Application of automated software development techniques in Geographic Information Systems

Author: Suilen Hernández Alvarado

PhD thesis UDC / 2023

Advisors:
Oscar Pedreira Fernández
Miguel Ángel Rodríguez Luaces

UNIVERSIDADE DA CORUÑA

Aplicación de técnicas de desarrollo automatizado de software en Sistemas de Información Geográfica

Autor: Suilen Hernández Alvarado

Tesis doctoral UDC / 2023

Directores:
Oscar Pedreira Fernández
Miguel Ángel Rodríguez Luaces

Programa Oficial de Doutoramento en Computación

UNIVERSIDADE DA CORUÑA

PhD thesis supervised by
Tesis doctoral dirigida por

Óscar Pedreira Fernández
Departamento de Computación y Tecnologías de la Información
Facultad de Informática
Universidade da Coruña
15071 A Coruña (España)
Tel: +34 881016028 ext. xxxx
`opedreira@udc.es`

Miguel Angel Rodríguez Luaces
Departamento de Computación y Tecnologías de la Información
Facultad de Informática
Universidade da Coruña
15071 A Coruña (España)
Tel: +34 981167000
`luaces@udc.es`

Óscar Pedreira Fernández y Miguel Angel Rodríguez Luaces, como directores, acreditamos que esta tesis cumple los requisitos para optar a los título de doctor en computación y autorizamos su depósito y defensa por parte de Suilen Hernández Alvarado cuya firma también se incluye.

“Todo lo que la mente puede concebir se puede lograr.”
William Clement Stone

Agradecimientos

Es difícil escribir estas palabras cuando existen tantas personas a las que agradecer en la vida. En primer lugar, quiero dar gracias a Dios por permitirme llegar hasta aquí y escribir las siguientes líneas:

A mis padres y a mi familia, gracias a ellos he aprendido la motivación de seguir siempre adelante.

A Jose, la mitad de mi vida, que fue capaz de dejarlo todo para que yo cumpliera este sueño.

A mi Gastón, por acompañarme en cada momento.

A mis amigos, de ahora y de siempre.

A Gonzalo y a todos los profesores del Master en Ciencias de la Computación de la UDEC, Chile, que hicieron posible el comienzo de este camino.

A Nieves Rodríguez Brisaboa, que confió en mi cuando yo era una desconocida que estudiaba al otro lado del mundo.

A Carmen Cao, por su ayuda y apoyo incondicional.

A mis directores de tesis Óscar y Miguel, por su profesionalidad y guía.

Al Laboratorio de Bases de Datos de la UDC por haber formado parte de esta aventura.

A todas las personas que me han ayudado a llegar hasta aquí.

¡¡¡Gracias!!!

Abstract

Geographic Information Systems (GIS) has been widely adopted within different areas such as infrastructure administration, traffic control or environmental management. Despite each application can be very specific in terms of its functional scope, they share a set of elements that make all systems very similar. These characteristics have made us consider investigating software engineering techniques that may be useful to support the automation of the development of these applications from high-level specifications. Our contributions are as follows: (i) a declarative, close to natural, domain-specific language for the development of GIS, that allows users without deep programming knowledge to specify and generate a basic system; (ii) the theoretical bases for the application of multilevel modelling to GIS-based applications in different real-world scenarios. This can improve the simplicity, expressiveness, and flexibility of the modelling process and (iii) new mutation operators, the automation of the workflow to generate the mutate versions and a new tool to support the application of mutation testing in the GIS domain.

Resumen

Los Sistemas de Información Geográfica (SIG) se han adoptado ampliamente en diferentes áreas como la administración de infraestructura, el control del tráfico o la gestión ambiental. A pesar de que cada aplicación puede ser muy específica en cuanto a su alcance funcional, comparten un conjunto de elementos que hacen que todos los sistemas sean muy similares. Estas características nos han hecho plantearnos investigar técnicas de ingeniería de software que puedan ser útiles para apoyar la automatización del desarrollo de estas aplicaciones a partir de especificaciones de alto nivel. Nuestras contribuciones son las siguientes: (i) un lenguaje declarativo, cercano al natural, de dominio específico para el desarrollo de SIG, que permite a los usuarios sin conocimientos profundos de programación especificar y generar un sistema básico; (ii) las bases teóricas para la aplicación de modelos multinivel a aplicaciones basadas en SIG en diferentes escenarios del mundo real. Esto puede mejorar la simplicidad, expresividad y flexibilidad del proceso de modelado y (iii) nuevos operadores de mutación, la automatización del flujo de trabajo para generar las versiones mutadas y una nueva herramienta para apoyar la aplicación de pruebas de mutación en el dominio SIG.

Resumo

Os Sistemas de Información Xeográfica (SIX) adoptáronse amplamente en diferentes áreas como a administración de infraestrutura, o control do tráfico ou a xestión ambiental. A pesar de que cada aplicación pode ser moi específica en canto ao seu alcance funcional, comparten un conxunto de elementos que fan que todos os sistemas sexan moi similares. Estas características fixéronnos expornos investigar técnicas de enxeñería de software que poidan ser útiles para apoiar a automatización do desenvolvemento destas aplicacións a partir de especificacións de alto nivel. As nosas contribucións son as seguintes: (i) unha linguaxe declarativo, próximo ao natural, de dominio específico para o desenvolvemento de SIX, que permite aos usuarios sen coñecementos profundos de programación especificar e xerar un sistema básico; (ii) as bases teóricas para a aplicación de modelos multinivel a aplicacións baseadas en SIG en diferentes escenarios do mundo real. Isto pode mellorar a simplicidade, expresividade e flexibilidade do proceso de modelado e (iii) novos operadores de mutación, a automatización do fluxo de traballo para xerar as versións mutadas e unha nova ferramenta para apoiar a aplicación de probas de mutación no dominio SIG.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

The main characteristic that differentiates Geographic Information Systems (GIS) from other information systems is that they merge entities with a geoespatial component that plays a central role in the systems functionalities and the processing of data. The geoespatial data of the entities is usually represented with data types such as points, lines, polygons, or variant of these data types. GIS are developed with component and technologies that allow the user to capture, store, analyze and manage any type of referenced geographic information associated with territory. The first GIS was a costly system developed by public administrations or large private companies to support the management of infrastructures. However, the evolution of the technologies for GIS development and the availability of multiple cheap devices with GPS capabilities (such as smartphone, for example) has made GIS much more accessible for many companies.

As a consequence of the widespread utility of GIS-based software and due to factors such as the development of urbanization, smart cities, or the increase in the availability of data in the cloud, there is a high demand for geographic information systems solutions that allow the analysis of territorial data applied to various fields. For example, in transport and logistics management, they have a fundamental role since they allow optimizing times and distances, managing and planning routes, analyzing traffic, etc. Likewise, in the agricultural sector, this type of tool is essential for, for example, the control, recognition, and monitoring of crops or the study of fire zones. GIS also contribute to strengthening the potential of the tourism sector, thanks to these systems it is possible to develop maps, tourist databases, and tools that allow real-time generation of routes and trails of interest, create visit plans, or administer and manage areas of touristic interest.

Also, GIS has contributed to developing sectors such as public administration, in the development of infrastructure management systems such as roads, public buildings, signage, traffic control, etc. Also in environmental management or supply infrastructure such as water, electricity, fuel, or telecommunications. These are some examples of the various applications of use and potential offered by this type of system to solve problems in different sectors of society.

Regardless of the application area of each GIS, these systems share many characteristics and functionalities. When developing these systems there are common requirements that are demanded regardless of the business area to which they are oriented, such as the digitization of geographic data, the capacity to answer database queries based on spatial data, the visualization of geolocated data in viewers maps, structuring map visualization with layers, common map-related tools like pan and zoom, route calculations, etc.

This has been acknowledged by the ISO/TC 211[1] and OGC[2], which have defined a set of evolving standards related to all levels of GIS, from conceptual definition to many aspects regarding the implementation. Therefore, nowadays GIS not only share the functional features but also the models, procedures, services, a common architecture. Currently, for the development of applications based on GIS there are commercial solutions based on frameworks such as ARGIS or GEOSERVER can be used to develop GIS, or a set of architectures, components, tools, and libraries open source based on ISO and OGC standards are also available.

GIS applications are built largely by applying software development techniques that sequentially cover the stages of specification, design, development, and testing in a traditional way. The development of GIS does not start strictly from scratch, since there are many technologies that support this development from the database to the user interface. However, apart from those technologies, the development of a specific GIS application does start from scratch, that is the development teams writes all the code in a traditional way, without applying automated or semi-automated development techniques. However, although each application can be very specific, in term of its functional scope, there is a set of common elements that make all GIS applications very similar. These characteristics have made us consider investigating software engineering techniques that may be useful to support the automation of the development of these applications from high level specification.

The goal of this thesis is to develop techniques that allow the development of Geographic Information Systems to be automated as much as possible. The motivation that leads to this objective is that we believe that this is possible given the similarity and the number of elements in common that all these systems have. In previous works this approach was addressed with SPLE, however in this research, we have considered addressing it with MDE, in addition, we have also paid attention to the testing phase, addressing this process through the automation of the tests.

Two of the software engineering areas focused on automating the software development process are Software Product Line Engineering (SPLE) and Model-Driven Engineering (MDE). SPLE apply the principle of "series production" to the development software to create families of systems that share common characteristics [CN15]. In this way instead of developing each of these systems individually, they develop platforms that automate the creation of software systems aimed at a specific market segment. In SPLE, systems of the same family are modeled but considering the variability of each of the products individually. From a selection of the characteristics that will be present in each system, these are generated in the platform in an automated way through the integration, adaptation, and

[1]International Organization for Standardization committee on Geographic information/Geomatics: `https://committee.iso.org/home/tc211`

[2]The Open Geospatial Consortium: `http://www.opengeospatial.org/`

configuration of a set of core assets. This technique, was applied in [CLP+17, CLPP17] to create web-based GIS integrating MDE and SPLE concept. On the other hand, the MDE [BCW17a] is a paradigm centered on two fundamental concepts: abstraction, by specifying the system to be developed in high-level abstraction models and the automation of the transformation processes of these abstract models to models closer to the source code of the system.

An approach based on a similar idea that of Domain-Specific Languages (DSL). In this case, instead of specifying the system through a model, we specify it using a high-level language that manage concepts specific to a given app domain and that allow us to delineate in a formalized way software systems of specific application domains. Thus, an expert using this formal specification language in a specific domain can achieve not only a formal representation of the system to be created but also have a better understanding of the situation to be solved [Fow10].

A DSL concerning a general-purpose language provides us with multiple benefits to represent and model solutions. For example, one of these advantages is less complexity of understanding for users, since it focuses on expressing the concepts and terms of a specific domain that are directly aligned with how domain experts express themselves. These languages have a greater expressiveness of solutions within their domain so they can improve the productivity, maintainability, and reusability of the applications. When writing a solution using a DSL, as long as the language constructs are correct, any written statement can be considered correct since the validations are performed at the domain level. Currently, there is a great variety of DSL that have been specified to cover different purposes, some of the best known in the area of informatics are for example the SQL language for the definition of queries, HTML to define web user interfaces, XML for transport or persist data, REGEX to form search patterns or character substitution, WSDL to specify web service data interfaces, CSS to define the appearance of user interfaces at the presentation level, among many others, which facilitate the way to work.

Part of our research has focused on defining a DSL to represent and describe the GIS domain. In our proposal, we have focused on representing the most used basic spatial data types in this domain, which serve as a basis for generating any functional application. In our DSL it is possible to write declarative sentences that allow us to represent maps, layers, and entities without the need to emphasize how to implement these elements. In chapter chapter 3 we go into this topic in-depth and illustrate several use cases of our proposal.

Another idea on which we have focused our attention in this thesis is to apply multilevel modeling to the development of GIS. In previous works [CLP+17, CLPP17], this domain has been modeled in the traditional way posed by MDE. This traditional form considers two levels of abstraction, first, a metamodel is created with the most representative elements of this type of application and from this metamodel, the model instances are generated that will be later transformed into source code. Based on these proposals we have considered reformulating the GIS modeling following an approach that instead of having two levels, has several. We have applied this technique in order to show the modeling flexibility, simplicity, and expressiveness that this type of design can have in our domain. In chapter chapter 4 we go into detail about this modeling technique and show three scenarios that we consider representative of GIS applications that illustrate the benefits of using multilevel modeling.

Finally, we have focused our attention on the software testing stage and on how we could benefit the domain of GIS applications with techniques that help improve the quality of these types of applications. For this, we have paid attention to how to automate the testing process, in our case we have investigated the mutation testing technique [DLS78, Bud80]. This technique is a type of white box test and its objective is to introduce artificial errors in the source code, to generate faulty versions of the system and thus implement good sets of test cases that detect these errors. These artificial errors are introduced to run our test suite and see which of these errors are detected. If the errors we introduce make sense because a programmer could make them, our tests should detect them. If there is an undetected error, this will give us information on how we can improve our test suite.

