Model Driven ActiveRecord with yEd

Matthias Sedlmeier, Martin Gogolla

Abstract. Since its release in 2004, Ruby on Rails has evolved into a widely used full stack model-view-controller (MVC) framework. But despite the fact, that Rails (short for Ruby on Rails) is also used for developing enterprise-scale applications like Github or scientific tools like QTREDS, there is no official support for graphical modelling. This paper introduces a proposal to fill this gap by suggesting a model driven approach using the free yEd diagram editor as well as a specifically developed transformation tool and ER dialect. The implementation is based on the Rails data abstraction layer ActiveRecord and its provided domain specific languages.

Keywords. model driven development, MDD, yEd, DSL, Ruby, Rails, ActiveRecord, graph, diagram, ER, modelling

1. Introduction

Since its release in 2004, Ruby on Rails [22] has become a widely used open source web application framework. Rails is purely implemented in Ruby, a programming language developed by Yukihiro Matsumoto in the mid-1990s [21]. Primarily extracted from a project management tool called Bootcamp, Rails evolved into a full stack model-view-controller (MVC) framework [11].

The framework essentially consists of 6 components, namely `ActionMailer`, `ActionPack`, `ActiveRecord`, `ActiveModel`, `ActiveSupport` and `Railties` as shown in Figure 1, where adjacent elements indicate an implementation or usage relation. While `ActionMailer` provides logic for email exchange, the `ActionPack` component is responsible for handling HTTP requests by providing controller code and view templates. This component roughly *implements* the view-controller part and makes usage of `ActiveSupport` functionality.

`ActiveRecord` [6] is responsible for mapping business objects to relational databases and for establishing connections between those objects carrying both persistent data and access logic. It brings its own domain specific languages for defining SQL schema migration and ORM (Object Relational Mapping) [6] model class files and hence *implements* the model part of the MVC triumvirate.

The `ActiveModel` component enables Rails to work with non-`ActiveRecord` models following the ORM principle, while `ActiveSupport` mainly delivers utility logic and extensions for the Ruby language. Finally, there is `Railties`, which glues together all components, handles the bootstrapping process and provides additional developer tools.

Rails development basically means creating text files containing Ruby code. While this is a common way to define business logic, it lacks of comfort when defining data models, because every single entity spawns 2 Ruby files. A schema containing 10 entities, for example, spreads at least over 20 separate files. The file count rises, if additional

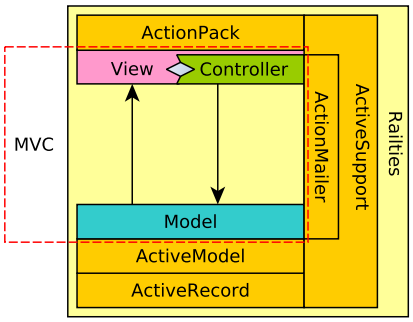

Figure 1. Ruby on Rails component structure

attributed associations are modelled. And despite the fact, that Rails is also used for developing enterprise-scale applications like Github [13] or scientific tools like QTREDS [18], there is no official support for graphical modelling although it brings advantages like clarity, comprehensibility and changeability.

This paper presents a proposal to fill this gap by introducing a model driven approach using the free yEd graph editor and a specifically developed transformation tool, which translates the yEd output into a valid textual `ActiveRecord` representation.

2. Textual Rails Modelling

To understand the advantages of the introduced approach, it is reasonable to look at the standard Rails data modelling process. The ActiveRecord DSL provides a clean abstract way to define models, their attributes and associations as well as restrictions. Each entity is represented as a Ruby model class depicting its table counterpart in a relational database. The necessary SQL schema is created by so-called migrations, which are also represented by Ruby classes.

A short minimal *Rental* example will illustrate this process in detail, while consciously avoiding any diagrammatic representations to reveal the significance of this work. An impatient reader might directly have a look at Figure 8.

We want to model the fact, that a person can rent a car and therefore extract Customer, Car and Rental as schema artefacts. Customer and Car are modelled as separate entities, while Rental is represented as a relationship between those entities. The Customer entity features a name, age, gender and customer number attribute. A Car instance has specific values for manufacturer, model, size and color. The Rental relationship keeps information about duration, free mileage and insurance coverage. A Customer may rent one or more cars at the same time, while one Car can only be attached to one renter at any given moment.

The formulated aspects are now implemented as an ActiveRecord data layer. In the first step the migration files as well as the empty model class files are created via special helper scripts, called generators (where g stands for generate), see Listing 1.

Listing 1: Command line generator calls

```
$ bin/rails g model Customer name:string age:integer
```

```
    gender:string number:integer

$ bin/rails g model Rental duration:integer
    mileage:integer insurance:string
    customer:references car:references

$ bin/rails g model Car manufacturer:string model:string
    size:integer color:string
```

2.1. Generated ActiveRecord Migration Files

The generated migration files are used to set up the SQL schema needed to save instances to a particular relational database (like SQLite [10], MySQL [23] or PostgreSQL [17]). As we modelled 2 entities and 1 attributed relationship, 3 migrations are generated.

Listing 2: Customer entity migration file

```
class CreateCustomers < ActiveRecord::Migration
  def change
    create_table :customers do |t|
      t.string  :name
      t.integer :age
      t.string  :gender
      t.integer :number

      t.timestamps
    end
  end
end
```

The Customer migration in Listing 2 defines a change on the empty SQL schema in form of a table creation. The added relation is named `customers` and maintains four attribute columns, for which the data type is explicitly indicated. All migrations use a special `timestamps` directive, which creates additional columns for creation and update time.

Listing 3: Rental relationship migration file

```
class CreateRentals < ActiveRecord::Migration
  def change
    create_table :rentals do |t|
      t.integer     :duration
      t.integer     :mileage
      t.string      :insurance
      t.references  :customer, index: true
      t.references  :car, index: true
```

```
      t.timestamps
    end
  end
end
```

Besides the description of the attribute columns, the Relation migration in Listing 3 introduces two foreign key columns. These columns will be used to join corresponding Customer and Car instances. The index option states, that the used database system defines an index on each of these fields for faster access. The last migration for Car is represented in Listing 4 and requires no additional remarks.

Listing 4: Car entity migration file

```
class CreateCars < ActiveRecord::Migration
  def change
    create_table :cars do |t|
      t.string  :manufacturer
      t.string  :model
      t.integer :size
      t.string  :color

      t.timestamps
    end
  end
end
```

The suggested solution models the Rental relationship explicitly in a separate join table. Due to the 1:n connection between Customer and Car this is not strictly necessary, because the Car table could save all the required information. There are also two different ways to place the foreign keys, assuming the Rental relationship is modelled explicitly. In the present example, Customer and Car instances are connected via two foreign keys in the Rental table. It is also possible to place a reference in the Car table pointing to the associated Rental instance, which again points to the corresponding Customer instance.

