSE Technical Symposium on Computer Science Education.

Verdaasdonk, P. 2003. An object-oriented model for ex ante accounting information. Journal of Information Systems 17 (1): 43.