In our case, we have studied the main technologies and tools used to build GIS-based applications and based on this analysis, we have proposed a set of mutation operators that simulate the most common errors made by programmers in the development stage.

We believe that advances toward this goal will provide with tools, techniques and technologies that world allow to design and develop GIS with less effort, and with higher quality levels. Therefore, the **main goal** of this thesis is to investigate the application of automated software development techniques in GIS so that the creation of tools that facilitate its development.

1.2 Contributions

In this thesis, we have focused on the investigation of automated software development techniques applicable to GIS domain and that contributes to automating the development stage of GIS-based applications. Our contributions are as follows:

1. **Implement of GIS with a domain-specific language**: Our first contribution is the definition of GIS-DSL, a declarative domain-specific language for the development of GIS. This language allows the developers to specify the system simply by specifying its components at a conceptual level with entities with a spatial component, map, and layers. Also, GIS-DSL is declarative and close to natural language, so even users without deep programming knowledge can specify and generate a basic system. From the system specification using GIS-DSL, it is possible to generate the source code of the basic functionalities for managing the defined elements and their visualization.
2. **Modelling of geographic information system with multilevel modelling**: Our second contribution is the proposal of the application of the multilevel modeling approach to GIS-based application. We have applied this approach to different scenarios that versus two-level modeling we have identified benefits in terms of simplicity, expressiveness, and flexibility of modeling both types and entities, where it is possible to add these elements dynamically, as well as instantiate elements or relationships in levels lower models as required by business rules. In addition, defining the models at the lowest level of abstraction would require less effort, since we would not need to repeat structures that could be moved to upper levels and more understandable solution.
3. **Mutation-based testing of geographic information system**: Our latest contribution is the application and automation of the mutation testing technique in the GIS domain. Three derived contributions are: *(i)* the definition of new mutation operators for GIS; *(ii)* The automation of the workflow carried out to generate the mutation operators and generate the mutated versions of the system using the

operators that were defined; and *(iii)* The implementation of a tool supports this process. As proof of concept, we developed 2 case studies where we use two real applications and we apply the defined mutation operators and generate the mutants with our tool.

1.3 Thesis Outline

The rest of this thesis is structured as follows:

- In Chapter 2, we provide a brief introduction to GIS and their development, including the architecture, tools, and technologies most used for its development.
- In Chapter 3, we present GIS-DSL, a domain-specific language for the development of GIS. GIS-DSL is a declarative language that allows the developer to define the entities, relationships, maps, and layers of the system.
- In Chapter 4, we present and discuss the idea of applying multilevel modeling to the development of geographic information systems.
- In Chapter 5, we address the mutation-based testing of geographic information systems (GIS), by proposing a set of mutation operators that address the typical errors that can be made with the technologies used to develop these systems.
- In Chapter 6, we present the conclusions on the results of the research done in this theses and possible lines for future work are discussed.

Chapter 2

Background and state of the art

This chapter presents the general context in which this thesis is developed and a description of the state of the art relevant to it. In each section, we describe the background of the techniques applied to geographic information systems.

2.1 An overview of GIS

Geographic information systems is a field focused on information systems with geospatial characteristics and capabilities that allow us to represent and manipulate knowledge regarding geographical information [WD04]. GIS were traditionally used by some organizations, such as public institutions, for territorial administration, but since the major advances happened in communication technologies, more and more companies and organizations in many fields and domains are adopting GIS solutions to improve their workflows.

GIS were traditionally used by public institutions for administering the territory and managing public resources. Lately, the major advances in communication technologies and the technologies used in GIS development have increased the availability of GIS applications, and organizations from many domains are adopting GIS software. Moreover, the appearance of smartphones with GPS capabilities marked an important milestone in the development of GIS because gathering geospatial information is now cheap and easy for any company. This context has made it mandatory in some application domains to use a GIS-based solution to be competitive. For example, in warehouse logistics, GIS are needed to plan transportation routes in the most efficient way; in public transportation, to know which lines are overused or underused, and to decide how to change them accordingly; or even in social networks and advertisement, since knowing the position of the users and their publications enhance the information they collect to improve their algorithms or to enrich the data that afterward is used by ad services. Regardless of the application area or the purpose of each GIS, there are a set of features that are very common among them, such as digitizing geographic data, representing geo-located data in map viewers, the common tools related to these map viewers (from panning and zooming to measuring dimensions or objects within the map, or sorting the different layers), route calculation, etc.

GIS have changed a lot since they first appeared many decades ago [Tom69]. At first, for a long time, each GIS application was developed ad-hoc, totally independent from any other GIS application, and even when the functional features provided by the systems were quite similar, the concepts behind GIS, such as the definition of what is a *polygon*, were different among several systems. The problem of that approach is that interoperability between GIS was not addressed in any way. Nowadays, that situation has changed thanks to two organizations, the ISO committee on geographic information/geomatics (ISO/TC 211[1]) and the Open Geospatial Consortium (OGC[2]), which have defined a set of evolving standards related to all levels of GIS. Most GIS software assets follow these standards and, therefore, GIS applications are quite similar, geographic data can be used in different GIS, and GIS components are, in general, interoperable.

For example, the standard *ISO 19107: Geographic Information - Spatial Schema* [Inta] defines all the geographic data types, such as *Point*, *Line*, *Polygon*, or *Sphere*, and all the GIS related operations, such as the predicates *intersects*, *overlaps*, or *within a specified distance*. OGC also defines a set of web services that are widely used and supported by map servers and map viewers. Some of the most important ones are the *web map service* [Thec] and the *web feature service* [Theb]. Regarding architecture, the standard *ISO 19119: Geographic Information - Services* [Intb] identifies architecture patterns for service interfaces and the relationship between them, proposes a geographic services architecture, and provides some guidelines for the selection and specification of geographic services.

These standards are followed by most GIS software products and libraries, and therefore most GIS applications are quite similar. In the case of web-based GIS applications, the similarities affect even to the specific software assets used, since most of them are widely popular, such as GeoServer, a map server that provides most of the standard services, or OpenLayers and Leaflet, two map viewer libraries.

The main characteristic of Geographic Information Systems (GIS), which is also their main difference from other information systems, is that they manage entities with a geo-spatial dimension. That is, an entity is defined by a set of attributes, each one being of a particular type. Commonly used data types, such as *String* or *Integer*, store alphanumeric information. In a GIS, there are specific data types to store a geometric structure in the space, like a point or a surface in a specific position in the world. The attributes of geographic data types can store, for example, the location of a building or a meteorological station, the paths of a road network, or the area covered by a forest. Having entities with such attributes allows an application to visualize them with maps instead of typical alphanumeric listings, but it also allows an application to make certain operations or analysis with these geographic data, such as getting the ten closest entities to a user position, or the most efficient path across a set of entities. Summing up, GIS have particular features and use specific technologies that allow us to collect, store, process, and visualize spatial information [WD04].

[1]`https://committee.iso.org/home/tc211`
[2]`https://www.ogc.org/`

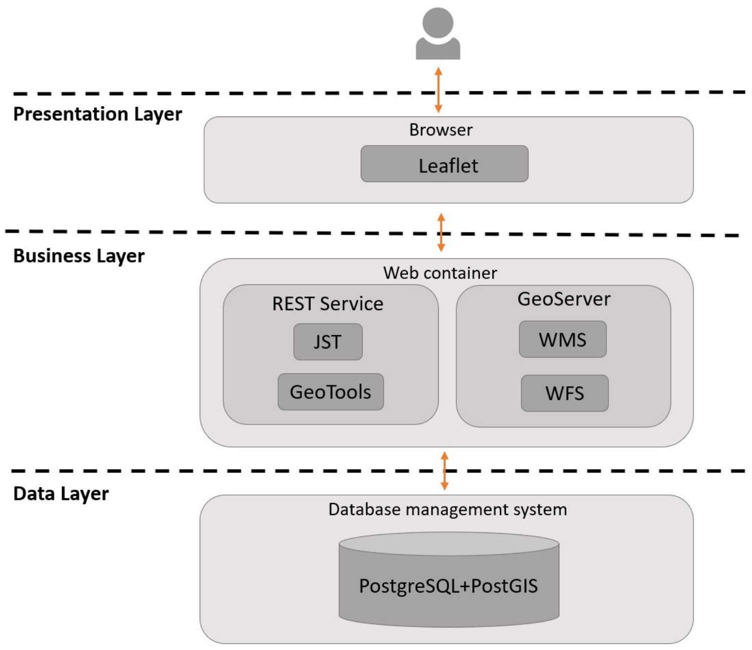


Figure 2.1: GIS Application, generic architecture

2.1.1 GIS Architecture

Figure 2.1 shows the generic architecture of a GIS application, which is divided into three layers, described below:

- *Presentation layer:* this layer implements the user interface that allows visualizing and managing the geographic information. The most used library to support this layer is nowadays Leaflet, which acts as a client of the standard cartography publication services (WMS and WFS).
- *Business layer:* this layer supports the processing of user requests related to spatial operations, analysis techniques, and generation of response information. The components of this layer, such as the Java Topology Suite or GeoTools, implement operations that allow the spatial part of the entities to be processed. For example, operations can be performed to check whether two geometries meet a certain spatial predicate, or new geometries can be created from others that we already have stored in the database. The components of this layer generate results that will be displayed in the view layer through services such as WMS and WFS.
- *Data layer:* This layer includes the Database Management System (DBMS), files belonging to the application, and external data sources. In this layer, the most used DBMS for storing and querying geographic information is nowadays PostgreSQL

together with the PostGIS extension. These systems support most of the functionality required in a GIS regarding data storage, such as data types, operations, and indexes to represent and query geographic information in the database in a robust and efficient manner.

2.1.2 GIS Technologies

As we have already pointed out, there is a large set of libraries and frameworks that can be used for developing a GIS. We have focused on the most popular technologies used to implement a GIS based on the architecture we have just described. The view layer is usually implemented with web libraries such as *Leaflet*[3]. However, parts of the system not implemented in Java are out of the scope of our analysis.

- *Leaflet* is a library to visualize and manage geographic information in the user interface layer. This library is used for the creation and presentation of interactive maps in Javascript-based Web applications. Acts as a client of standard cartography publishing services.

  We have analyzed class structures such as *FeatureJSON*, *GeomUtils* or *FeatureCollection*. These classes contain methods that enable interface-services connectivity. These methods manipulate the exchange of data in GeoJSON format through REST and WFS services.

- *Java Topology Suite (JTS)* is an open source Java software library for the processing of linear geometries in two-dimensional Cartesian spaces. It provides a model of spatial objects and 2D geometric functions. This library is used in the development of geospatial applications. JTS implements a set of functionalities to support and manipulate objects that have a spatial component, as well as algorithms and data structures used in the implementation of geometric operations.

  The class structures *GeometryFactory* and *Geometry* has been analyzed. These structures implement geometry types such as *Points*, *MultiPoints*, *LineString*, *Polygons*, etc., and contain geometric operations such as spatial predicates, overlay functions, metrics, and other that allow you to manipulate these types of objects.

- *PostGIS* is an extension for the PostgreSQL relational database, which adds features such as spatial data types, spatial indexes and functions that operate on them. In this way, PostgreSQL becomes a spatial database and expands its capabilities for spatial support allowing to manipulate, store and query vector geometric objects.

  The classes *Connection*, *PreparedStatement*, *Statement* and *DriverManager* has been analyzed. These classes contain methods such as *executeQuery (..)*, *prepareStatement (..)*, *prepareStatement (..)* or *getConnection (..)*, which enable connection to the database server and the realization of spatial queries.

- *Geoserver* is an open source server for sharing geospatial data. This J2EE application provides two geographic information publishing services in the Web layer following the open Web Map Service (WMS) and Web Feature Service (WFS) standards defined by the Open Geospatial Consortium.

[3]https://leafletjs.com/

- *GeoTools* is an open source Java software library for working with geospatial data. This library is used to the development of GIS applications. Classes like *FeatureJSON*, *GeometryJSON* or *GetMapRequest* has been analyzed, that handle geometries in GeoJSON format through REST or WFS services.

2.2 Model-driven software engineering and Donamin Specific Languages

Model-driven engineering (MDE) is a software development approach that promotes the use of models as active artifacts in all stages of software development [PM07a, BCW17b]. In MDE, high-level models are automatically transformed into models at lower levels of abstraction, and finally, into the source code of the system (or part of) according to a set of transformation rules. The goal of this paradigm is to produce software following an approach similar to that of other traditional industries. One of the approaches within MDE is *model-driven development* (MDD) [BCW17b], which defines models as the main artifact for modeling software systems at a level of abstraction higher than the allowed by programming languages. The goal of this approach is to increase the levels of automation, quality, and productivity. A *domain-specific language* (DSL) is based on a similar idea. Instead of specifying the system through a model, we specify it using a high-level language that directly supports the main elements of the application domain. These approaches lead to a reduction of the development effort and the number of errors introduced in the implementation since part of the source code is generated automatically.

