Universität Rostock Traditio et Innovatio

Professur für Geodäsie und Geoinformatik
Prof. Dr.-Ing. Ralf Bill
M.Sc. Markus Berger

Tutorial - Lecture: Modeling in UML and ER

Modeling in UML and ER

In this course unit, the main features of modeling are presented and explained using the Unified Modeling Language (UML) and the Entity Relationship Model (ER) as examples. In a separate tutorial the learned can be deepened practically.	 Bildquelle: [1]

Introduction

The problems of geoinformatics are always related to the real world. We measure really existing objects, we work on data that represent real world objects and we plan structures that should be built in the real world. This fact presents us with a problem: reality is complex. The more you work your way into a problem, the more undreamt-of connections open up. Take the example of a tree cadastre: The basic idea is to keep a directory of existing trees in an area. But what information do we want to record? The fact that it is impossible to describe every tree in its entirety is quickly apparent. The condition of every single leaf is not only uninteresting, but also invalid after the bite of a beetle.

Instead, we try to break down reality and its systems to a level that we can understand. Our brain does this naturally at every moment of our perception. If one defines a system as a totality of interconnected elements, then it is obvious that we always only work with knowledge about a subset of these elements. This subset is further limited by the fact that it must be processable by information technology for (geo)computer science.

Model

To work out exactly how this subset looks like and to understand it, we use models. A model is a simplified abstraction of a real system. When modeling, we usually pursue the following goals:

1 The model should help us to **visualize** an existing or planned system.

2. The model should **specify** the structure and behavior of a system.

3. The model shall **serve as a reference** during the implementation of a system.

4. The model should **document** our implementation decisions.

In order for a model to meet all these requirements, three characteristics are mandatory. No matter how abstract, the model must always be a **reproduction** of an existing or planned system. Depending

[1] Picture source: https://de.wikipedia.org/wiki/Unified_Modeling_Language#/media/File:UML_logo.svg

on the level of abstraction, the model is a **reduction** to relevant aspects of the original system. At the same time, the **practicability** in relation to the current task must always be kept in mind.

These characteristics should make it clear that there can be no single model for any system that exists in reality. The desired precision and context must always be taken into account. For non-trivial systems, therefore, collections of models are usually required that focus on different issues. The choice of these models can have a profound influence on the later work with a system.

Working with models can be summed up by two frequently quoted wisdoms:

"As simple as possible, as complex as necessary."

"All models are wrong, but some are useful."

In order to make models usable, they must be expressed in some way. Mostly this happens graphically, partly textually or by means of physical objects.

Some examples of models that everyone has come across before:

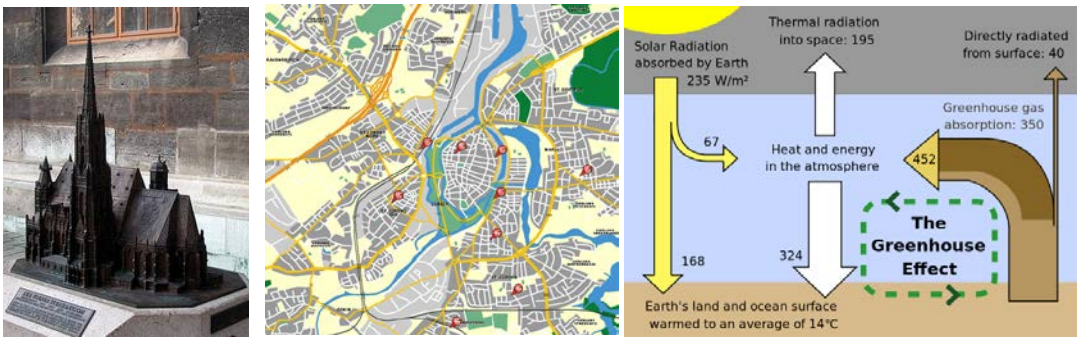

Figure 1: From left to right: A scale model of a real building, a map visualization of a settlement, a flow chart of the greenhouse effect.