The current design decision is justified by the fact, that ActiveRecord offers additional support for 1:n connections when implemented as described.

2.2. Generated ActiveRecord model class files

Besides the migration files, also model class files are generated. These are incomplete yet and must be manually adjusted to fit the requirements. Here, we also use a ActiveRecord specific DSL to define associations and constraints. This way, we obtain 3 model class files, which are shortly described.

Listing 5: Customer entity ORM model class file

```
class Customer < ActiveRecord::Base
```

```
  has_many   :rentals
             :dependent => :destroy

  has_many   :cars,
             :through => :rentals

  validates_presence_of :name,
                        :age,
                        :gender,
                        :number

  validates_uniqueness_of :number

end
```

The first model class file represented in Listing 5 represents the Customer entity. A customer has one or more rentals, which is expressed by the `has_many` directive. ActiveRecord recognizes, that `rentals` is the lower case plural version of `Rental`, for which reason no additional information must be given. The ActiveRecord component now expects a foreign key named `customer_id` in the `Rental` table. The `dependent` option states, that associated `Rental` instances are removed, when the `Customer` is deleted. This prevents orphaned records in the rentals table. The second `has_many` directive enables direct navigation from `Customer` to `Car` instances via `Rental` records by using the `through` option. It expects a reference to `Car` in the `Rental` table, which is given by the corresponding foreign key.

Both directives implicitly establish so-called *association proxies* representing entity connections. Further, the ActiveRecord DSL allows the definition of so-called *validations*. The `Customer` model class checks, whether all attribute values are given, before an instance is saved. The customer number is additionally checked for its uniqueness.

Listing 6: Rental relationship ORM model class file

```
class Rental < ActiveRecord::Base

  belongs_to :customer
  belongs_to :car

  validates_uniqueness_of :car_id

  validates_presence_of :duration,
                        :mileage,
                        :insurance

  validates_numericality_of :duration,
                            :mileage

end
```

In the second model class file two `belongs_to` directives are used. These directives state, that each `Rental` instance is connected to one `Customer` and one `Car` instance by the corresponding foreign keys `customer_id` respectively `car_id`. To make sure each car can only be rented by one customer at the same time, the uniqueness of the `car_id` is checked. Furthermore, the `Rental` model ensures the presence of all its attribute values, of which `duration` and free `mileage` must be numeric.

Listing 7: Car entity ORM model class file

```
class Car < ActiveRecord::Base

  has_one :rental

  has_one :customer,
          :through => :rental

  validates_presence_of :manufacturer,
                        :model,
                        :size,
                        :color

  validates_numericality_of :size

end
```

The third model class file represents the `Car` entity and describes `has_one` connections to `Rental` and `Customer` via `Rental`. These directives can be seen as the counterpart of the `has_many` directives in the `Customer` model. Here, the ActiveRecord component expects a foreign key reference to `Car` and `Customer` in the `Rental` relation. As seen before, further attribute checks are declared.

It is worth to mention, that *structural constraints* or *value restrictions* are usually described and checked on application level within the Ruby code. However, it is possible to define them in the SQL schema using the described migration file mechanism as it is typically done when describing the column data types or allowing *null* values.

We have seen, that ActiveRecord is able to simplify the data modelling process by structuring the tasks necessary to implement a data layer with regard to SQL schema and ORM model class definitions. ActiveRecord also supports developers declaring associations and validations in a more abstract way by providing generators and domain specific language elements, which are basically self-explanatory using natural language. However, data modelling can be shaped more clearly and efficiently by introducing a model driven approach as we will see in the next part of this work.

In conclusion to this section, we give a short outline of the ActiveRecord migration and model class DSL fragments used in the examples above.

`create_table:`
defines a SQL table (t)

`t.string:`
defines a string column on table t

`t.integer:`
defines an integer column on table t

`t.references:`
defines a reference (id) column on table t

`t.timestamps:`
defines a create and update column on table t

`has_one:`
defines a one-to-one association to another model class;
the foreign key is expected to reside in the table of the other model class

`has_many:`
defines a one-to-many association to another model class;
the foreign key is expected to reside in the table of the other model class

`belongs_to:`
defines an association to another model class;
the foreign key resides in the current table;
usually used as counterpart to has_one and has_many directives

`validates_presence_of:`
checks the presence of an attribute or association

`validates_uniqueness_of:`
checks the uniqueness of an attribute value

`validates_numericality_of:`
checks, whether an attribute value is numeric

3. Graphical Modelling with yEd

Using yEd for the graphical representation of ActiveRecord concepts, which are automatically transformed parsing the yEd GraphML [3] output, is a novelty in the area of Rails development. yEd is free of charge, although not open source, cross-platform and uses an XML-based format called GraphML for loading and saving created diagrams, which can be easily evaluated using XPath [7] expressions.

It ships with graphical language elements for a multitude of diagram types like flowcharts, UML diagrams, BPMN and so on. For this work, the provided Entity Relationship [4] palette is used to express ActiveRecord data models and to map graphical language elements to Rails DSL fragments. Basically, it is possible to adapt the concrete

graphical yEd syntax with its nodes, edges and edge ends to fit the developer's requirements by creating custom palettes with even custom graphical language elements.

The developed transformation tool maps all graphical elements automatically to corresponding ActiveRecord migration and ORM model class files using the provided DSLs and releases the developer from manually creating any textual definitions in the best case.

Before translating the introduced textual example into a graphical representation, we give a short outline on the specifically developed ER dialect with its language elements.

Domain specific languages [5], no matter if textual or graphical, are specific, meaning, that they focus on certain main aspects of the modelled target domain. The graphical DSL introduced in this work especially aims for fast and easy description of entities, their attributes and notably their relationships also with regard to agile development processes with frequently evolving requirements. The assumption is, that restraining the expressiveness helps developers to model faster and less error-prone. Another aim is to offer a kind of standard pattern catalogue for structural composition of artefact connections, i.e. a default mapping for explicit 1:n relationships as seen in the given example.

3.1. Basic Language Description

Based on the ER notation, the proposed language supports the Entity artefact. An Entity represents an independent concept and is drawn as a rectangle labelled with its name. In contrast to popular ER dialects entities cannot be weak. In UML class diagrams they would be expressed as classes. As usual for object relational mapping approaches, an Entity is generally mapped to a single table, provided that relational database systems are used, see Figure 2.