A standard defined by the Object Management Group (OMG)[4] on its particular vision of MDD is the *model-driven architecture* (MDA)[5] [PM07a], structured into four layers: CIM (computational independent models), PIM (platform-independent models), PSM (platform-specific models), and ISM (implementation-specific model). These models can be defined using general-purpose modeling languages, such as UML, or domain-specific languages.

A domain-specific language is a high-level language designed for software development in a specific application domain. Some of the definitions of a DSL are: According to Voelter, a DSL is “a language that is optimized for a given class of problems, called a domain. It is based on abstractions that are closely aligned with the domain for which the language is built” [VBD+19]. Fowler defines a DSL as “a computer language that’s targeted to a particular kind of problem, rather than a general-purpose language that’s aimed at any kind of software problem” [Fow10].

DSLs are part of many of the IT tools we use every day. The evolution of a solution to a problem goes through different stages. First, we make a first draft of the solution without any clear guidelines, then as we gain experience and knowledge of the business we can align ourselves with good practices that allow us to have portability and scalability. In later stages of maturity, we can abstract these good practices and turn them into patterns that solve various problems focused on the same application domain. These patterns can be combined to form a framework, which can finally be increasingly configurable and

[4]OMG: http://www.omg.org
[5]MDA: http://www.omg.org/mda

automated and evolve to what we know as DSL. There are many domains that can be modeled and specified based on their business rules, validations, and concepts.

The difference between a DSL and a general-purpose programming language is that a DSL allows us to work directly with domain-specific concepts and constructs, which leads to a greater expressiveness [MHS05, Fow10, Fra13]. Although implementing a DSL can require a significant effort, their main benefit is that they allow us to specify/implement a system with significantly less effort.

The Figure fig. 2.2, based on [TA19], shown the main elements that contain a DSLs. The abstract syntax specifies the structure, constructs, connectors, and properties that the language can implement. The concrete syntax specifies the notation with which users of the language will be able to use it. Static semantics or formation rules define the restrictions on the abstract syntax. Semantics defines the meaning of concepts in the abstract syntax.

There is a way to categorize DSLs, based on your implementation approach, internal and external DSLs. An external DSL is a domain-specific language represented in a separate language to the main programming language it's working with. This language may use a custom syntax, or it may follow the syntax of another representation such as XML. An internal DSL is built on top of a General-Purpose Language(GPL), such as Python, CSharp or Java. In this case, libraries and frameworks that implement DSL can be built on the syntax and native from the base language [SZ09b, VDE+].

2.3 Automated Testing

Another approach that we have addressed is the automated testing, in this case, we have focused on the *Mutation-based testing.* This technique involves artificially and intentionally injecting errors into a system under test (SUT), running a test suite on the SUT, and checking which of those errors are detected by the test suite [DLS78, Bud80]. A *mutant* is a version of the SUT in which a change has been injected which, in most cases, will produce, under certain circumstances, some difference in the behavior compared to the behavior of the SUT. If a test case can detect the error, we say that it *kills* that mutant. In this context, the effectiveness of a test suite is measured as its *mutation score*, which is defined as the percentage of mutants killed by the test suite. The higher the mutation score, the more effective the test suite. If the test suite kills all the mutants, then it is said to be *mutation-adequate.* If, in addition, the test suite does not find errors in the SUT, then we can be relatively sure about the quality of the SUT. However, this also depends on other factors, one of them being the quality of the errors injected to create the mutants.

Mutants are generated by injecting errors in the SUT through the application of *mutation operators.* The quantity and quality of the mutants depends on the number of mutation operators we apply to the SUT and the variety and interest of the errors they can introduce. A mutation operator should try to introduce an error that programmers are likely to make during the development. Two of the principles underlying mutation testing are (*i*) the *coupling effect*, that is, if a test case is sensitive enough to find the simple errors, then it will also find more complex errors, and (*ii*) the *competent programmer*, that is, a good programmer writes almost optimal programs, making only small mistakes.

Mutation operators can be general-purpose or specific to a given domain or technology. General-purpose mutation operators [OLR+96, MOK05] affect the most basic elements of programming languages, such as arithmetic and logical operations, for example, and they

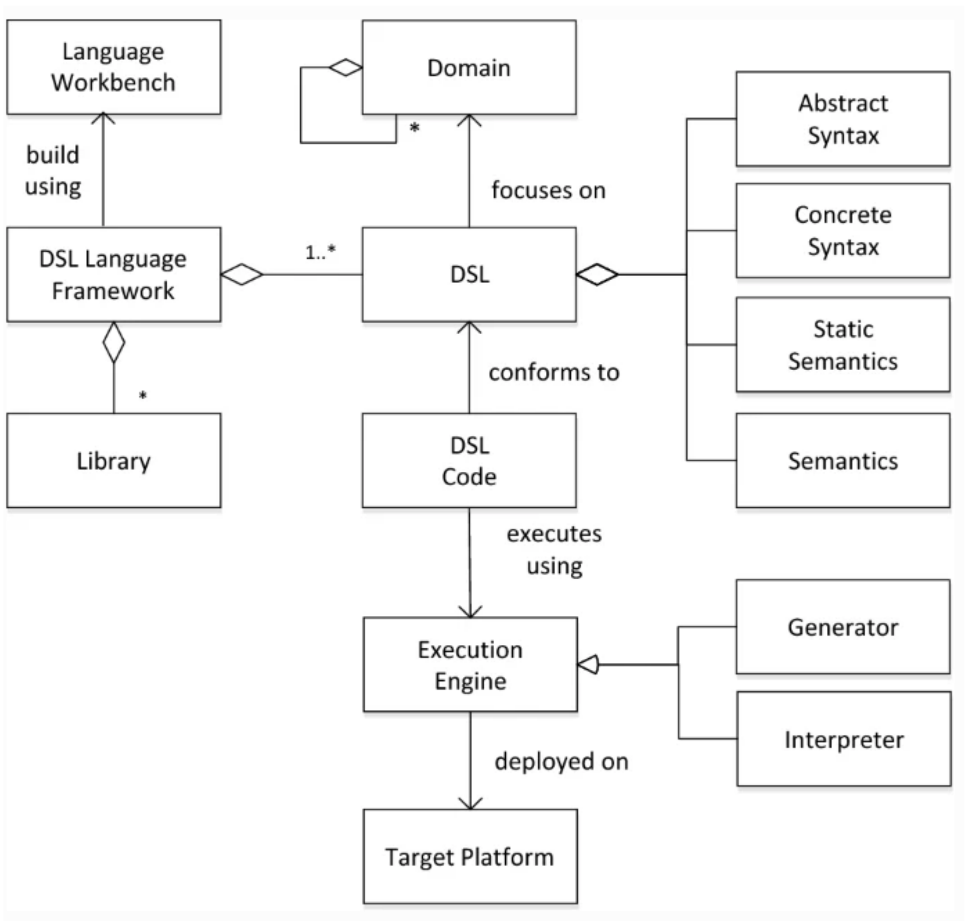


Figure 2.2: Basic elements of DSLs. Based on [TA19],

try to reproduce typical errors when using those basic operators. For example, a mutation operator could replace a $\leq$ operator by a $<$, since this is a typical error that a good test suite should detect. Most of the existing mutation operators in the literature fall in this category. General-purpose mutation operators are considered a traditional technique, but they are not able to reproduce errors specific to a certain technology or application domain. Technology-specific mutation operators try to reproduce errors that are probable when using a specific technology, so they enrich the variety of errors that we can inject into the system.

The original idea of the mutation-based testing was proposed by Richard Lipton [DLS78], and developed by Budd et. al. [Bud80]. Subsequently, this idea has been extended and mutation operators have been defined for programming languages such as C [ADH+89], C# [Der06], PHP [SZ09a], Python [DH14], [DPMBDJ+15] or Java [MO05a, MO05b].

Mutation operators can be classified into general-purpose operators, and operators specifically designed for a given programming paradigm, technology, or domain. General-purpose operators are applicable to any programming language [OLR+96] and address the injection of errors in operations that can be used in any system, such as arithmetic and logical operators. Other mutation operators have been developed for specific technologies, such as, for example, object-oriented programming [MOK05].

Many of these operators have been implemented in different mutation analysis tools such as: MuJava [MOK05, MOK06], Jumble [IPT+07], Judy [MR10], Major [JSK11, Jus14] or Bacterio [MU12]. Given the need to facilitate the incorporation of new operators to the existing catalogs of these tools, scalable architectures have been designed to facilitate their inclusion [URRH17].

Recently, mutation tests have been addressed through the paradigm of aspect-oriented programming. This paradigm has been used to intercept and manipulate the source code of a system, expressing the mutation operators as *pointcuts*. The first research that used this approach [BW06a, BW06b] within the technique of testing based on mutation were based on a prototype to simulate the mutations through the interception of the calls to the SUT that are made from the cases test, altering the return values of the methods. Subsequently Polo Usaola [Pol14], expresses the mutation operators as aspect files to capture the SUT source code and modify its behavior at runtime.

In the state of the art [JH11] we have found mutation operators specific to different technologies. Saleh et. al. [SN14] propose a scalable cloud-based mutation testing framework that reuses the MapReduce programming model to improve the performance of mutant generation and testing of large-scale software projects. Cañizares et. al. [CNM18], implement an mutation testing framework to detect errors in distributed applications running in simulated environments.

Tuya et. al. [TSCDLR07] propose a set of mutation operators to modify the behavior of SQL queries (SELECT) that recover data from a database. These operators were designed to cover different features of this language such as mutations related to the handling of NULL values, replacement of column reference identifiers, or mutations to the main SQL clauses. Later, Zhou et. al. [ZF09] extends the mutation testing approach for SQL by Tuya et al. to adapt it to Java applications that interact with a database through the JDBC API (Java Database Connectivity). This research describes the mutation testing tool for Java JDAMA (Java Database Application Mutation Analyzer) programs, which perform mutation testing on SQL statements executed by JDBC.

Operators oriented to mobile technologies have also been developed. Deng et al. [DOAM17] proposes 11 mutation operators to test various features of Android applications such as source code, XML files, or permissions. Saifan et. al.[SA20] define 42 mutation operators and a tool to generate mutants according to the features of android applications: apps with user location and maps, apps with multimedia, apps with graphics and apps with content sharing, Linares-Vasquez et al. [LVBT$^+$17] also proposed introduced 38 mutation operators for Android apps and Polo Usaola et. al [PURT21] analyses how the combination of different cost reduction techniques improves the execution time of mutation testing in mobile apps.

The mutation technique has also been addressed to mutate written programs in the aspect paradigm. Ferrari et. al. [FRM13] define a set of mutation operators for specific AspectJ constructs along with the implementation of a tool that automates this approach.

Another context in which mutation testing has been developed is in the detection of memory bugs. Nanavati et al. [NWH$^+$15, WNH$^+$17] study memory faults and propose 9 memory mutant operators targeting common faults.

2.4 Developing GIS with SPL

Model-driven engineering (MDE) is an approach to software development in which models play a central and active role, far beyond just describing the system. In MDE, models describe the software system and are artifacts that can be processed to be successively and automatically transformed into models at lower levels of abstraction, and, finally, into the source code of the system [BCW17c, PM07b]. A common approach to MDE is based on the OMG's[6] *model-driven architecture* (MDA)[7], which defines four layers: computational independent models, platform independent models, platform-specific models, and system code. The OMG also defined a standard for *meta-object facility* (MOF), that defines the way to create *domain-specific modeling languages* (DSML), usually, through meta-modeling based on two levels of abstraction.

The traditional approach to MDE considers two-levels. In the most abstract one, a metamodel defines the main concepts of the domain. In a lower level, a domain specific modeling language can be defined from the metamodel to allow the designer to create models of the system. A promising trend within MDE is that of *multilevel software modeling* [AK01, AK03, AK08]. In contrast to a more "traditional" approach, multilevel modeling does not constrain the number of levels, so the designer could use the number of levels that better fit a particular domain. This approach aims at simplifying the complexity of the models through the separation of specific domain concepts that can be modeled at several levels. Multilevel modeling solves some drawbacks and restrictions that can occur in the traditional two-level modeling [dLGC14]. As explained in [Fra18], many modeling languages for this purpose have been proposed and, although they are different in some elements, they all share common features, such as considering that all classes at any level are also objects and allowing for deferred instantiation of attributes.

De Lara et al. present in [dLGC14] a research work focused on when and how to use multilevel modeling in software development. The authors mention that "unfortunately,

[6]Object Management Group: http://www.omg.org
[7]Model Driven Architecture: http://www.omg.org/mda

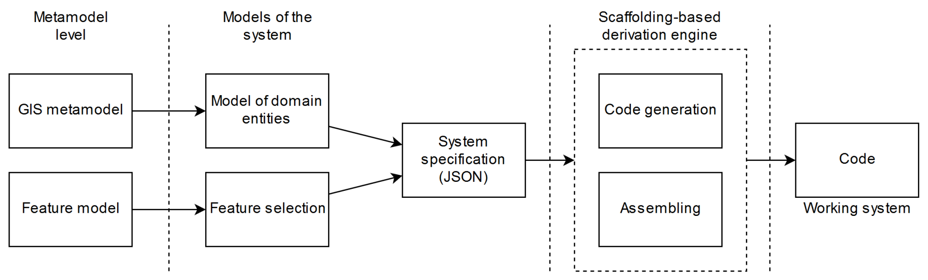


Figure 2.3: Tool architecture

there are scarce applications of multilevel modeling in realistic scenarios [. . .]". After analyzing a large set of metamodels from different sources, they identified many domains in which a multilevel approach could be more beneficial than a two-level approach, and they also identified a set of patterns where multilevel modeling may bring advantages.