They all show a simplified representation of a real structure or process. In addition, they all serve for visualization. They are intended to capture facts that are difficult to understand and make them comprehensible to the observer.

However, we want to use models for problems in geoinformatics. And as such these are information technology problems. How do we present a situation in a database? How do we classify measurement results and which values are interesting for our task? How do the steps of a planning and implementation process depend on each other?

Modeling languages

In order to deal with such questions, special modeling languages have been used both in business and science for years. These standardise the modeling process and bring many advantages. Anyone who is familiar with the language can immediately draw conclusions from the model and make changes themselves. If the model is machine-readable, the language ensures compatibility between different tools. And even if you model without tools or other people, it allows you to concentrate on solving the problem, not on redesigning a suitable form of representation.

Due to the proximity of geoinformatics to basic computer science, the two most popular modeling languages are usually used:

Unified Modeling Language (UML) **Entity-Relationship Model (ER Model)**

The ER model comes from the field of database systems and describes data structures as they are to be reflected later in the tables of a relational database. UML not only describes data structures, but also processes and system participants. The language is designed with a focus on object-oriented data storage, as it is often used in programming. Users will therefore often encounter concepts such as classes, objects and inheritance. UML is a very flexible modeling language, and offers the possibility to formulate a UML equivalent for each ER model. At the same time, it is much more complex and can make the same model more difficult for a reader to understand. Here you should always check exactly which language is best suited for the task and the target audience.

In this way we now enter all attributes and methods for modeling.

Basics of modeling languages

Although ER models can be expressed verbally, they are almost always graphically represented in practical applications. These representations are called **ER diagrams**.

For databases with high complexity it is indispensable to think about the structure before the implementation. It is very difficult to make comprehensive changes at a later stage, as there are often many dependencies and references. Using the ER model one can quickly check whether the desired abstraction makes sense in terms of information technology.

Let us first look at an example of what kind of abstraction we are talking about here. Databases are essentially collections of tables whose data can reference each other. Such tables are called relations in the world of databases. For the sake of clarity we will also use this term from here on. So let us take the following two relations as they could be in a tree cadastre of a church:

Table 1: Tree relation

tree ID	sort	height	Community ID	coordinates
001	Birke	4	1302	Punkt(20,10)
002	Fichte	6	1303	Punkt(0,100)
003	Eiche	6	1302	Punkt(20,30)

Table 2: Community relation

Community ID	community name	population
1301	Buchholz	3500
1302	Laubitz	1200
1303	Fichtenwerder	8300

What the modeling is explicitly not about are the individual datasets/lines in these relations. Individual trees (concrete objects of our reality) do not have to be modeled. Instead we model what we store about the trees and how this is stored. We already talked about detail levels, the "what" in the introduction, i.e. it is unnecessary for a tree cadastre to store the DNA sequence of the tree. For ER models, however, the "how" is even more interesting. Community names and population could be stored directly in the tree relation. However, this makes little sense from an information technology point of view, since a simple change in the number of inhabitants would mean that the number of inhabitants would have to be changed in all tree data records. Therefore, we store only one unique

reference to each municipality data set in the relation. Thus, both data sets can be edited independently from each other. The ER model should help us to plan such dependencies at an early stage and to build them up accordingly.