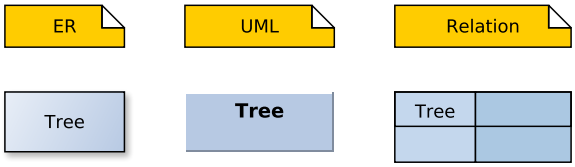

Figure 2. The entity artefact

Entities usually express their uniqueness in form of an attribute set. Attributes are drawn in circles, which are connected to entities via plain edges. Those circles are labelled with the attribute name followed by a colon and the data type. Figure 3 shows a comparative example. In some ER dialects also multi-valued attributes and derived attributes are supported. The introduced DSL does it without.

One of the main modelling tasks concerns the mapping of relationships between entities. The proposed language generally differentiates 4 basic *multiplicity types*, where type 3 represents the inverse of type 2 with swapped source and target.

1. one-to-one (1:1)
2. one-to-many (1:n)
3. many-to-one (m:1)
4. many-to-many (m:n)

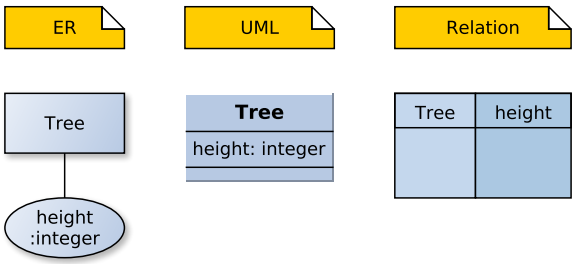

Figure 3. The entity artefact with attributes

Each basic type has multiple variations derived by its rendering, shape, presence, multiplicity and connection type. Another special case handled respects the potential self reference of entities.

The *many-to-one* multiplicity type was introduced to enhance modelling semantics in cases, where the relationship direction mismatches the intended foreign key placement. Figure 4 shows a corresponding annotated schema example. We assume, that a person has a `name` and a `gender`, while the latter one is not literally saved, but represented by an `GenderEnumeration` instance. Therefore, a foreign key column is placed in the `Person` table. Using a *one-to-many relationship* forces us to draw the edge from `GenderEnumeration` as source to `Person` as target, but that does not match our wording. We phrased, that *a person has a gender* and not, that *a gender has many persons*, which is indeed also true, but does not match the drawing direction implied by the word order, which causes confusion to no purpose.

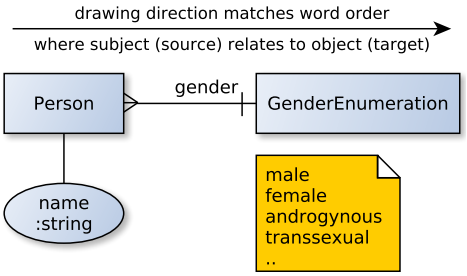

Figure 4. Exemplary usage of the many-to-one relationship

The *connection type* can be set to *association*, *aggregation* or *composition*. An association relationship does not demand any existence constraints. If A associates B and A is removed, then B remains. An aggregation relationship demands, that if A and C aggregate B and A is removed, then B remains, because C is still referencing it. If C is also removed, then B will also be destroyed. You can say, that B is *shared* between A and C. A composition relationship however demands, that if A composes B and A is removed, then B is also immediately destroyed. In this case B may not be shared between instances.

This paper will only give examples for association and composition relationships, because they can be directly mapped to ActiveRecord DSL fragments under some limitations. Supporting aggregation requires some more custom logic. It should be made clear,

that some combinations of relationship and connection types make no sense and are thus not supported. A many-to-many relationship, for example, will only have the connection type association, because it usually does not (or rather should not) model a whole-part aspect expressed as aggregation or composition.

Relationships have different *presence types* denoting, whether B instances must exist. A one-to-one relationship, for example, is this way specialized to one-to-zero-or-one (B is *optional*) or one-to-exactly-one (B is *required*).

The *rendering type* specifies, whether a relationship is modelled as a simple edge (*implicit*) or as an *explicit* relationship entity (represented by a diamond) carrying further attributes. This latter concept is known from the UML association class.

The *shape type* specifies, whether a relationship allows only one target type (*uniform*) or accepts participating entities of different types (*polymorphic*).

This paper will not discuss every reasonable combination, but will rather present selected examples to reveal the principles.

3.2. Model Examples

Figure 5 shows several 1:1 relationship variations. The upper example models the fact, that one `Citizen` maybe owns one `IdentityCard`. The relationship is expressed by an edge connecting the given entities, while the edge ends take different shapes depending on the desired configuration. In the first case the source end is clean and the target end is zero-or-one, denoted by a small circle and a single vertical line. Thus, we see an *implicit uniform optional one-to-one association relationship.*

The second example is an *implicit uniform required one-to-one association relationship.* The additional single vertical line drawn at the source end is redundant and it is the developer's choice to use it. Besides, role names are introduced at this point.

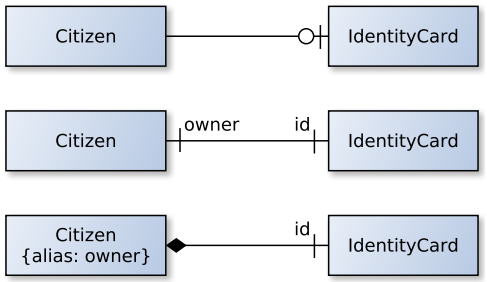

Figure 5. Implicit 1:1 relationship variants

The last case models the connection between `Citizen` and `IdentityCard` as an *implicit uniform required one-to-one composition relationship* and introduces the *alias* modifier. This modifier defines an alternative default role name, which is used instead of the real entity name. If one wants to navigate from entity A to entity B, then B is

accessible by its role name (in the context of a specific relationship) respectively alias, if given, or by its primary name.

Figure 6 shows a model, which connects planets with its (astronomic) satellites. In the first version an implicit composition relationship is modelled. The second version however uses association and is explicit, since the relationship is actually visible. The `Orbit` relationship is drawn as a diamond and introduces a `distance` attribute. In both cases target ends with circles and tripods indicate, that zero-or-more (optional) connections are allowed. The special *participation edge* is also introduced linking the diamond (`Orbit`) to its finally destiny (`Satellite`). Thus, the second case represents an *explicit uniform optional one-to-many association relationship*.

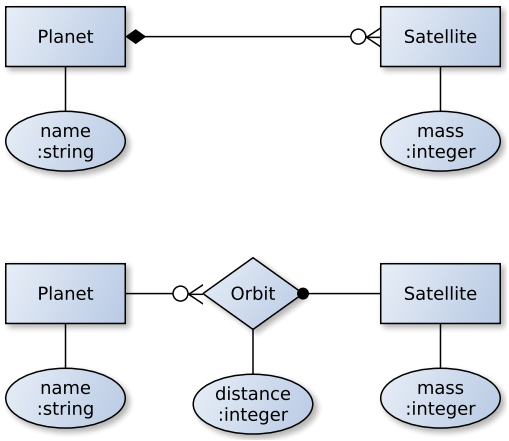

Figure 6. Implicit and explicit 1:n relationships

Figure 7 introduces implicit and explicit polymorphism. Both examples model an `Article`, `Gallery` and `Profile` entity, which are connected to the `Image` entity in different ways. The first connection (from left) denotes, that an `Article` instance is allowed to compose zero-or-more `Image` instances when a state is built. If a specific `Article` instance is deleted, then all referenced images are removed, too. The second connection depicts, that a `Gallery` associates at least one `Image` instance without any existence constraints. The last connection is a composition, which states, that a `Profile` instance composes exactly one `Image` instance, which is existence dependent from that specific profile.