In [CLP+17, CLPP17] we have already considered the application of automated software development to the GIS domain and we proposed an architecture and a tool for this purpose combining SPLE and MDE approaches.

Software product lines engineering is a field that pursues the industrialization in software development by applying the same processes that were carried out in the factories industrialization, such as reusing components and assembling specific products from a selection of features from the whole set supported. This whole set of features supported by a product family is represented by feature models [SHTB07]. The first step to apply SPLE techniques to GIS is to analyze different GIS products and extract the set of possible features of these products, classifying these features into common features or variability. Each one of these features must be implemented with a software asset or component. In fact, most of the features related to processing geographic information, such as *importing a shapefile*[8] or *geocoding postal addresses*, are implemented in encapsulated components that provide interfaces and that can be used "as they are" independently of the domain or the particular product, maybe with small variations such as *supported file formats* or *using Google Maps geocoding*[9] *or OpenStreetMaps Nominatim*[10]. Therefore, from a specification of the features to be included in the system, we can build the application combining and configuring the components related to those features.

Although the features and components are common to the family of GIS products, there is one caveat: the data model depends on the particular domain and, even when there are some models that appear frequently, it is required that each product is defined by means of the data it should support with total flexibility. The only difference between the data model in GIS and other applications is that in GIS we need geographic data types.

Generating code from models was addressed with model-driven engineering, so we defined a metamodel to specify how the data model of the products of our family is

[8]Shapefile reference: `https://doc.arcgis.com/en/arcgis-online/reference/shapefiles.htm`
[9]Google Maps geocoding: `https://developers.google.com/maps/documentation/geocoding/intro`
[10]OSM Nominatim: `https://nominatim.openstreetmap.org/`

described.

Figure 2.3 shows the architecture of our tool for the automatic generation of GIS using SPLE and MDE. In the metamodel level, our design combines two metamodels: the first one defines the entities of the GIS domain, allowing to model entities with a geographic component and the relations between them; the second one is a feature model [SHTB07] which contains the features that we can select for a specific product and the set of constraints between them. The metamodel considers the definition of entities with their attributes and relationships, with the particularity that spatial data types can be used. That is, the MDE part of the platform presented in [CLPP17, CLP$^+$17] allows the designer to create models with entities using spatial data types. To generate a product, these metamodels are instantiated into models, defining both the data model and the selection of features of a particular system. These two models combined are the system specification, which in our case is represented by a JSON document. This specification is processed by the derivation and code generation engine that finally generates the source code of a working system.

Our tool, and the metamodels it handles are very flexible and complete, and they allow us to define GIS applications for any domain. For example, if we need to develop a product handling public resources, we can define entities representing hospitals and other kinds of medical centers, education buildings such as universities or schools, the public bus transportation network, the sections of a water supply network, etc. However, this design following the two-level modeling approach has some caveats. First, it would force us to define all possible elements of a GIS in a single metamodel. This would not allow us to reflect in the model or set of models of the system common situations in this domain, such as defining entities that refine other entities at higher levels of abstraction. Second, since GISs manage entities with a spatial component in the real world, it is relatively common that some structures appear repeatedly with some adaptations and particularities. Working with a two-level approach does not allow us to take any advantage of this scenario.

As we will explain in section 4.2, these disadvantages can be addressed by applying a multilevel approach. For example, let us assume we need to develop a GIS that allows a city manager to handle the road networks, the public transportation networks and also the electricity, water, and telecommunication supply networks. As we will see in the next section, all these networks share the same structure, although they may differ in specific attributes of the network elements. Our proposal shows that applying a multilevel approach allows us to metamodel, at an intermediate level, the most common GIS structures, so we can use these structures to make simpler models in the lower level.

Chapter 3

A Domain Specific Language for Web-based GIS

This chapter presents GIS-DSL, a domain-specific language for the development of GIS. GIS-DSL is a declarative language that allows the developer to define the entities, relationships, maps, and layers of the system. According to this specification, a software tool then generates the source code of a GIS supporting the management of all those elements. We present the metamodel, the language, a use example, and a case study in which we develop two sample applications to analyze the resulting software products. This research has been published as a conference and journal paper in [ACL^{+}19] [REF].

The rest of the chapter is structured as follows: in section 3.1 we review background and related work. In section 3.2 we present GIS-DSL, including an analysis of the architecture and main components of a GIS, the metamodel, the language, and a use example. section 3.3 details the implementation of the tool that allows us to transform a system specification in GIS-DSL into source code. section 4.3 presents a case study on two sample applications. Finally, section 5.5 presents the conclusions of the paper and lines for future work.

3.1 Background and Related Work

A wide variety of DSL have been developed in different domains. For example, in software engineering, DSLs have been proposed to support the process of generating source code for desktop-based database applications in Java [LK11], or to model performance tests for web applications [BZR16]. In [DSDS16] describes DSL3S, a domain specific modeling language for Spatial Simulation in the field of GIS that synthesizes relevant concepts of spatial simulation in a UML profile. This Profile forms a graphical language, and its companion MDD3S framework, that involves a modeling and a model-to-code transformation infrastructure allows the design of simulation models through the arrangement of graphical elements.

The systematic mapping presented in [KBM16] highlights some open lines of DSL research. For example, this mapping revealed that most articles focus on the design and implementation of DSLs, but few of them considered aspects such as validation and usability evaluation, domain analysis, or maintenance.

In previous works [CLP$^+$17, CLPP17], it has been explored the automated development of GIS through a combination of *software product line* (SPL) technologies and basic MDE techniques applied to the generation of the database and data model. The platform allowed the user to define the data model of the system, and to specify a selection of optional features that could be included in the final system. In this chapter, we further explore the application of MDE techniques for the development of web-based GIS through the definition of a DSL that considers the definition of the domain geospatial entities, and also how they will be visualized in the web.

The application of MDE techniques to GIS development has been explored in previous works. For example, [LFSNdVB10] and [SNF10] presented an UML profile to support GIS-related concepts in UML conceptual models. Later, [JFD$^+$13] presented a work in which that UML profile was used in a MDA architecture to generate the SQL code for the creation of spatial databases. Many GIS standards (such as those from ISO, OGC, and the INSPIRE[1] initiative) include metamodels that cover different concepts and application areas. Kutzner [Kut16] addressed the model-driven transformation of geospatial data according to different metamodels that can present differences between them.

3.2 A Domain-specific Language for Web GIS

Although some GIS applications are developed for desktop, the web has become the preferred choice. The ISO and the OGC have defined a set of evolving standards that define most of the aspects for the GIS domain, including models, procedures, services, and architectures. We can see the focus on the web in these standards since many network-based services were defined, such as the *web map service*[2] (WMS) or the *web feature service*[3] (WFS).

In this section, we present a DSL for web-based GIS. Its main characteristics are: *(i)* it allows the developer to specify the entities to be managed and how they will be visualized in the web through layers and maps, *(ii)* it is a declarative language, that is, the developer specifies the entities of the system and how the data will be visualized, without having to implement any details related to these features.

3.2.1 GIS Architecture and Main Constructs

A GIS manages entities with spatial properties such as points, line-strings, and polygons. A *Point* is defined by its *latitude* and *longitude* and represents a position in the space. A *LineString* is a set of joined points, and it is commonly used to represent objects such as roads or pipes, for example. *Polygons* are used to represent areas, such as divisions of the territory. In some cases, we need to work with collections of these basic spatial types. These collections are supported by the data types *MultiPoint*, *MultiLineString*, and *MultiPolygon.* The *Geometry* data type is a superclass of all these types. Figure 3.1 shows

[1]`http://inspire.ec.europa.eu`
[2]`http://www.opengeospatial.org/standards/wms`
[3]`http://www.opengeospatial.org/standards/wfs`

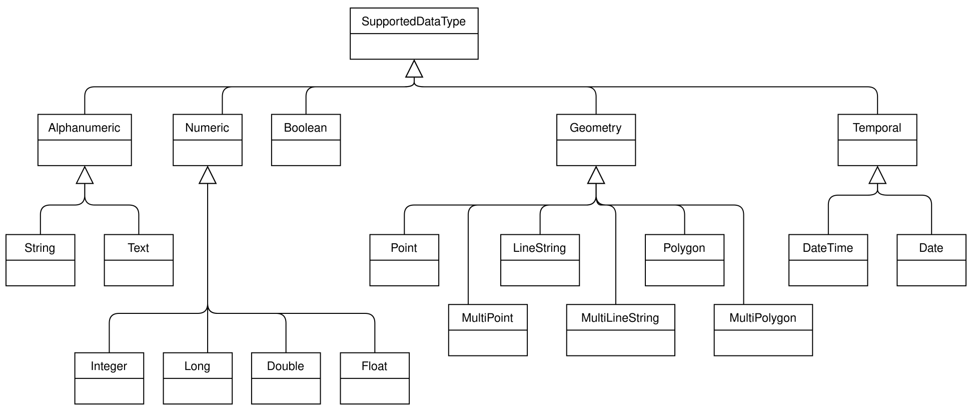


Figure 3.1: Supported data types for our metamodel of web-based GIS.

the data types supported in the metamodel of our DSL, which includes common data types (numbers, booleans, strings, and dates) and the spatial data types we have mentioned.

The GIS domain is large and other data types exist. The standards by ISO, OGC, and INSPIRE include a large number of metamodels. Previous works addressed GIS metamodeling too. For example, the UML profile presented in [LFSNdVB10] and [SNF10] includes data types to represent networks and other spatial phenomena. In [Kut16], Kutzner addressed the model-driven transformation of geospatial data according to different metamodels. We have decided to include only the basic spatial data types in our metamodel as we believe they are the most common to any GIS application, and the purpose of the metamodel is to serve as the basis for the design of a DSL that will allow to automatically generate base functional applications that can manage entity types with a spatial component. However, both the metamodel and the DSL we present could be easily extended with other data types and constructs.

Spatial properties need to be defined within a *spatial reference system* (SRS) that defines the map projection used by some spatial data or by a map viewer. Using a specific SRS is required to transform coordinates into the actual position of an object. Depending on the spatial context for which a GIS is built, we may prefer to use one SRS or another.

Spatial data types and operations are supported by specific tools and technologies that comply with the GIS standards.There are relational database extensions that handle GIS features, such as PostGIS[4] or Oracle Spatial[5]. At higher levels, there are Java libraries to work with spatial data, such as the *Java Topology Suite* (JTS) or the library collection *Java GeoTools*. We can handle spatial data in JavaScript with *GeoJSON* and libraries such as *Turf*. The view layer is built with the help of tools such as *OpenLayers* or *Leaflet*.

The visualization of geospatial data involves three concepts: layers, styles, and maps. A *layer* is an image that can be geographically bounded. This image can be composed of a set of real photos, as in the case of a satellite view, or it can be generated from geographic data by applying a given *style*. When a *layer* is loaded in a map viewer, the viewer is

[4]`https://postgis.net/`

[5]`https://www.oracle.com/database/technologies/spatialandgraph.html`

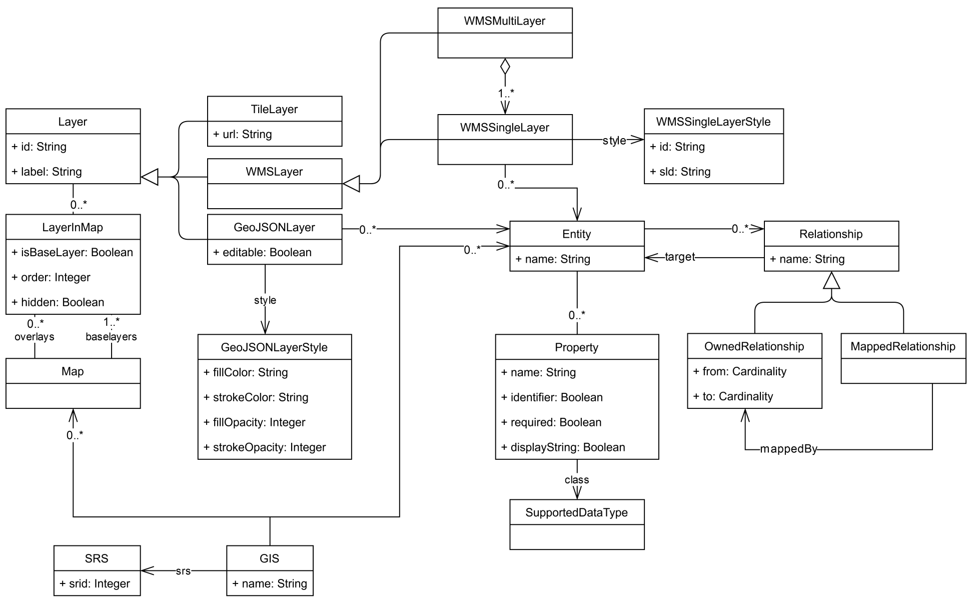


Figure 3.2: A metamodel of web-based GIS (reduced version, adapted from [ACL+19]).

responsible for asking the specific image needed depending on the bounds of the view, using a specification such as *TileLayer* or WMS. In some cases, the image is generated by the map viewer itself when we are dealing with raw data loaded with GeoJSON documents. The *styles* determine how the data behind a *layer* is transformed into images. Depending on the type of *layer*, we have different *style* specifications. For example, styles are usually not necessary for satellite images. If we are handling a WMS *layer* we need to use a *style layer descriptor* (SLD). A *map* is composed of a set of *layers* with their *styles* rendered in a particular order. Usually the *layers* are generated from data of the application itself, that is, from *entities* with spatial properties. For example, we can have a *layer* of the traffic lights of a city, or a *layer* that shows the roads of a region. The metamodel presented in fig. 3.2 formalizes these concepts and how they relate to each other.

fig. 3.3 shows our architecture for a web-based GIS. The server side provides two services for the clients: a REST service handles most of the alphanumeric data and can provide geographic data in a serializable format (such as GeoJSON), and a WMS that provides cartography images by using a map server. Both the data and cartography services are fed from the same database. In the client side, the data layer is the component in charge of handling the REST communication, the logic of the application is handled by JavaScript code, and the templates are created using HTML. There is also a map viewer library that is working as a closed component and that can handle direct communication with the WMS.