Entities and relations

What does such a model look like? As the name says, we start with entities. A single record (row) is an entity. A whole relation is represented by an entity type. An entity type is represented in ER diagrams by a rectangle with the name of the relation:

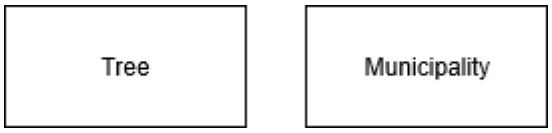

Figure 2: Entity types as rectangles

How are the dependencies displayed now? The second part of the name is responsible for this: Relationships. A relationship always exists between two entities. In our example above, each tree is in a commune. The relationship between a tree and a church could therefore be called **"is in"**. Again, we do not work at the level of relationships, but with relationship types. These summarize the relationships between the individual entities at entity type level, and use so-called cardinalities for this. These describe how many entities can be involved on each side of the relationship. Thus a tree is always located in exactly one commune, but the commune itself can contain no (0) to any number of trees (hereinafter referred to as "N"). This is represented in ER diagrams by labeled lines:

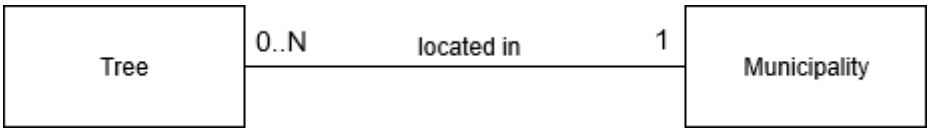

Figure 3: Relationship types as labeled lines

Primary keys

Now only one component is missing to represent several relations and their links: The columns of the *relations*. This is called *attributes*. These are not only of importance for the content, but also for the implementation of the relations. In our example table there was e.g. the *attribute "Community ID".* This has created a relation of each tree to exactly one church. An *attribute* or a collection of attributes that uniquely identify a data set in a *relation* is called a *primary key*.

In ER diagrams, attributes are represented by ellipses bound to relations, and primary keys by double ellipses.

Figure 4: Attributes and primary keys as ellipses

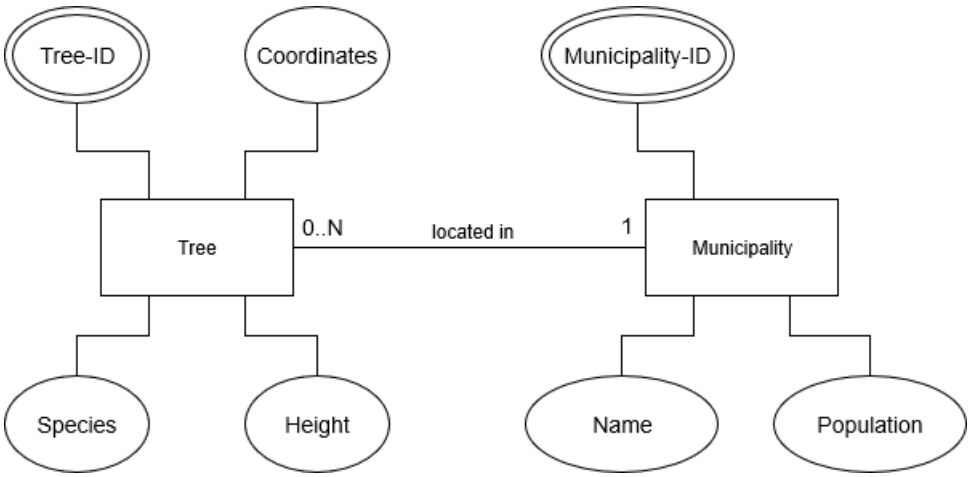

Figure 1: Attributes and Primary keys as ellipses

Attentive readers may have noticed that the community ID attribute is not associated with the tree entity type, although it occurs in its relation. This is because this attribute is only contained in the relation as a so-called *foreign key*, i.e. as a reference to the primary key of another relation. A relationship implies at least one such foreign key.

A database developer can clearly read from such an *ER diagram* which relations and keys are required. However, one problem is still hidden in our diagram: We have given the commune a "0..N" *cardinality* to the trees. This is not represented in our initial tables. To reach them, a list of tree IDs would have to be kept for each commune. However, since they can not be merged into a single field in a well-constructed database, a new relation is typically created here that contains only the two primary keys of the relations as attributes. Thus each individual relation is stored as a pair. Relationships can therefore originate not only from entity types, but also from relationship types.