Usually these connections exist without interdependencies, that means, that in a potential SQL migration, the `Image` relation would hold 2 foreign key columns for `Article` as well as `Gallery` and the `Profile` would reference `Image`. But as all connections have the same source role name `imageable`, the transformation tool derives an *implicit polymorphic relationship*. Naturally speaking, an image is able to reference an article *or* a gallery *or* a profile, which is realized by an additional type column in the `Image` SQL representation and directly supported by ActiveRecord.

This mechanism is also used in the second example, which introduces an explicit version. Here, the `Has` relationship enables the developer to attach additional information to the specific connection, a `caption` attribute in this case.

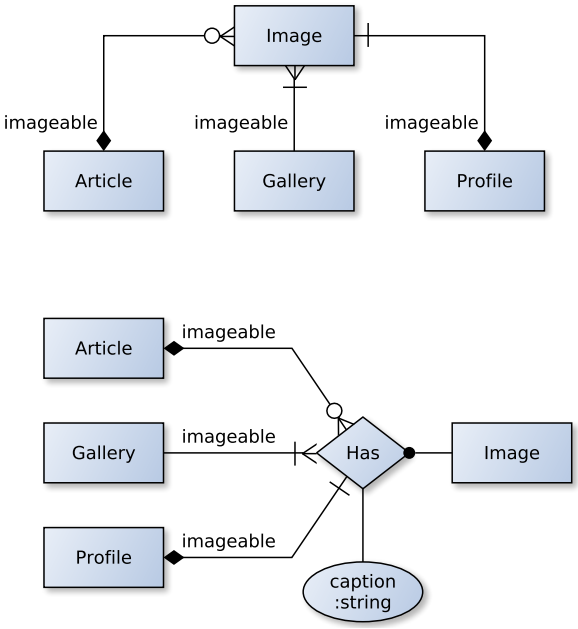

Figure 7. Implicit and explicit polymorphic relationships

3.3. Rental Example Translation

The right tools at hand, we can now translate the example introduced in the last paragraph into a graphical representation shown in Figure 8.

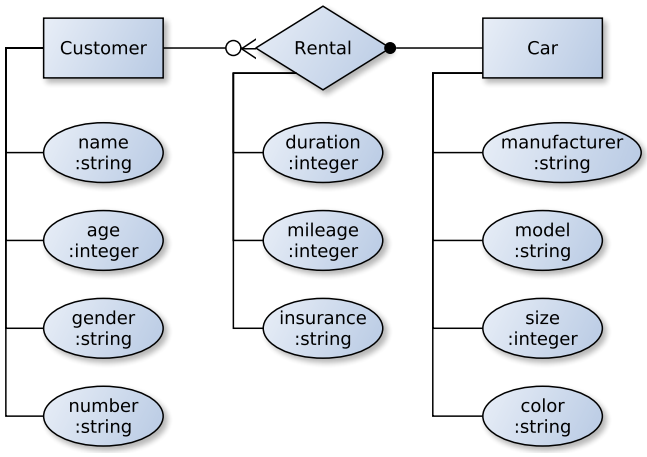

Figure 8. Rental example as diagram

This model contains 2 entities named `Customer` and `Car` as well as an *explicit uniform optional one-to-many association relationship* called `Rental`. All types shown are connected with their specific attributes by plain edges.

This model can now serve as input for the transformation tool to generate the migration and model class files considered above – thus completing the circle. By means of a real world example, the next section shows, how the generated code can be used to establish a runtime ActiveRecord data layer. It will also give some query examples.

4. A Real World Application

In this chapter we show the everyday usability of the described approach by introducing a domain model representing the political administrative structure of Germany based on the census data provided by the German Federal Statistical Office (Destatis) [2]. This data can be officially obtained on the Destatis web portal and contains one monolithic ASCII file as well as human readable record descriptions required to interpret the provided data line by line.

The different ASCII records contain information about federal states (German: Bundesland), districts (German: Bezirk), counties (Kreis), municipalities associations (Gemeindeverband) as well as municipalities (Gemeinde). Further, the hierarchical connections between those units are given as well as the places of administration. On the lowest level of the municipalities additional geographical position information was collected from public data of the Open Street Map project [1].

The domain model in Figure 9 was derived by interpreting the Destatis record descriptions and by examining examples of the instance data given. It was afterwards transformed into valid ActiveRecord migration and model class representations. The result of this translation can be found in the appendix of this work. This appendix will not be included in later versions of the paper and will be replaced by a reference to an appropriate document on the web. Furthermore, all existing data was extracted from the ASCII file, converted and used to build a corresponding state in an SQLite database.

After a short discussion on the rendered domain model and the ActiveRecord initialization process, we show some example queries formulated in ActiveRecord linguistics.

4.1. The Destatis Domain Model

The model contains elements introduced before, like entities, attributes and *implicit uniform optional* and *required one-to-many association relationships*. Furthermore, it makes use of so-called *Base Entity Types*, which are used to generalize standard entities. In the current example, `BaseUnit` as well as `UnitType` represent abstractions of concrete concepts passing common attributes. This way, entities like `FederalState` or `Municipality` inherit additional features while specializing their conceptual ancestors. Generalization is indicated by an edge between the concerned artefacts ending with a large white arrow and is *virtual*, because base types are not directly transformed into ActiveRecord representations.

The actual model also reveals different attribute variations. The `area` attribute of `Municipality` is drawn with a dashed line indicating, that this attribute is optional. Besides, the `zip_code` attribute shows a double lined border stating, that the value range must be unique. The underlined name and data type express, that this attribute must be indexed on database level, i.e. for faster access.