In the current implementation of GIS-DSL, we aim at generating GIS applications according to a specific architecture and a set of technologies. Therefore, we have not

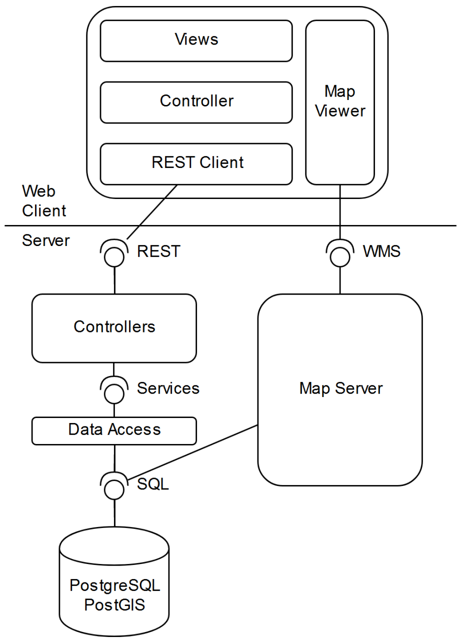


Figure 3.3: Typical architecture of a web-based GIS (reproduced from [ACL+19]).

followed the complete MDA architecture, since we transform the specifications of the DSL directly into code. However, the DSL would still be valid if we were interested into a MDA-based implementation, able to generate applications for different platforms.

3.2.2 GIS-DSL

GIS-DSL is a declarative language composed of sentences that allow programmers to specify the entities, maps, and layers they need, without needing to specify any detail on how to implement them or the control flow associated to their processing [Fow10, Seb16].

3.2.2.1 CREATE GIS and USE sentence

The specification of an application starts with the sentence `CREATE GIS`. We specify the name of the project, and the spatial reference system that will be used (each reference system is defined by a specific id, `srid`).

```
CREATE GIS name USING srid;
```

Listing 3.1: `CREATE` and `USE` sentences.

3.2.2.2 CREATE ENTITY sentence

The domain can be specified using the sentence `CREATE ENTITY` (see listing 3.2). Each entity has a name and a set of properties. A property is defined by its name and its data type, which can be any of the types shown in Figure 3.1. Each entity must have an identifier, defined by adding the keyword `IDENTIFIER` to the properties that compose it. Relationships between entities can be defined as well, indicating the name of the relationship, its cardinally, and its navigability.

```
CREATE ENTITY entityName (
    propertyName1 dataType1 [ IDENTIFIER ] [ REQUIRED ] [ DISPLAY_STRING ] [ UNIQUE ],
    propertyName2 dataType2 [ IDENTIFIER ] [ REQUIRED ] [ DISPLAY_STRING ] [ UNIQUE ],
    ...
    relationshipName1 entityName RELATIONSHIP{ (
        { 0..1 | 1..1 | 0..* | 1..* },
        { 0..1 | 1..1 | 0..* | 1..* }
    ) [ BIDIRECTIONAL ] | MAPPED_BY relationshipNameInTheOtherEntity },
    ...
);
```

Listing 3.2: `CREATE ENTITY` sentence.

3.2.2.3 CREATE LAYER sentence

Once the domain model of the system has been defined, we can define the layers available to be visualized in the map viewers of the application (see listing 3.3). The `CREATE LAYER` sentence allows us to create three different types of layers: Tile Layers, WMS Layers, and GeoJSON Layers. Tile layers are defined by an external URL, and they are used normally as base layers. GeoJSON layers are generated from an entity of the application, which is loaded into the map from the REST service applying a certain style. Entities loaded using this type of layers can be editable using the forms of the application. Finally, WMS layers are loaded as cartography through a map server, and they can be generated from one or several entities from the application.

```
CREATE TILE LAYER name [ AS label ] (
    url STRING
);

CREATE GEOJSON LAYER name [ AS label ] (
    entity [ EDITABLE ],
    fillColor HEX,
    strokeColor HEX,
    fillOpacity FLOAT,
    strokeOpacity FLOAT
);

CREATE WMS_STYLE name (
    styleLayerDescriptor FILE_PATH
);

CREATE WMS LAYER name [ AS label ] (
    entity1 WMS_STYLE,
    entity2 WMS_STYLE,
    ...
);
```

Listing 3.3: `CREATE LAYER` sentence

3.2.2.4 CREATE MAP sentence

The `CREATE MAP` sentence allows us to define the map viewers (see Listing 3.4). A map is composed of a set of layers, with one of them acting as the base layer. A layer can be defined as hidden by default in the view.

```
CREATE [ SORTABLE ] MAP name [ AS label ] (
    layer1 [ IS_BASE_LAYER ] [ HIDDEN ],
    layer2 [ IS_BASE_LAYER ] [ HIDDEN ],
    ...
);
```

Listing 3.4: `CREATE MAP` sentence.

3.2.2.5 GENERATE GIS sentence

The sentence `GENERATE GIS` transforms all the specifications made with previous sentences into the source code of a working system. The resulting GIS provides the users with forms and listings to create, edit, list, and remove any of the entities defined in the data model. The map viewer also includes all the layers, styles, and maps defined. The resulting system may be missing complex functionalities required by the users. It must be noticed that the purpose of the DSL is not to generate a complete system with arbitrarily complex functionalities but to generate a functional system that can be extended with more complex functions implemented in the general-purpose programming language.

```
GENERATE GIS name;
```

Listing 3.5: `GENERATE GIS` sentence.

3.2.3 Use Example

To illustrate the use of the DSL, we present examples with two basic GIS applications, Administrative Office and Point of Interest.

3.2.3.1 Administrative office

Local administrations usually need to manage a set of buildings with different functions, such as water distribution buildings (pipes, wells, tanks, chlorination stations, etc.), road networks (streets, municipal roads, bridges, etc.), cultural-related buildings (schools, sport halls, community centers, etc.), or administrative buildings. Most applications managing this kind of data use GIS technologies, allowing the users to visualize the information through map viewers and to digitize new elements.

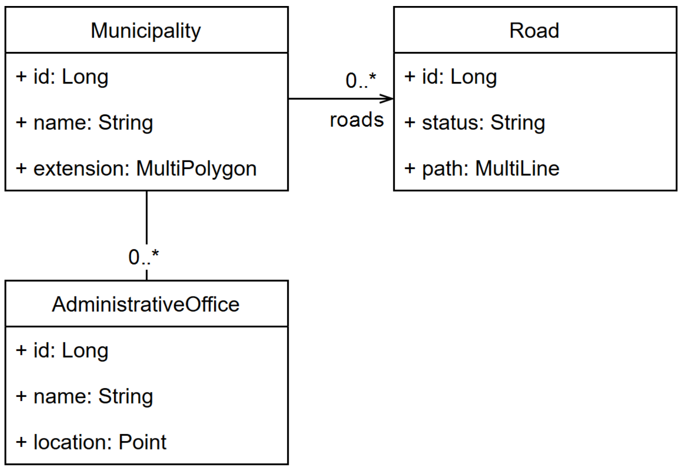


Figure 3.4: Data model of the example application (reproduced from [ACL+19]).

fig. 3.4 shows a data model that specifies that we manage municipalities, roads, and administrative offices, each one with its geographic component (a multi-polygon for municipalities, a multi-line for roads, and a point for administrative offices) and with their relationships. listing 5.22 shows the GIS-DSL code that specifies the web-based GIS application that manages that application model. First, a new GIS is created. We use the SRID EPSG:25829, which is a local reference system commonly used when working in the north-west of Spain.

Next, we define the application model, creating its three entities. The names of the entities will be used afterward for defining the different layers that will be provided by the application. These layers are also linked to the only map viewer we define, in which it will appear a TileLayer from Open Street Maps (OSM) that works as the base layer, a WMS Layer that combines both the municipalities and the roads, and a GeoJSON Layer with the administration offices. The latter also allows accessing a form directly from the map viewer so the offices can be edited. Finally, the GIS application is generated, with forms, listings, and maps to manage the entities.

```
CREATE GIS local_administration_manager USING
     25829;

CREATE ENTITY Road (
  id Long IDENTIFIER DISPLAY_STRING,
  status String,
  path MultiLineString
);

CREATE ENTITY Municipality (
  id Long IDENTIFIER,
  name String REQUIRED DISPLAY_STRING,
  extension MultiPolygon,
  roads Road RELATIONSHIP(1..1, 0..*),
  offices AdministrativeOffice
     RELATIONSHIP(1..1, 0..*) BIDIRECTIONAL
);

CREATE ENTITY AdministrativeOffice (
  id Long IDENTIFIER DISPLAY_STRING,
  status String,
  location Point,
  municipality Municipality RELATIONSHIP
      MAPPED_BY offices
);

CREATE TILE LAYER base AS "Base Layer" (
  url "https://{s}.tile.osm.org/
    {z}/{x}/{y}.png"
);


CREATE GEOJSON LAYER offices AS
     "Administrative Offices" (
  AdministrativeOffice EDITABLE,
  fillColor #243452,
  strokeColor #eeeee3,
  fillOpacity 0.8,
  strokeOpacity 0.9
);

CREATE WMS STYLE BasePolygonStyle (
  styleLayerDescriptor
     "/home/user/sld/file_polygon_sld.xml"
);

CREATE WMS STYLE BaseLineStyle (
  styleLayerDescriptor
     "/home/user/sld/file_line_sld.xml"
);

CREATE WMS LAYER defaultOverlay AS "Overlay" (
  Municipality BasePolygonStyle,
  Road BaseLineStyle
);

CREATE SORTABLE MAP theMap AS "Map Viewer" (
  base IS_BASE_LAYER,
  defaultOverlay,
  offices HIDDEN
);

GENERATE GIS local_administration_manager;
```

Listing 3.6: Example of application Administrative office defined by the DSL.