For more details on using ER models with databases, see Heuer/Saake/Sattler (2007).

Unified Modeling Language (UML)

Basics

The *UML* offers far more possibilities than ER models. It contains different types of diagrams for different purposes. Originally from the field of software development, it has now spread to business, politics and research.

Just like ER models, it is usually visualized in the form of diagrams, but also offers textual representations. For advanced uses, concepts such as metamodeling and domain profiles are offered. However, these are extremely complex, and in many cases superfluous. Here we focus on a selection of the basic UML diagram types.

As already indicated, UML is based on an object-oriented way of thinking. This is not completely dissimilar to the relational thinking of the ER models, but offers some new possibilities. At the center of object-oriented thinking is - of course - the object. An object corresponds to an entity in the ER model. It contains attribute fields in which the data is stored. Which attribute fields an object has depends on its class. Classes are the equivalent of entity types. Relationships, relationship types and cardinalities exist here exactly as in ER models, and are called associations in their basic variant. However, key attributes no longer have to be specially marked. An explanation is provided in the box for those interested.

Primary and foreign keys are no longer required with the object-oriented paradigm. Instead, direct references to other objects are used. The data storage is not handled by tables, but individual objects are always addressed. If you want to change all objects of a class, you must first create a list with references. If an object is referenced, it behaves exactly like any other attribute, but its class is not "text field" or "integer", but "tree" or "list(community)".

So far, our UML diagram as a class diagram would look almost the same as an ER diagram. Tree and community are classes, individual trees and communities are objects. Relationships remain the same, only the primary key is omitted. However, attributes are noted differently. They are written with the object in the rectangle, and the data type must always be specified. This is necessary because they are less self-explanatory than in relations. In the object-oriented paradigm there is not only text, data and number fields, but also other classes can be used as attribute types.

Note: At this point not necessarily relevant, but interesting: This means that basic attributes like numbers or text fields are internally based on classes. These classes are considered to be given, and do not have to be entered in the model every time.

Our final ER diagram translated as a UML class diagram would look like this:

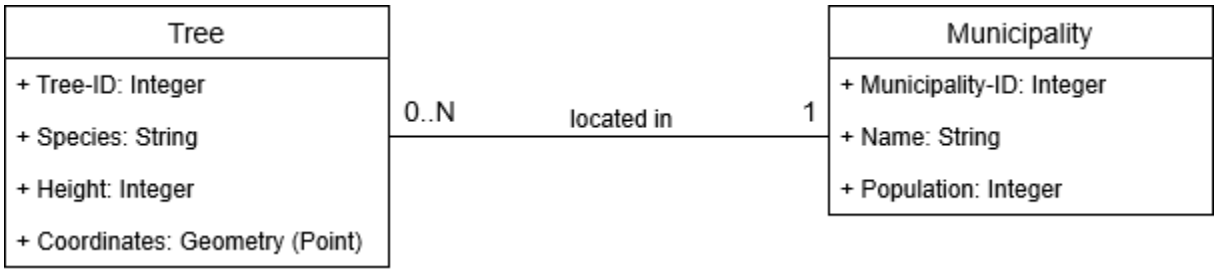

Figure 5: Example as UML class diagram

Dynamics

But we do not use UML diagrams because of the more efficient notation. There are indeed fundamental differences between ER and UML modeling. At the core of these differences lies the dynamism. *UML objects* are not only static data stores, but can also execute processes. They are called *methods*, which are bound to objects like attributes. In each of these methods sequences of steps are stored. Usually it is about changing data. A tree object could, for example, have a method "Transplant", which changes the "municipality" and the "coordinates" attribute of the tree. For this the new municipality and the new coordinates are needed. These can therefore be "passed" when calling methods, as so-called parameters. Other methods simply return information about the object to the caller without requiring parameters. So a method of a tree could be called "giveRealHeight", which adds the tree height to the height above NN and returns it as a number. Just like attributes, methods have a data type/*class* that says what data they return.