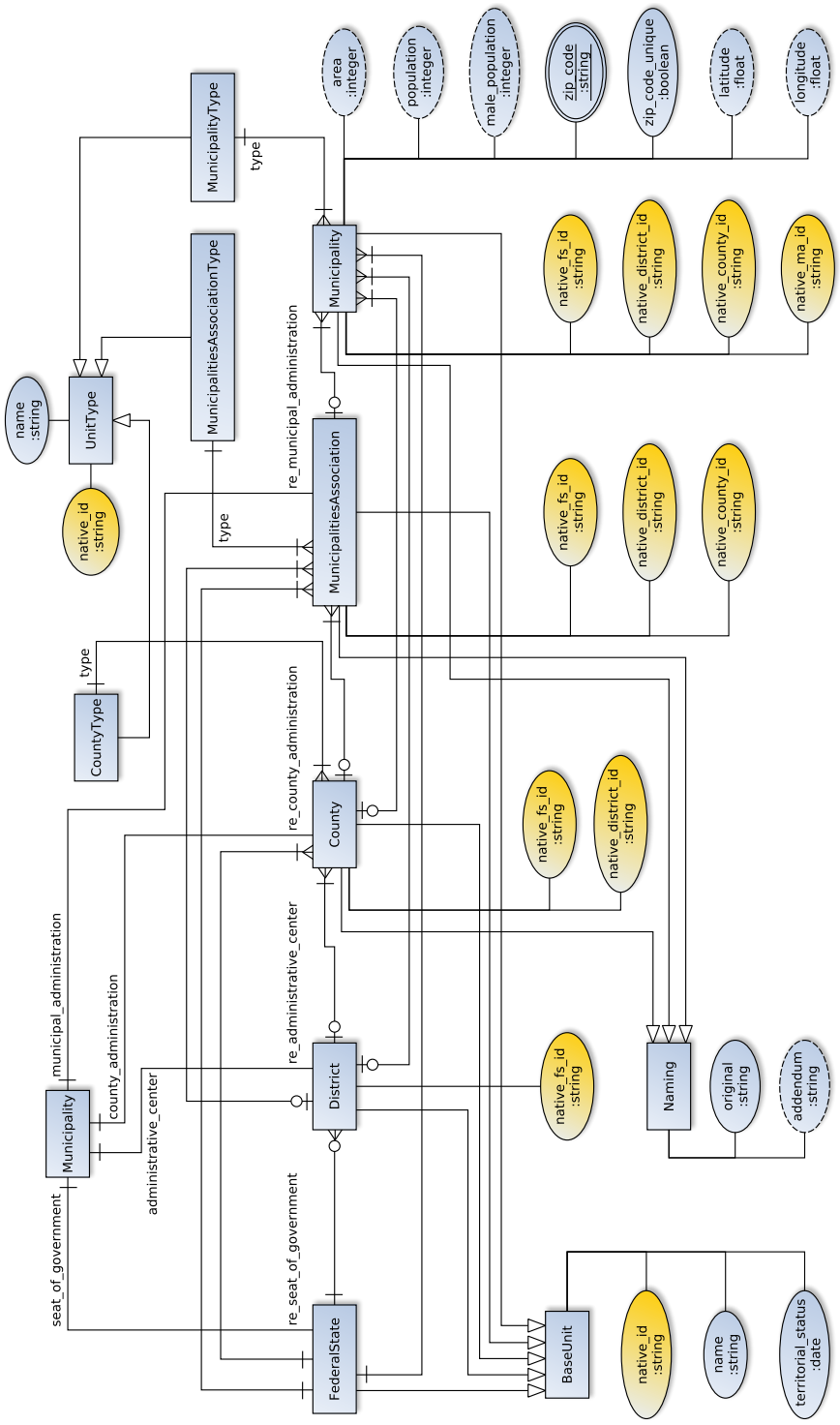

Figure 9. Model of Germany's administrative structure

The different attribute coloring has no syntactical meaning. It just highlights those attributes, which represent a reference in the native ASCII dataset and are used to load the original Destatis data.

Another noteworthy point is the double emergence of the `Municipality` entity. The transformation tool allows the multiple reference of artefacts enabling the developer to expand attribute or relationship definitions in several places in aid for clear diagram structuring even across multiple GraphML model files. Heavy connected items can be represented much easier this way as our example demonstrates.

4.2. Initializing the ActiveRecord Layer

Prior to presenting some example queries, we will shortly describe how the ActiveRecord data layer is made available at Rails runtime (see Listing 8). The transformation tool generates 16 migration and 16 model class files as described before (line 1). Afterwards, the migration files are executed to establish the SQL schema via the Ruby task management and build automation tool *rake* (line 2). Finally, another task is started, which creates the database state according to the given ASCII dataset. This process is called *seeding* and realized by separate Ruby seeding task files, which were also specifically created for this paper.

Listing 8: Rake commands

```
1 $ rake tibet:output
2 $ rake db:migrate
3 $ rake db:seed
```

When the Rails environment is booted, i.e. for a live console or in the context of a web application server instance, the model class files are loaded automatically and establish connections to the SQL database holding the according schema. The transformation tool implemented supports a special mode, which does not generate a textual representation of the model class files at all, but loads them dynamically when the Rails environment emerges. Therefore, it keeps a representation of the model in the memory, which exists during the execution time of the particular Rails process and injects the classes via metaprogramming.

The generated model class files can be customized by creating specifically named Ruby modules. If, for example, the `Municipality` model class has to be extended beyond its basic definition, the developer can create a corresponding Ruby module named `MunicipalityCustom`, which is automatically included. At this point, it is important to mention, that all additional functionality presented in this work is unobtrusive towards the Rails framework, meaning that the framework components are not changed in some way and that only the official Rails API is used. The whole approach is modelled as an additional layer upon the already existing ones.

4.3. Example Queries

As announced, we give some query examples executed in the Rails development console. We will at first specify each query in natural language and then present the ActiveRecord translation as well as the results. Internally, the so-called *ActiveRecord Query Interface*

is used rendering corresponding SQL queries.

The first query takes the first `Municipality` instance and reads the `longitude` attribute value.

Listing 9: Example query 1

```
> Municipality.find(1).longitude
= 9.43333
```

The second example returns the names of all municipalities, whose latitude value ranges from 51 to 51.5 and whose longitude value lies between 10.48 and 10.5.

Listing 10: Example query 2

```
> Municipality.where(latitude: 51..51.5)
  .where(longitude: 10.48..10.5).map(&:name)
= ["Flarchheim", "Kammerforst", "Oppershausen",
   "Vogtei"]
```

Example 3 returns the zip codes of all municipalities lying in all counties belonging to the federal state of Schleswig-Holstein.

Listing 11: Example query 3

```
> FederalState.where(name: 'Schleswig-Holstein')
  .first.counties.map(&:municipalities)
  .flatten.map(&:zip_code)
= ["25770", "21493", "23898", "25917", "25853",
   "25938", "25860", "23730", "23738", "23758", ..]
```

Query 4 sums up the population of all municipalities, which do not belong to a county. These municipalities are linked to a corresponding `MunicipalityType` instance.