3.2.3.2 Point of interest

Points of interest is a mobile application that allows you to register places of interest for the user and track their entry/exit points. From this information, the application records the routes through which the user has traveled and generates a visit plan. Our model is represented by four entities: *City*, *Streets*,*Points of interest*, and *Point type*. Each *City* has a collection of *Streets*, defined spatially by a multi-line. In each *City* the system will allow storing *Points of interest* that belong to a given *Point type* and are spatially defined by a point. fig. 3.8 shows the simplified class diagram of this application. In the listing 3.7 is shown an example of the application point of interest defined by the DSL proposed.

```
CREATE GIS point_of_interest USING 25829;

CREATE ENTITY City (
  id Long IDENTIFIER,
    name String REQUIRED DISPLAY_STRING,
    extension MultiPolygon,
    street Street RELATIONSHIP(1..1, 0..*)
     BIDIRECTIONAL,
    pinterest PointOfInterest
     RELATIONSHIP(1..1, 0..*) BIDIRECTIONAL
);

CREATE ENTITY Street (
  id Long IDENTIFIER,
  name String REQUIRED DISPLAY_STRING,
    type String,
    path MultiLineString,
    city City RELATIONSHIP MAPPED_BY street
);

CREATE ENTITY PointOfInterest (
  id Long IDENTIFIER,
  name String REQUIRED DISPLAY_STRING,
    location Point,
    city City RELATIONSHIP MAPPED_BY pinterest,
    poiType PoiType RELATIONSHIP MAPPED_BY
     pinterest
);
```

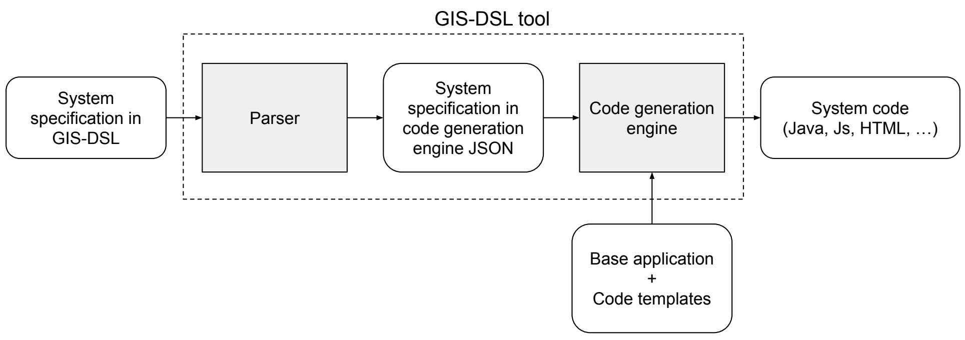

GIS-DSL tool
System specification in GIS-DSL
Parser
System specification in code generation engine JSON
Code generation engine
System code (Java, Js, HTML, ...)
Base application + Code templates

Figure 3.5: Architecture of the GIS-DSL code generation engine.

```
CREATE ENTITY PoiType (
  id Long IDENTIFIER,
  name String REQUIRED DISPLAY_STRING,
    pinterest PointOfInterest
     RELATIONSHIP(1..1, 0..*)
);

CREATE TILE LAYER base AS "Base Layer" (
  url "https://{s}.tile.osm.org/
    {z}/{x}/{y}.png"
);

CREATE GEOJSON LAYER pointOfInterest AS "My
     PointOfInterest" (
  PointOfInterest EDITABLE,
  fillColor #243452,
  strokeColor #eeeee3,
  fillOpacity 0.8,
  strokeOpacity 0.9
);

CREATE WMS STYLE BasePolygonStyle (
  styleLayerDescriptor
     "/home/user/sld/file_polygon_sld.xml"
);

CREATE WMS STYLE BaseLineStyle (
  styleLayerDescriptor
     "/home/user/sld/file_line_sld.xml"
);

CREATE WMS LAYER defaultOverlay AS "Overlay" (
  City BasePolygonStyle,
  Street BaseLineStyle
);

CREATE SORTABLE MAP theMap AS "Map Viewer" (
  base IS_BASE_LAYER,
  defaultOverlay,
  pointOfInterest HIDDEN
);

GENERATE GIS point_of_interest;
```

Listing 3.7: Example of application Point of interest defined by the DSL.

3.3 Implementation of the DSL

The implementation of GIS-DSL is based on the use of the *generation engine* presented in [CLPP17]. This engine receives an input consisting of a specification of classes, relationships, and other auxiliary elements, which is processed and combined with a set of templates to generate source code applying *scaffolding* technologies.

Figure 3.5 presents a diagram with the architecture of our tool.The input is a file containing the specification of the GIS application using the DSL. This specification is first processed by a parser, which reads all its elements. and transforms it into an intermediate specification in JSON, which is the input for the generation engine. The intermediate specification is then processed by the code generation engine and combined with a set of templates to generate the source code of the final system, which includes files in different programming languages, such as Java, JavaScript, and HTML. The code templates are part of a *base application* that also contains other source code that will form part of any application generated by our tool.

3.3.1 GIS-DSL Parser

The parser was implemented with ANTLR[6]. Besides producing the parser that can read, validate, and process the language, it provides easy mechanisms to run native code as the grammar rules are processed.

listing 3.8 shows the GIS-DSL grammar, omitting the definition of the lexer. Every token that finishes with the suffix `_SYMBOL` corresponds to the text that comes before the *underscore*. For example, `SORTABLE_SYMBOL` is the word `SORTABLE`, and `MAP_SYMBOL` is the word `MAP`. The symbols `OPAR`, `CPAR` and `SCOL` are "`(`", "`)`", and "`;`", respectively.

```
parse: sentence+;

sentence: createStatement | useGIS |
     generateGIS;

createStatement:
  CREATE_SYMBOL (
    createGIS | createEntity | createLayer
  )
;

createGIS:
  GIS_SYMBOL identifier USING_SYMBOL srid
     SCOL_SYMBOL
;

createEntity:
  ENTITY_SYMBOL identifier OPAR_SYMBOL
    property (COMMA_SYMBOL property)*
  CPAR_SYMBOL SCOL_SYMBOL
;

createLayer: createTileLayer |
     createGeoJSONLayer | createWmsStyle |
     createWmsLayer | createMap |
     createSortableMap;

createTileLayer:
  TILE_SYMBOL LAYER_SYMBOL identifier
     (AS_SYMBOL text)? OPAR_SYMBOL
    URL_SYMBOL text
  CPAR_SYMBOL SCOL_SYMBOL;
```

```
createGeoJSONLayer:
  GEOJSON_SYMBOL LAYER_SYMBOL identifier
     (AS_SYMBOL text)? OPAR_SYMBOL
    identifier (EDITABLE_SYMBOL)? COMMA_SYMBOL
    FILL_COLOR_SYMBOL hexColor COMMA_SYMBOL
    STROKE_COLOR_SYMBOL hexColor COMMA_SYMBOL
    FILL_OPACITY_SYMBOL floatNumber
     COMMA_SYMBOL
    STROKE_OPACITY_SYMBOL floatNumber
  CPAR_SYMBOL SCOL_SYMBOL;

createWmsStyle:
  WMS_SYMBOL STYLE_SYMBOL identifier
     OPAR_SYMBOL
    SLD_SYMBOL text
  CPAR_SYMBOL SCOL_SYMBOL;

createWmsLayer:
  WMS_SYMBOL LAYER_SYMBOL identifier
     (AS_SYMBOL text)? OPAR_SYMBOL
    wmsSubLayer (COMMA_SYMBOL wmsSubLayer)*
  CPAR_SYMBOL SCOL_SYMBOL;

wmsSubLayer: identifier identifier;

createSortableMap: SORTABLE_SYMBOL createMap;

useGIS: USE_SYMBOL GIS_SYMBOL identifier
     SCOL_SYMBOL;

createMap:
  MAP_SYMBOL identifier (AS_SYMBOL text)?
```

[6]ANTLR: https://www.antlr.org/

```
     OPAR_SYMBOL
    mapLayer (COMMA_SYMBOL mapLayer)*
  CPAR_SYMBOL SCOL_SYMBOL;

mapLayer: identifier (IS_BASE_LAYER_SYMBOL)?
     (HIDDEN_SYMBOL)?;


generateGIS: GENERATE_SYMBOL GIS_SYMBOL
     identifier SCOL_SYMBOL;

property: propertyDefinition |
     relationshipDefinition;

propertyDefinition:
  identifier TYPE (
    IDENTIFIER_SYMBOL
    | DISPLAYSTRING_SYMBOL
    | REQUIRED_SYMBOL
    | UNIQUE_SYMBOL
  )*
;

relationshipDefinition:
  ownedRelationshipDefinition |
     mappedRelationshipDefinition;

mappedRelationshipDefinition:
  identifier identifier RELATIONSHIP_SYMBOL
     MAPPEDBY_SYMBOL identifier;

ownedRelationshipDefinition:
  identifier identifier RELATIONSHIP_SYMBOL
     OPAR_SYMBOL cardinality COMMA_SYMBOL
     cardinality
    CPAR_SYMBOL BIDIRECTIONAL_SYMBOL?;

cardinality:
  ZERO_ONE_SYMBOL
  | ONE_ONE_SYMBOL
  | ZERO_MANY_SYMBOL
  | ONE_MANY_SYMBOL
;

srid: INT_NUMBER;
identifier: IDENTIFIER;
text: QUOTED_TEXT;

hexColor: HEX_COLOR;
floatNumber: FLOAT_NUMBER;
```

Listing 3.8: GIS-DSL grammar.

As an example, Listing 3.9 shows the specification of an entity *AdministrativeOffice*. The parser transforms this specification into an intermediate JSON file, shown in listing 3.10.

```
CREATE ENTITY AdministrativeOffice (
  id Long IDENTIFIER DISPLAY_STRING,
  status String,
  location Point,
  municipality Municipality RELATIONSHIP MAPPED_BY offices
);
```

Listing 3.9: Specification of *AdministrativeOffice* in GIS-DSL.

```
{
  "name": "AdministrativeOffice",
  "properties": [{
      "name": "id",
      "class": "Long (autoinc)",
      "pk": true,
      "required": true,
      "unique": true
    },{
      "name": "status", "class": "String"
    },{
      "name": "location", "class": "Point"
    },{
      "name": "municipality",
      "class": "Municipality",
      "owner": true,
      "bidirectional": "offices",
      "multiple": false,
      "required": true
    }],
  "displayString": "$id"
}
```

Listing 3.10: Specification of *AdministrativeOffice* in JSON.

3.3.2 Code Generation Engine

The generation engine[7] used in our tool has been previously designed and developed to generate code from a system definition using a set of annotated code templates [CLPP17].

Based on [CLPP17] we User interface specification DSL use the *Code Generation Engine* to handle different components that allow define an API to generate the different software products. These components, as show the Figure 3.6 are the *file manager*, *template engine* and the *assets handler*.

- The *assets handler* allow validate the features that has been selected by the analyst. To do that, we need load the different assets throw the features model. This feature model is generated from annotations that previously has been created from our DSL.
- The *file manager* allow handle all related to access to the templates to generate source code. For example, template annotations are defined as comments of the programming language in which the template is written and its content can be any JavaScript code. These annotations allow you to select or exclude a set of features that are desirable for the system. Subsequently, these characteristics through a generation engine are translated into implemented source code and assembled into a final product.
- The *template engine* has been designed to simultaneously support concepts from *software product lines engineering* (SPLE) and *model-driven engineering* (MDE). From SPLE, it supports *feature models* to manage the *variability* of the products that can be generated. A *feature model*, briefly explained, is a way to organize and describe the characteristics or functionalities that appear in a family of products [KCH+90]. There are a set of operations related to *feature models* [BSRC10a], mostly used to determine if a particular configuration for a new product of the family is valid. From MDE, our generation engine uses *scaffolding* techniques to transform models and text into text, that is, system specifications and annotated code templates are transformed into the final source code of the system and the product assembles.

[7]spl-js-engine: `https://github.com/AlexCortinas/spl-js-engine`

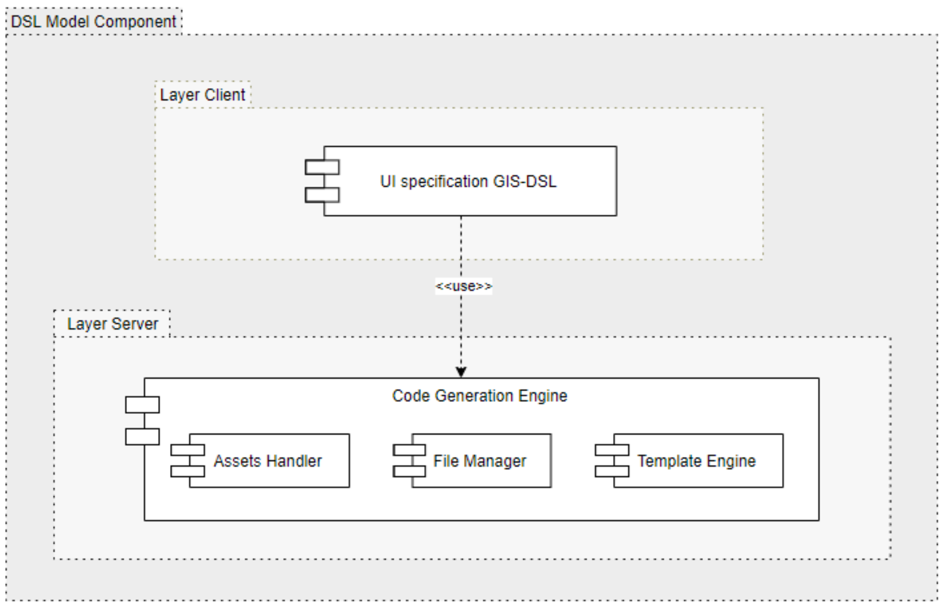


Figure 3.6: DSL Model Component.

3.3.3 Generated Code

The fig. 3.7 shows the structure of the low-level software elements generated by our tool for each element in the input specification, and the relationships between these components. The components belong to each one of the layers of our architecture, and they are implemented in different languages: Java, JavaScript, JSON, and HTML. On the server side, the *data persistence* is implemented with PostGIS and JPA (Hibernate specifically), so for each entity, we generate a class representing the entity, `Entity`, and a DAO, `EntityRepository`. Besides that, the data is provided by a *REST service*, so we generate a *RESTController* for each entity, `EntityResource`. On the client side, we can differentiate two types of visualization: lists and forms. For the former, we generate a component, `entityList`, with its controller and router definition (JS), and with the view (HTML). For the latter, we need two different components: one for the detail view, `entityForm-detail`, and one for the edition view, `entityForm-update`, each one of them with the controller (JS), router definition (JS) and the view (HTML). The rest of the files generated for each entity on the client side handle the communication with the REST service, `entity.resource`, and the message internationalization files, in JSON format.