Our example model could look like this with dynamics:

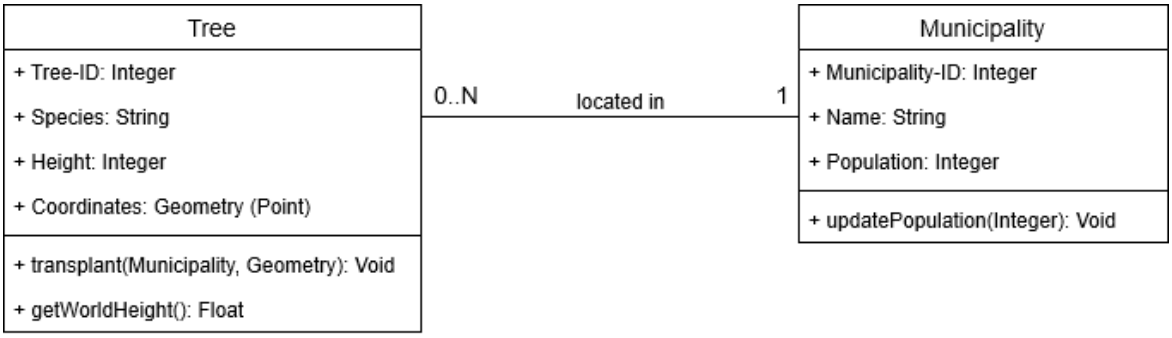

Figure 6: Example with methods (Note: The type "Void" is used if a method does not return any data to the caller)

The second major difference lies in the advanced concepts of object orientation, in particular inheritance. These are explained in more detail in the section on class diagrams.

UML diagram types

At the beginning of the chapter we mentioned that UML offers several types of diagrams. This is mainly due to the fact that dynamic processes can be modeled. It is not only possible to list which methods are available, but also how they depend on each other, at which point which object is addressed, or who calls these methods in which use case. The current version of UML (2.5.1) contains 14 diagram types. They are divided into structure and behavior diagrams, depending on whether they deal with static or dynamic processes. Behavioral diagrams also include models for user interaction with data and methods.

The diagram we created in Figure 6 is a so-called *class diagram*, the most commonly used type of diagram. It can visualize structural aspects of a system quickly and efficiently, and often serves as a basis for planning and implementation. It is a structural diagram. In the following, we will discuss the class diagram in more detail, since some important aspects have not yet been shown in our example. Then we will look at an example of a behavioral diagram.

The following table lists all currently defined types:

structure diagrams	**behaviour diagrams**
class diagram	activity diagram
component diagram	use case diagram
compositional structure diagram	interaction overview diagram
distribution diagram	communication diagram
object diagram	sequence diagram
package diagram	time course diagram
profile chart	state diagram

Class diagram

The previously described diagram is already a valid class diagram. In contrast to ER diagrams, class diagrams offer even more possibilities. The most important of these are the concepts of generalization and aggregation.

In *generalization*, we connect classes with each other, with one class being more specific and the other more general. In other words, one class abstracts the other. So we could have a class "plant" which generalizes the class "tree". A tree is a special kind of plant. Such relationships are very common in object-oriented modeling, and are indicated by a white arrow. The effect of such constructs is

that the more specific class "*inherits*" all attributes and methods of the more general class, but additionally adds its own characteristics. Thus, a user knows that he can "replant" all plants, no matter what kind of plant he has in front of him. Another effect is that you save a lot of work in the "community" class. You do not have to create a new attribute for each new type of plant, instead you can add more types later without changing anything in the "Community" class.