Listing 12: Example query 4

```
> MunicipalityType.find(2).municipalities
  .sum(:population)
= 23561333
```

The following query example returns the count of all municipalities belonging to the district Oberbayern, which have a male population between 1000 and 10000 and whose area ranges between 1000 and 2000 units.

Listing 13: Example query 5

```
> District.where(name: 'Oberbayern')
```

```
  .first.municipalities
  .where(male_population: 1000..10000)
  .where(area: 1000..2000).count
= 6
```

And finally, the last query 6 returns the original names of all municipalities starting with the letters “Be”. The original name is the name taken from the Destatis dataset before normalization.

Listing 14: Example query 6

```
> Municipality.where('name␣LIKE␣?', 'Bas%')
  .map(&:original)
= ["Basedow", "Basthorst", "Bassum,␣Stadt",
   "Basdahl", "Bassenheim", "Basberg",
   "Bastheim", "Basedow", "Bastorf", "Basdorf"]
```

5. Related Work

There are several contributions that can be related to our present work. The importance of visual model representation is particularly emphasized in [9], where drawing sketches are used to design complex dynamic systems. Using available graph editors to generate software in modelling environments is also done in [8], where Microsoft Visio is employed. The freely obtainable yEd diagram editor introduced in this work also finds application in [20], where it helps creating schemas for a prototypical graph database. Using yEd as modelling environment is also considered in [14] discussing example-driven meta-model development. And in [12], the GraphML format underlying yEd is employed to describe UML diagrams characterizing existing PHP code.

In [16], ActiveRecord models are transformed to Alloy formal data model specifications and afterwards checked for errors using bounded verification techniques. [15] uses Ruby on Rails to show, that the developed security tool is an effective aid in the implementation of secure web application features like authentication and authorization.

Our approach is the only one that employs an everyday drawing tool for the ER-based design of an information system on the basis of Ruby on Rails and that is validated by a complex case study.

6. Conclusion and Future Work

The practical experiences made with the described approach reveals its capability. There is actually no need to define migration and model class files by hand anymore. Further, the developer does not have to cope with the question, how to represent specific artefact connections like relationships on the relational level. This makes establishing a Rails data layer significantly more comfortable and efficient. Given the graphical abstraction of the domain, the developer can immediately start working on specific application code, like controllers, views or additional model class code.

The presented approach satisfies the requirements for comprehensibility as the domain structure becomes immediately receivable. It brings clarity, because artefact definitions can be spread over multiple places, even files, and thus prevents overloaded views due to heavy connected items. Because of yEd's sophisticated drawing interface, developers can easily change the model. In combination with the entirely automated transformation process, especially agile approaches can benefit from the presented work. Consequently, developers save time and routine work.

However, there is certainly some more work to do. Currently, every model evolution causes a complete schema reset. That means, if a model *M1* is changed to model *M2*, all migration files are recreated and the database is rebuilt. Although it is possible to work with that restraint, the transformation tool should adapt to the Rails migration workflow, which allows successive schema evolutions by altering existing versions. It is possible to implement such a feature by calculating model deltas and to create migrations only for the changed parts.

Furthermore, it should be analyzed if using the ActiveRecord model inheritance mechanism brings any advantages compared to the current implementation. And besides the checking of multiplicity constraints and the uniqueness of ActiveRecord collections, one could think about realizing validations based upon OCL expressions [19].

References

[1] OpenStreetMap (OSM). http://www.openstreetmap.org/, accessed: 2015-01-18

[2] Statistisches Bundesamt (Destatis). https://www.destatis.de/DE/Startseite.html, accessed: 2015-01-18

[3] Brandes, U., Pich, C.: GraphML Transformation. In: Pach, J. (ed.) Graph Drawing, 12th International Symposium, GD 2004, New York, NY, USA, September 29 - October 2, 2004, Revised Selected Papers. Lecture Notes in Computer Science, vol. 3383, pp. 89–99. Springer (2004), http://dx.doi.org/10.1007/978-3-540-31843-9_11

[4] Chen, P.: The Entity-Relationship Model: Toward a Unified View of Data. ACM Transactions on Database Systems 1, 9–36 (1976)

[5] Consel, C.: Domain-Specific Languages: What, Why, How. Electr. Notes Theor. Comput. Sci. 65(3), 1 (2002), http://dx.doi.org/10.1016/S1571-0661(04)80422-5

[6] Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)

[7] Gottlob, G., Koch, C., Pichler, R.: XPath processing in a nutshell. SIGMOD Record 32(2), 21–27 (2003), http://doi.acm.org/10.1145/776985.776988

[8] Horkoff, J., Yu, Y., Yu, E.S.K.: OpenOME: An Open-source Goal and Agent-Oriented Model Drawing and Analysis Tool. In: de Castro, J.B., Franch, X., Mylopoulos, J., Yu, E.S.K. (eds.) Proceedings of the 5th International *i** Workshop 2011, Trento, Italy, August 28-29, 2011. CEUR Workshop Proceedings, vol. 766, pp. 154–156. CEUR-WS.org (2011), http://ceur-ws.org/Vol-766/paper27.pdf

[9] van Joolingen, W.R., Bollen, L.: Interactive drawing tools to support modeling of dynamic systems. In: Goldman, S.R., Pellegrino, J., Gomez, K., Lyons, L., Radinsky, J. (eds.) Learning in the Disciplines: Proceedings of the 9th International Conference of the Learning Sciences, ICLS '10, Chicago, IL, USA, June 29 - July 2, 2010, Volume 2. pp. 169–171. International Society of the Learning Sciences / ACM DL (2010), http://dl.acm.org/citation.cfm?id=1854589

[10] Kreibich, J.A.: Using SQLite - Small. Fast. Reliable. Choose any Three. O'Reilly (2010), http://www.oreilly.de/catalog/9780596521189/index.html

[11] Leff, A., Rayfield, J.T.: Web-Application Development Using the Model/View/Controller Design Pattern. In: 5th International Enterprise Distributed Object Computing Conference (EDOC 2001), 4-7 September 2001, Seattle, WA, USA, Proceedings. pp. 118–127. IEEE Computer Society (2001), http://computer.org/proceedings/edoc/1345/13450118abs.htm

[12] Lemos, M.: GraphML Generator: Generate UML diagrams from PHP code using GraphML. http://www.phpclasses.org/package/6025-PHP-Generate-UML-diagrams-from-PHP-code-using-GraphML.html, accessed: 2015-01-18

[13] Lima, A., Rossi, L., Musolesi, M.: Coding Together at Scale: GitHub as a Collaborative Social Network. CoRR abs/1407.2535 (2014), http://arxiv.org/abs/1407.2535

[14] Lopez-Ferndandez, J.J., Cuadrado, J.S., Guerra, E., de Lara, J.: Example-Driven Meta-Model Development. Software and System Modeling (2015)