On the other hand, each map defined in the specification only generates one file, a JSON that indicates the configuration of the different layers of the map. The component that renders the map takes this file and dynamically generates every layer, applies the required styles and configures the different options of the map, such as allowing to sort the layers.

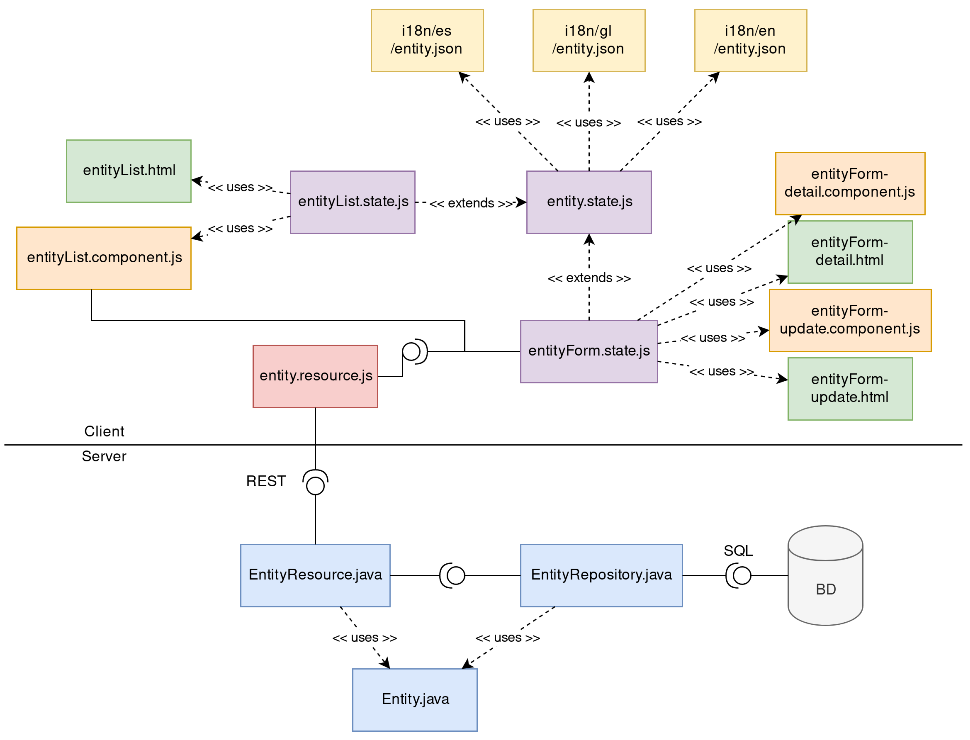


Figure 3.7: Low-level components of the generated source code.

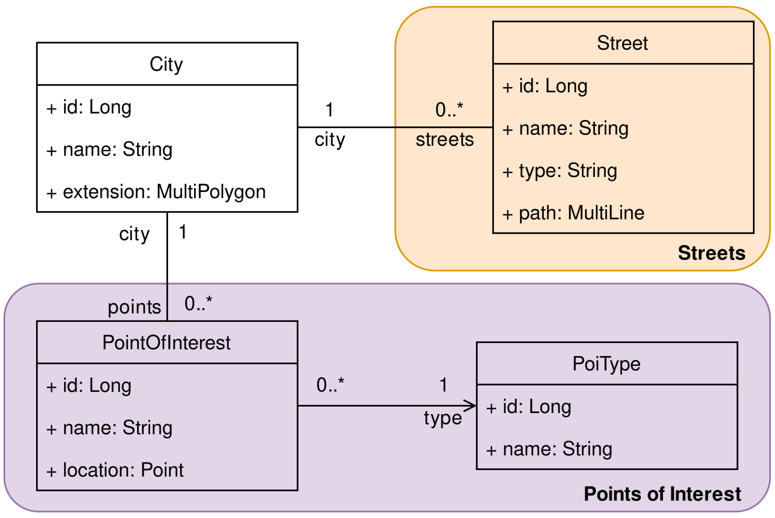


Figure 3.8: Class diagram of the first example project.

3.4 Case Study and Evaluation

In this section, we present a case study on the use of GIS-DSL that allowed us to evaluate the language and its implementation, focusing on the size and characteristics of the generated products, which have a direct impact on the development effort. We used two sample projects of different sizes. Each sample project is characterized by its number of entities, properties, relationships, maps, and layers since these are the elements that define the size of the system and the ones we can specify with the DSL. For both of them, we analyzed the resulting GIS application in terms of the total number of source code files, and the number of generated lines of code.

In the rest of this section, we describe the sample projects and their models. We then compare the results obtained in the generated products and discuss the findings in these results, and the limitations of the case study.

3.4.1 Sample Project 1: Points of Interest

Figure 3.8 shows a class diagram that defines the first sample project. In this case we are defining a simple application with just four entity types that will allow storing data of cities, defined spatially by a multi-polygon that defines the city boundaries. Each *City* has a collection of *Streets*, defined spatially by a multi-line. Also, the system will allow storing *Points of interest* that belong to a given *Point type* and are spatially defined by a point.

The two colors in the diagram identify the two maps that we want to define. The first one (orange background) will show the streets in the cities using a WMS layer, applying a style that depends on the *type* property of each street. The second map (purple background) will show the points of interest of the city in a GeoJSON layer. Using a GeoJSON layer is useful if we want to facilitate the edition of the elements of the layer since the users can access the edition form from a popup that shows clicking on an element of the map.

3.4.2 Sample Project 2: Local Civil Infrastructure Management

A typical problem for many public administrations is civil infrastructure management. The list of elements to manage is large but, to keep the example within a reasonable scope, we decided to focus on five areas: urban planning, road management, population information, medical facilities, and water supply facilities. Figure 3.9 shows a class diagram that defines the second sample project (as in the previous example, the background colors indicate the entities that will be shown together in a map):

- *Urban planning* (orange background): local administrations in Spain must define an urban plan that structures the territory of the municipality into areas with different construction permissions. In this way, the administration establishes the types of buildings that can be built in each zone. This information is kept in the system with the entities *UrbanPlan* (since a municipality can define many urban plans over time) and *UrbanPlanZone*, defined spatially with a multi-polygon.
- *Road management* (purple background): our system will store the information of the *Roads* and their *Road sections*. Each road section is defined spatially by its *route* (a multi-line string) and its surface (a multi-polygon), which can be also of interest in many cases.
- *Population information* (yellow background): demography is an important aspect of municipal administration. The *PopulationEntity* class represents the partition of each municipality into lower-level entities with their boundaries represented with a multi-polygon and storing the population that lives in the entity without living in a population settlement (i.e., outside cities, towns or villages). The *PopulationSettlement* class represents the settlements withing each population entity using a multi-polygon for the boundary and storing the number of inhabitants.
- *Medical and social facilities* (green background): our system will allow storing information of medical and social buildings, such as *Hospitals*, *Medical centers* (both defined spatially by a point), and *Social centers* (in this case, defined by two multi-polygons defining the area of the building and the surrounding parcel respectively).
- *Water supply network* (red background): the water supply network is defined by a set of nodes, that typically include water *Collection* and *Storage* locations, *Purification* and *Pumping* premises, and a set of edges, the *SupplyPipes*. The nodes are defined spatially by points, and the pipes are defined by multi-line strings.

In this example we defined 15 layers, where 14 of them correspond to the entities with an spatial component (that is, all entities except *UrbanPlan*), and an additional layer used to include the roads in the rest of the maps. In fig. 3.10 we show a capture of the application on the map “Urban planning”, which includes a base layer from OpenStreetMap and 4 layers from our data (entities *Municipality*, *Building*, *RoadSection* and *UrbanPlanZone*).

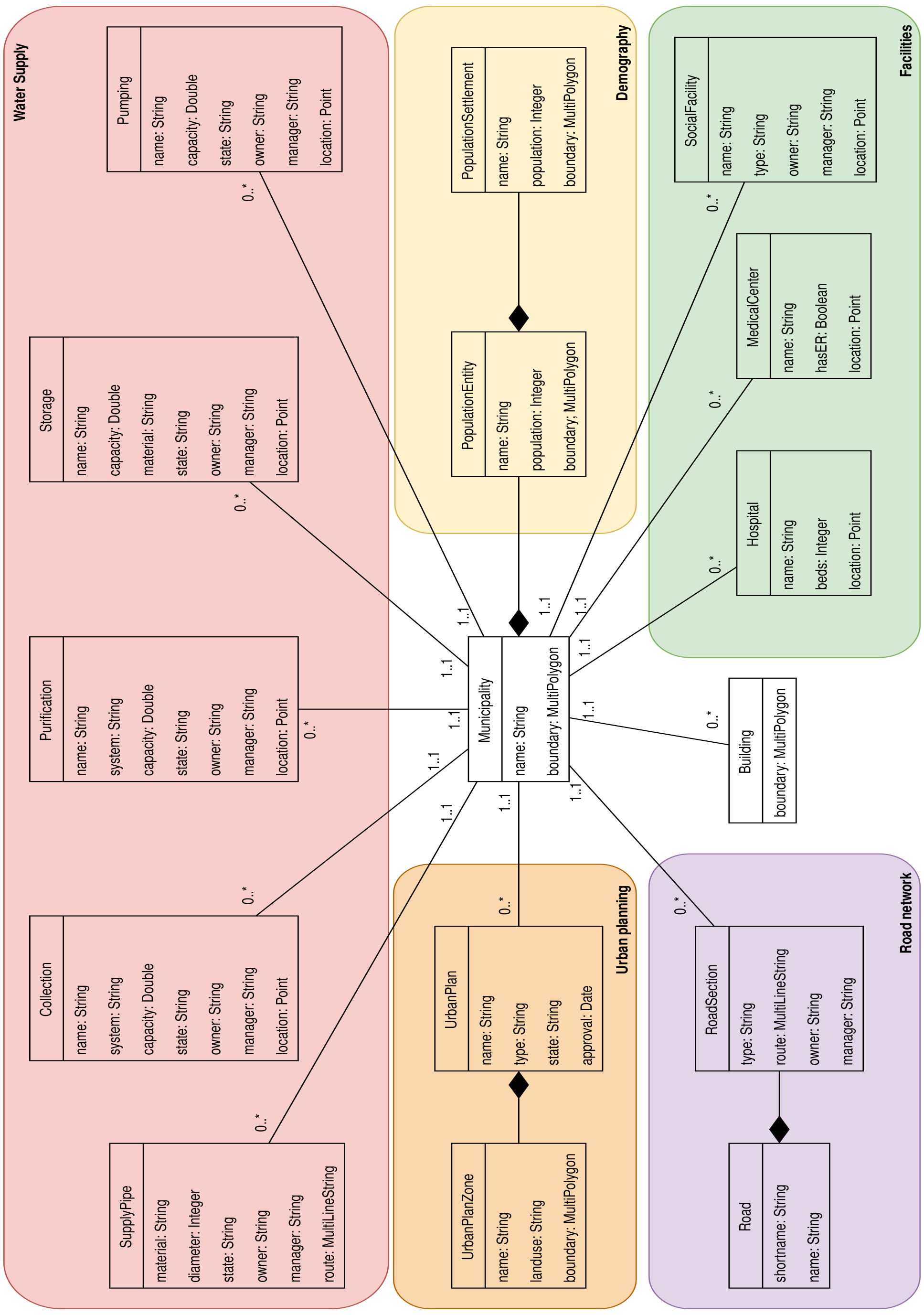


Figure 3.9: Class diagram of the second example project.

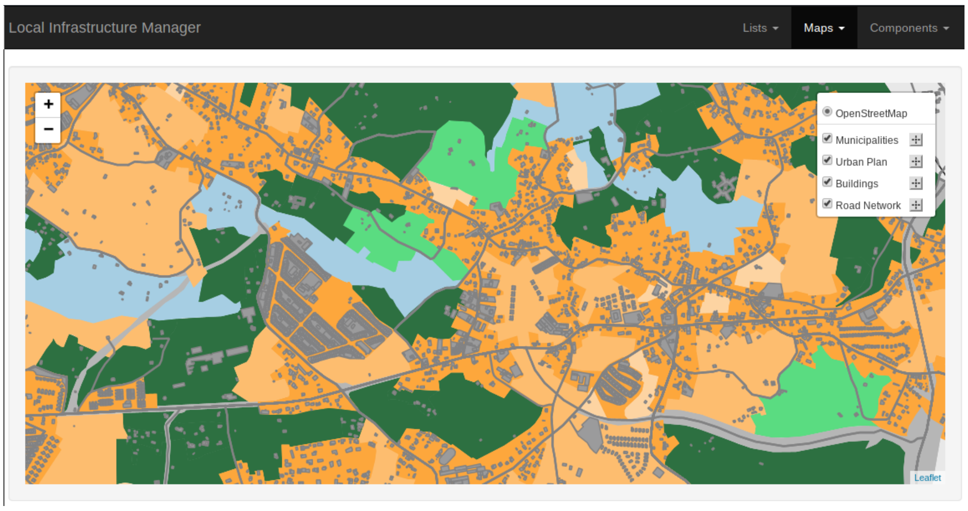


Figure 3.10: Example of a map defined for the product described in section 3.4.2.