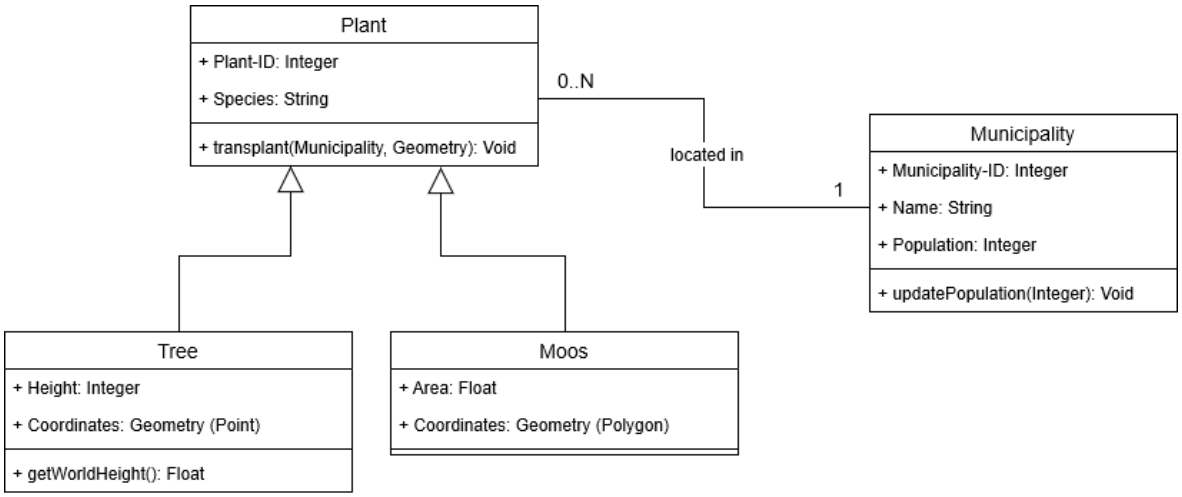

Figure 7: Class diagram with generalization (white arrows are to be read in arrow direction as "specialized")

UML relationships

When modeling, you should always look for possibilities for generalization, as these are one of the cornerstones of modeling: *Abstraction*. Finding these relationships is one of the main goals when creating class diagrams.

Another concept is *aggregation*. This is a special form of association, the relationship between two objects. While an association is basically only meant for two objects to communicate with each other and to reference each other, an aggregation states that one class is a "container" for the other class. The latter is therefore part of a whole. It can even be part of several containers, i.e. in an aggregation the part is not dependent on its whole.

A *composition* is again a stricter aggregation in which the part is dependent on its whole and can only be part of a maximum of one whole. In an aggregation the parts can be separated from the whole, in a composition not. This definition is not quite intuitive, and even in well cured sources it is often wrongly explained and applied. We therefore try to illustrate the idea with our example: Let's assume that the administrative unit under the municipality is called "district". In addition, there are usually several settlements (villages, cities, etc.) in a municipality, which can possibly be located across a border in two municipalities. The district is a bureaucratic classification, the settlement a real existing construct. One of these relationships is an aggregation, the other a composition. Why?

First of all, both are aggregations, since they are content of the community. The community not only has a referential relationship with them, but part of its purpose is to gather districts and settlements among itself. In order to find out which relationship is a composition, we need to ask ourselves the following question: What happens if the commune disappears all at once - e.g. as part of a territorial reform. The settlements would, of course, continue to exist, since hopefully they will not be demol-

ished in the course of the reform. This is therefore an aggregation. However, the districts in the municipality would disappear, as they only make sense in connection with this. The congregation is therefore a composition of districts.

Note: An error is often made here. Just because the object components are lost in a composition when their container is destroyed does not necessarily mean that they need a container. Compositions are possible with cardinalities of 1 and 0..1. In our example, this could mean, for example, that there could be districts that are independent of municipalities, as we know it from cities that are independent of municipalities.