[15] Munetoh, S., Yoshioka, N.: Model-Assisted Access Control Implementation for Code-centric Ruby-on-Rails Web Application Development. In: 2013 International Conference on Availability, Reliability and Security, ARES 2013, Regensburg, Germany, September 2-6, 2013. pp. 350–359. IEEE Computer Society (2013), http://dx.doi.org/10.1109/ARES.2013.47

[16] Nijjar, J., Bultan, T.: Bounded verification of Ruby on Rails data models. In: Dwyer, M.B., Tip, F. (eds.) Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011. pp. 67–77. ACM (2011), http://doi.acm.org/10.1145/2001420.2001429

[17] Obe, R., Hsu, L.: PostgreSQL - Up and Running: a Practical Guide to the Advanced Open Source Database. O'Reilly (2012), http://www.oreilly.de/catalog/9781449326333/index.html

[18] Palla, P., Frau, G., Vargiu, L., Rodriguez-Tomé, P.: QTREDS: a Ruby on Rails-based platform for omics laboratories. BMC Bioinformatics 15(S-1), S13 (2014), http://dx.doi.org/10.1186/1471-2105-15-S1-S13

[19] Richters, M., Gogolla, M.: On Formalizing the UML Object Constraint Language OCL. In: Ling, T.W., Ram, S., Lee, M. (eds.) Conceptual Modeling - ER '98, 17th International Conference on Conceptual Modeling, Singapore, November 16-19, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1507, pp. 449–464. Springer (1998), http://dx.doi.org/10.1007/978-3-540-49524-6_35

[20] Sedlmeier, M., Gogolla, M.: Design and Prototypical Implementation of an Integrated Graph-Based Conceptual Data Model. In: B. Thalheim, H. Jaakkola, Y.K. (ed.) Proceedings of the International Conference on Information Modelling and Knowledge Bases (EJC 2014). KCSS, vol. 2014/4, pp. 376–395. Department of Computer Science, Faculty of Engineering, Kiel University (June 2014), http://ebooks.iospress.nl/volumearticle/38513

[21] Ueno, K., Fukasawa, Y., Morihata, A., Ohori, A.: The Essence of Ruby. In: Garrigue, J. (ed.) Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings. Lecture Notes in Computer Science, vol. 8858, pp. 78–98. Springer (2014), http://dx.doi.org/10.1007/978-3-319-12736-1_5

[22] Viswanathan, V.: Rapid Web Application Development: A Ruby on Rails Tutorial. IEEE Software 25(6), 98–106 (2008), http://doi.ieeecomputersociety.org/10.1109/MS.2008.156

[23] Widenius, M., Axmark, D.: MySQL reference manual - documentation from the source. O'Reilly (2002), http://www.oreilly.de/catalog/mysqlref/index.html

A. Complete Transformation Output of Destatis Model

Please see next pages.

CreateFederalState.rb

```ruby

  class CreateFederalStates < ActiveRecord::Migration

    def change
      create_table :federal_states do |t|
        t.string :native_id, null: false
        t.string :name, null: false
        t.date :territorial_status, null: false
        t.references :seat_of_government, null: false
        t.timestamps null: false
      end
    end

  end

```

CreateDistrict.rb

```ruby

  class CreateDistricts < ActiveRecord::Migration

    def change
      create_table :districts do |t|
        t.string :native_fs_id, null: false
        t.string :native_id, null: false
        t.string :name, null: false
        t.date :territorial_status, null: false
        t.references :federal_state, null: false
        t.references :administrative_center, null: false
        t.timestamps null: false
      end
    end

  end

```

CreateCountyType.rb

```ruby

  class CreateCountyTypes < ActiveRecord::Migration

    def change
      create_table :county_types do |t|
        t.string :native_id, null: false
        t.string :name, null: false
        t.timestamps null: false
      end
    end

  end

```

CreateCounty.rb

```ruby

  class CreateCounties < ActiveRecord::Migration

    def change
      create_table :counties do |t|
        t.string :native_fs_id, null: false
        t.string :native_district_id, null: false
        t.string :native_id, null: false
        t.string :name, null: false
        t.date :territorial_status, null: false
        t.string :original, null: false
        t.string :addendum
        t.references :district
        t.references :federal_state, null: false
        t.references :type, null: false
        t.references :county_administration, null: false
        t.timestamps null: false
      end
    end

  end

```

CreateMunicipalitiesAssociationType.rb

```ruby

  class CreateMunicipalitiesAssociationTypes < ActiveRecord::Migration

    def change
      create_table :municipalities_association_types do |t|
        t.string :native_id, null: false
        t.string :name, null: false
        t.timestamps null: false
      end
    end

  end

```

CreateMunicipalitiesAssociation.rb

```ruby

  class CreateMunicipalitiesAssociations < ActiveRecord::Migration

    def change
      create_table :municipalities_associations do |t|
        t.string :native_fs_id, null: false
        t.string :native_district_id, null: false
        t.string :native_county_id, null: false
        t.string :native_id, null: false
        t.string :name, null: false
        t.date :territorial_status, null: false
        t.string :original, null: false
        t.string :addendum
        t.references :type, null: false
        t.references :county
        t.references :federal_state, null: false
        t.references :district
        t.references :municipal_administration, null: false
        t.timestamps null: false
      end
    end

  end

```

CreateMunicipalityType.rb

```ruby

  class CreateMunicipalityTypes < ActiveRecord::Migration

    def change
      create_table :municipality_types do |t|
        t.string :native_id, null: false
        t.string :name, null: false
        t.timestamps null: false
      end
    end

  end

```

CreateMunicipality.rb

```ruby

  class CreateMunicipalities < ActiveRecord::Migration

    def change
      create_table :municipalities do |t|
        t.integer :area
        t.integer :population
        t.integer :male_population
        t.string :zip_code, null: false, unique: true, index: true
        t.boolean :zip_code_unique, null: false
        t.string :native_fs_id, null: false
        t.string :native_district_id, null: false
        t.string :native_county_id, null: false
        t.string :native_ma_id, null: false
        t.float :latitude
        t.float :longitude
        t.string :native_id, null: false
        t.string :name, null: false
        t.date :territorial_status, null: false
        t.string :original, null: false
        t.string :addendum
        t.references :type, null: false
        t.references :municipalities_association
        t.references :federal_state, null: false
        t.references :district
        t.references :county
        t.timestamps null: false
      end
    end

  end