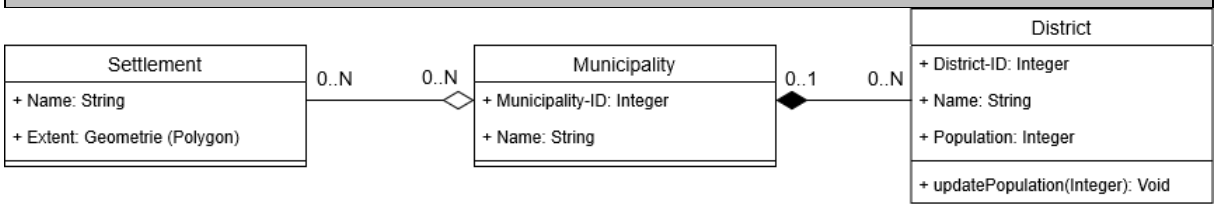

Figure 8: Example of aggregation and composition

This covers all important aspects of the class diagram. These aspects also form the basis of many other diagrams in UML, since here the object-oriented paradigm is modeled in its basic form.

Use case diagram

At the end of this introduction, a *behavior diagram* will also be explained, namely the *use case diagram*. In behavior diagrams, both the dynamic behavior of a system and the interactions of users with it are modelled. A use case diagram focuses on these interactions. Specifically, it focuses on which users exist and for what they can use the system.

The users are referred to here as *actors*. In the diagram they are marked as stick figures. Actors are connected via associations (similar to class diagrams) to *use cases*, which are represented as ellipses. Actors can be connected to each other via generalizations, just like use cases. The concept is the same as for class diagrams. Actors associated with the general use case have the same association with the specialized use cases, and specialized actors adopt the use cases with which their generalized actor is associated.

There are also two new relationships: The "*include*" and "*exclude*" relationship. They exist between use cases and describe how they depend on each other. If the "include" relationship is used, then the included use case (to which the arrow points) is a necessary condition for the calling use case. It must therefore be executed every time a user activates the calling use case. With the "extend" relationship, this is no longer a must, but optional.

Furthermore, the use cases can often have real requirements or requirements that go beyond the system. It is therefore possible to place all system-related use cases in a system context. This is drawn as a rectangle around the relevant use cases.

Below is an example, which is based on our "tree cadastre":