```

FederalState.rb

```ruby

  class FederalState < ActiveRecord::Base

    belongs_to :seat_of_government,
               class_name: :Municipality

    has_many :municipalities_associations,
             class_name: :MunicipalitiesAssociation,
             inverse_of: :federal_state,
             foreign_key: :federal_state_id

    has_many :counties,
             class_name: :County,
             inverse_of: :federal_state,
             foreign_key: :federal_state_id

    has_many :districts,
             class_name: :District,
             inverse_of: :federal_state,
             foreign_key: :federal_state_id

    has_many :municipalities,
             class_name: :Municipality,
             inverse_of: :federal_state,
             foreign_key: :federal_state_id

    validates_associated :municipalities

    validates_associated :municipalities_associations

    validates_associated :districts

    validates_associated :counties

    validates_presence_of :territorial_status

    validates_presence_of :name

    validates_presence_of :seat_of_government

    validates_presence_of :native_id

  end

```

District.rb

```ruby

  class District < ActiveRecord::Base

    belongs_to :federal_state,
               class_name: :FederalState

    belongs_to :administrative_center,
               class_name: :Municipality

    has_many :municipalities_associations,
             class_name: :MunicipalitiesAssociation,
             inverse_of: :district,
             foreign_key: :district_id

    has_many :counties,
             class_name: :County,
             inverse_of: :district,
             foreign_key: :district_id

    has_many :municipalities,
             class_name: :Municipality,
             inverse_of: :district,
             foreign_key: :district_id

    validates_associated :municipalities

    validates_associated :municipalities_associations

    validates_associated :counties

    validates_presence_of :federal_state

    validates_presence_of :territorial_status

    validates_presence_of :name

    validates_presence_of :native_id

    validates_presence_of :administrative_center

    validates_presence_of :native_fs_id

  end

```

CountyType.rb

```ruby

  class CountyType < ActiveRecord::Base

    has_many :counties,
             class_name: :County,
             inverse_of: :type,
             foreign_key: :type_id

    validates_associated :counties

    validates_presence_of :name

    validates_presence_of :native_id

  end

```

County.rb

```ruby

  class County < ActiveRecord::Base

    belongs_to :district,
               class_name: :District

    belongs_to :federal_state,
               class_name: :FederalState

    belongs_to :type,
               class_name: :CountyType

    belongs_to :county_administration,
               class_name: :Municipality

    has_many :municipalities_associations,
             class_name: :MunicipalitiesAssociation,
             inverse_of: :county,
             foreign_key: :county_id

    has_many :municipalities,
             class_name: :Municipality,
             inverse_of: :county,
             foreign_key: :county_id

    validates_associated :municipalities

    validates_associated :municipalities_associations

    validates_presence_of :district

    validates_presence_of :name

    validates_presence_of :territorial_status

    validates_presence_of :original

    validates_presence_of :native_id

    validates_presence_of :federal_state

    validates_presence_of :type

    validates_presence_of :native_district_id

    validates_presence_of :county_administration

    validates_presence_of :native_fs_id

  end

```

MunicipalitiesAssociationType.rb

```ruby

  class MunicipalitiesAssociationType < ActiveRecord::Base

    has_many :municipalities_associations,
             class_name: :MunicipalitiesAssociation,
             inverse_of: :type,
             foreign_key: :type_id

    validates_associated :municipalities_associations

    validates_presence_of :name

    validates_presence_of :native_id

  end

```

```ruby

  class MunicipalitiesAssociation < ActiveRecord::Base

    belongs_to :type,
               class_name: :MunicipalitiesAssociationType

    belongs_to :county,
               class_name: :County

    belongs_to :federal_state,
               class_name: :FederalState

    belongs_to :municipal_administration,
               class_name: :Municipality

    belongs_to :district,
               class_name: :District

    has_many :municipalities,
             class_name: :Municipality,
             inverse_of: :municipalities_association,
             foreign_key: :municipalities_association_id

    validates_associated :municipalities

    validates_presence_of :original

    validates_presence_of :native_county_id

    validates_presence_of :native_id

    validates_presence_of :name

    validates_presence_of :territorial_status

    validates_presence_of :native_district_id

    validates_presence_of :type

    validates_presence_of :county

    validates_presence_of :federal_state

    validates_presence_of :district

    validates_presence_of :native_fs_id

    validates_presence_of :municipal_administration

  end

```

MunicipalityType.rb

```ruby

  class MunicipalityType < ActiveRecord::Base

    has_many :municipalities,
             class_name: :Municipality,
             inverse_of: :type,
             foreign_key: :type_id

    validates_associated :municipalities

    validates_presence_of :name

    validates_presence_of :native_id

  end

```

```ruby

  class Municipality < ActiveRecord::Base

    include MunicipalityCustom

    belongs_to :type,
               class_name: :MunicipalityType

    belongs_to :municipalities_association,
               class_name: :MunicipalitiesAssociation

    belongs_to :district,
               class_name: :District

    belongs_to :federal_state,
               class_name: :FederalState

    belongs_to :county,
               class_name: :County

    has_one :re_municipal_administration,
            class_name: :MunicipalitiesAssociation,
            inverse_of: :municipal_administration,
            foreign_key: :municipal_administration_id

    has_one :re_administrative_center,
            class_name: :District,
            inverse_of: :administrative_center,
            foreign_key: :administrative_center_id

    has_one :re_seat_of_government,
            class_name: :FederalState,
            inverse_of: :seat_of_government,
            foreign_key: :seat_of_government_id

    has_one :re_county_administration,
            class_name: :County,
            inverse_of: :county_administration,
            foreign_key: :county_administration_id

    validates_associated :re_county_administration

    validates_associated :re_seat_of_government

    validates_associated :re_administrative_center

    validates_associated :re_municipal_administration

    validates_numericality_of :area,
                              :allow_nil => :true,
                              :allow_blank => :true,
                              :only_integer => :true

    validates_numericality_of :male_population,
                              :allow_nil => :true,
                              :allow_blank => :true,
                              :only_integer => :true

    validates_numericality_of :population,
                              :allow_nil => :true,
                              :allow_blank => :true,
```

```
62                              :only_integer => :true
63
64      validates_numericality_of :longitude,
65                              :allow_nil => :true,
66                              :allow_blank => :true
67
68      validates_numericality_of :latitude,
69                              :allow_nil => :true,
70                              :allow_blank => :true
71
72      validates_presence_of :native_id
73
74      validates_presence_of :native_county_id
75
76      validates_presence_of :native_district_id
77
78      validates_presence_of :native_ma_id
79
80      validates_presence_of :name
81
82      validates_presence_of :territorial_status
83
84      validates_presence_of :original
85
86      validates_presence_of :type
87
88      validates_presence_of :district
89
90      validates_presence_of :native_fs_id
91
92      validates_presence_of :zip_code_unique
93
94      validates_presence_of :county
95
96      validates_presence_of :zip_code
97
98      validates_presence_of :federal_state
99
100     validates_presence_of :municipalities_association
101
102     validates_uniqueness_of :zip_code
103
104 end
105
```