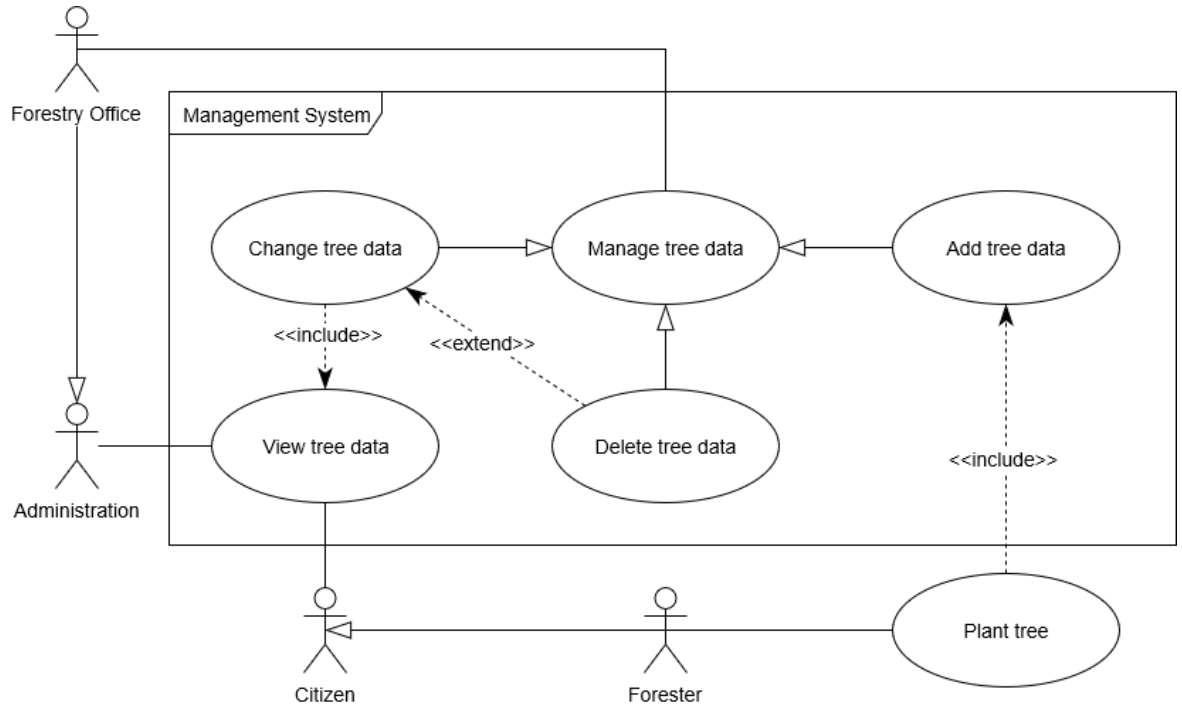

Figure 9: Example of a Use Case Diagram

Let's go through the elements step by step: One type of actor is the administration, i.e. the number of administrative employees. If they do not work in the forestry office, they can only access the data in the tree register. If they work in the Forestry Office, they have better access to the system and can change, delete and add records. However, they can still access the data as administrative staff, so they are a specialization of the administrative actor. Modifying, deleting and adding are all considered management tasks, and therefore fall under the general use case called "managing tree data". To make changes, you must first access the dataset, so there is an "include" relationship here. When adding, it is optional whether the data record is accessed beforehand, for example to check whether the tree already exists. Therefore, an "extend" relationship was used here.

Another kind of actor is the citizen. According to our model, the citizen can only access the data. The forester is abstracted as a special citizen who is allowed to plant trees. If a tree is planted, it must also be included in the data, which is illustrated by an "include" relationship.

Obviously not all possible use cases are modeled here. But this is exactly what is at the center of the modeling: We abstract to the level that is needed. If we wanted to go into more detail, it would be possible to specialize several forest office actors who take on different tasks. One could also model the felling of trees, where the question arises whether the data set should be deleted immediately or whether only one status should be changed to "like".

Outlook

UML offers many more diagrams, which will not be explained at this point. Interested readers can find more in-depth links in the source list. Apart from diagrams, UML offers much more, partly highly complex structures. Thus, for highly formalized systems, the diagrams can be combined with each other, with all diagrams accessing the same pool of elements (classes, actors, use cases, etc.) and illuminating

them from different angles. If you want to model at this level, you usually use specialized tools which allow you to export the models and diagrams directly into exchangeable formats.

If you work close enough to standards and formalisms, there are even approaches that try to derive software directly from UML specifications. If you want to experience the power, but also the complexity of UML in the field of geodata in action, take a look at the **Infrastructure for Spatial Information in the European Community (INSPIRE)** or the **AAA model**, which has been continuously modeled with UML.

Literature

Bill, R. (2016): Grundlagen der Geo-Informationssysteme. 6th edition. Wichmann Verlag. Offenbach-Berlin. 867 pages. Chapter 4.2.

Booch, G., Rumbaugh, J., Jacobson, I. (1999): The UML User Manual. Addison-Wesley. 592 pages.

Internet

A note: There are countless sources for UML modeling, from various technical points of view. However, many of these sources (even Wikipedia!) contain formal errors in their diagrams. In addition, there are many different notation standards that have evolved historically. For most applications, it is not important whether formalisms are followed exactly as long as the diagrams fulfill their purpose. However, if the goal is a formally perfect UML diagram, then the official documentation should always be consulted, however opaque it may be.

https://www.omg.org/spec/UML/ - Current specification

https://de.wikipedia.org/wiki/Unified_Modeling_Language - Overview article

https://de.wikipedia.org/wiki/Entity-Relationship-Modell - See above all different forms of notation

http://inspire.ec.europa.eu/ - INSPIRE