3.4.3 Results

In this section, we present the results of the analysis of the results obtained in the two sample projects. For each project, we wrote the system specification in GIS-DSL and generated the applications. In the previous section, we explained that the software is generated from a set of code templates and a base application, that comprises 175 files (Java, HTML, and JavaScript), and 13,569 lines of code. For both projects, we measured the number of source code files and the number of lines of code (LOC). In the case of the lines of code, we distinguished between the total number of lines and the newly generated lines of code. We also distinguished between the number of LOC in the back-end and the front-end.

Table 3.1: Characteristics of each example project.

Project	Entities	Properties	Relationships	Maps	Layers
Project 1	4	12	3	2	4
Project 2	16	82	15	5	15

Table 3.1 shows the characteristics of the two sample projects in terms of the number of entities, properties, relationships, maps, and layers. Table 3.2 shows the size of the resulting generated projects, in terms of the number of files and the number of lines of code (LOC).

As we can see in the tables, the differences between the two sample projects allow us

Table 3.2: Characteristics of the generated software for each example project.

Project	Files	Lines of code (LOC)			
		Total	Generated	Back-end	Front-end
Project 1	78	17,673	4,104	10,113	7,560
Project 2	273	29,573	16,004	13,988	15,585

to analyze the results of the code generation process in different settings. In both cases, we can see that the number of LOC is similar in the back-end. However, the number of LOC in the front-end is much higher in project 2 because it contains more maps and layers. In the case of project 1, the number of generated LOC is small compared to the number of lines of the base application. However, in project 2, with just 16 entities, the number of generated LOC is considerably higher. We can see also that the number of generated LOC per entity is almost the same in both projects, 1,026 in project 1, and around 1,000 in project 2.

A potential limitation of this case study is that its extent does not allow us to conclude that this ratio would be the same in any other project since the number of generated lines of code is also influenced by the number of properties, relationships, maps, and layers. However, it gives us an idea of the code generation ratio we can achieve, and of the savings we could obtain in larger projects, with tens or hundreds of entities. In addition, these data could be combined with the average cost per hour at a given company to compute the savings directly in economic terms. Therefore, these data give us an insight on the improvements in development productivity.

Other aspects that can be analyzed on the generated code are its quality and maintainability. The code generated by our implementation follows a clear architecture and a set of design patters, without any possible deviation from those prescriptions because is generated automatically.

A more extensive experimental evaluation remains as future work. Using the DSL to generate existing real applications would imply additional coding on top of the generated applications. That would allow us to evaluate the savings and productivity improvements more accurately and, also, it would allow us to analyze the break-even point. In addition, it would also allow us to better evaluate the quality of the generated code compared to that of custom-developed applications and its maintainability, according to models based on the family of norms ISO 25.000, such as [RP14], and even the satisfaction level of programmers using the DSL and users of the resulting software.

3.5 Conclusions

A GIS manages entities with a spatial component that plays a central role in the system and its functionalities. Most GIS share a common set of concepts, architecture, design patterns, and technologies, so their development is quite similar in many aspects. Therefore, we consider that GIS is a suitable domain for applying model-driven engineering techniques.

In this chapter, we have proposed GIS-DSL, a declarative domain-specific language for the development of GIS. This language allows the developer to specify the system by defining its entities, with their properties and relationships, maps, and layers. We have implemented the DSL in a tool that generates the source code of the system from the specification in GIS-DSL. Although the current implementation generates applications written in Java, HTML, and JavaScript, it could easily be modified to generate applications in another programming languages and platforms, just replacing the base application and the code templates.

As it happens in many applications of MDE, the purpose of GIS-DSL and the tool that implements the language is not being able to implement every functionality of any specific GIS, but to implement the basic management functions on the defined entities, and the visualization based on the layers and maps defined by the developer. Therefore, in most cases, the resulting software product will have to be completed to match all the functional requirements of the domain.

We have presented a case study in which we used two sample projects to evaluate the software products generated from the specifications in GIS-DSL. In particular, the first sample project is a simple application for managing points of interest (with four entities), and the second sample project is an application for managing civil infrastructures (roads, water supply networks, urban planning, population information, and medical and social facilities). These sample projects are of different sizes, which allowed us to analyze the lines of code generated in different scenarios, also considering the number of lines of code that form the base application. A limitation of the case study is the choice of technologies, architecture, and implementation decisions we have made, which could be different in other settings. However, it allowed us to conclude that the number of lines of source code we can generate is high if compared with the size of the specification of the systems. As we explained in the previous section, a more extensive evaluation, involving the generation of real applications, remains as future work.

Chapter 4

Multilevel Modeling of Geographic Information Systems

Geographic Information Systems (GIS) support the processes of capturing, managing, visualizing, and analyzing data with a geospatial component [WD04]. GIS are used in many application domains, such as the management of transportation networks, logistics, supply infrastructures, or territory administration, among many others. Although in their beginnings GIS were used mainly by public administrations and engineering companies, the evolution of technologies for GIS development and the appearance of cheap mobile devices with GPS capabilities has extended the use of GIS to companies in very different domains.

Despite the differences in their functional scope, most GIS applications share a common set of concepts, standards, architecture, components, and technologies. For example, all GIS deal with spatial data types (such as points, lines, polygons, or variants of these basic types), coordinate systems, maps, layers (used to organize the information shown in maps), and operations to process spatial data. All these elements are defined in different standards from the ISO committee on geographic information/geomatics (ISO/TC 211[1]) and the Open Geospatial Consortium (OGC[2]), hence existing technologies for GIS development support them in the same (or very similar) way. Therefore, two different GIS are modeled and developed similarly, even if they have a different purpose and functional requirements. Also, due to the nature of the information managed in these application domains, some common structures appear repeatedly, such as, for example, network structures (e.g., in domains such as road networks, telecommunications, or energy supply), or hierarchical decomposition of entities at different spatial levels (e.g., territory administration or facility management).

In model-driven engineering (MDE), models play a central and active role in the

[1]`https://committee.iso.org/home/tc211`

[2]`http://www.opengeospatial.org/`

software development process, far beyond just describing the system. Models describe the software system, and they are artifacts that can be processed to be successively and automatically transformed into models at lower levels of abstraction, and, finally, into the source code of the system [BCW17c, PM07b]. In MDE, a *metamodel* describes the elements that can be used in the models that conform to that metamodel. The traditional approach to MDE considers a fixed number of metamodeling levels. Typically, objects are described by models that define their state, behavior, and relations, and these models use concepts defined in a metamodel at a higher level of abstraction. At the same time, at the metamodel level, we can use elements defined in a meta-metamodel defined at a higher level. A common approach to MDE is based on the OMG's[3] *model-driven architecture* (MDA)[4], which defines four layers for software modeling: computational independent models, platform-independent models, platform-specific models, and system code. The OMG also defined a standard for *meta-object facility* (MOF), that defines the way to create *domain-specific modeling languages* (DSML), usually, through meta-modeling based on two levels of abstraction.

Working in one metamodel level implies balancing the scope of the metamodel and the flexibility of the solution. On the one hand, a simple metamodel would only define basic elements that could be potentially used in any other model. This solution would be simple but would force us to repeat the same information structures in different systems. On the other hand, creating a complex and rich metamodel that tries to define those information structures may be too rigid since it would be difficult to adapt to the particularities of a specific model. In [CLP$^+$17, CLPP17, ACL$^+$20], for example, we opted for creating a simple metamodel with just the basic elements common to any GIS (basic entities with a spatial component, layers, and maps). The resulting metamodel is very flexible, but it forces the designer to repeat many elements in different models.

A recent trend in MDE is *multilevel modeling* [dLGC14, AK01, Atk97]. The idea of multilevel modeling is that the number of metamodeling levels is not fixed, so the designer can use the number of levels that better fit a particular domain. This approach aims at simplifying the complexity of the models through the separation of specific domain concepts that can be modeled at different levels. Multilevel modeling solves some drawbacks and restrictions that can occur in the traditional two-level modeling, which forces the description of the application domain in one level, something that can lead to an unnecessary complexity [dLGC14]. Despite the attention multilevel modeling is attracting, few works focus on its application to real scenarios. This has already been pointed out by de Lara et al., that mention in [dLGC14] that "there are scarce applications of multilevel modeling in realistic scenarios", and by Frank who also pointed out in [Fra16] that "only little attention has been paid to applying multilevel modeling to particular domains".

In this chapter, we address the modeling of geographic information systems with a multilevel approach. Our main goal was to determine if a solution for GIS based on multilevel modeling could solve the drawbacks of a two-level solution, considering the following requirements: (R1) *Scope:* the set of models should be rich and include the typical elements in most GIS models; (R2) *Reuse of common structures:* the set of models must support the definition and reuse of common structures appearing in many GIS application domains; (R3) *Flexibility:* the solution must allow the designer to adapt high-level designs

[3]Object Management Group: `http://www.omg.org`

[4]Model Driven Architecture: `http://www.omg.org/mda`

to the particularities of an application, and be easily extensible to incorporate new elements; (R4) *Realistic:* the solution must consider (not necessarily in an exhaustive way) scenarios extracted from existing proposals for the modeling of GIS; and (R5) *Generality:* the solution must not be limited to one particular case.

We present a proposal with a set of models that are based on international standards for geographic information systems. The first scenario applies multilevel modeling to the conceptual standards defined by ISO TC/211 and the implementation standards defined by the OGC to bridge the gap between these two sets of standards. The following three scenarios were extracted from the data specifications of the European Union INSPIRE Directive, which defines a set of models for information regarding resource and environmental management so that EU member countries can follow them to ensure interoperability. We have selected the application domains of territory administration, spatial networks, and facilities management. The four scenarios show the advantages that multilevel modeling may bring when compared with a two-level approach.

The motivation and contribution of this theses are twofold. First, the models of the INSPIRE Directive of the UE provide a general design of GIS for different purposes in the public administration. That approach is based on well-known international standards and tries to provide a general solution to different GIS problems that may fit the needs of administrations and companies in different countries of the Union. However, that solution is based on deep and complex hierarchies of classes, which, based on our experience, should be extended even more to adapt them to the particularities of each country. In this research, we present an alternative solution using multilevel modeling and show that it is more flexible and easily adaptable to those particularities than a traditional solution based on two levels. Second, we believe the solution we present also provides a real application scenario for an emerging modeling approach as multilevel modeling, that allows us to compare it with an existing design based on a traditional two-level approach, and that can contribute to understanding the advantages of this modeling technique in a real scenario.

The rest of the chapter is structured as follows:

In section 4.1 we present background and related work, including a brief description of the main elements of a GIS, and a summary of previous work on applying a two-level MDE approach for their development. In section 4.2 we present our proposal for developing GIS under a multilevel modeling approach. We describe four scenarios with problems that appear in real-world GIS applications, we then present example metamodels based on international standards for each scenario, and we describe the advantages of using multilevel modeling.

Section 4.3 presents a discussion and evaluation of the solution presented in section 4.2. Finally, section 4.4 presents the conclusions of the paper, the work we are currently undertaking, and lines for future work.

4.1 Background and Related Work

In this section, we review existing works on the application of multilevel modeling. Finally, we present previous applications of MDE and software product lines engineering (SPLE) to the GIS domain.

4.1.1 Multilevel modeling and its applications

The traditional approach to MDE considers two metamodeling levels. In the metamodel level, the designer defines the main concepts of the domain. At a lower level, a domain-specific modeling language can be defined from the metamodel to allow the designer to create models of the system. A promising trend within MDE is that of *multilevel software modeling* [AK01, AK03, AK08]. In contrast to a more “traditional” approach, multilevel modeling does not fix the number of metamodeling levels, so the designer could use the number of levels that better fit a particular domain. This approach aims at simplifying the complexity of the models through the separation of specific domain concepts that can be modeled at several levels. Multilevel modeling solves some drawbacks and restrictions that can occur in the traditional two-level modeling [dLGC14]. As explained in [Fra18], many modeling languages for this purpose have been proposed and, although they are different in some elements, they all share common features, such as considering that all classes at any level are also objects, and allowing for deferred instantiation of attributes.

Few works have presented applications of multilevel modeling in real scenarios: for example, Al-Hilank et al. [AHJK$^+$14] applied multilevel modeling in the context of development process improvement in the automotive industry to model the mappings between the concepts that describe the software development process and different quality standards. In Al-Hilank’s work, multilevel modeling allowed to model the relations between domain concepts at different levels of abstraction. Frank [Fra16] applied multilevel modeling in the development of systems and models to support IT management, so the concepts of the IT domain could be refined in successive levels. Similarly, Benner [Ben17] applied it to model-based development of user interfaces. Benner’s proposal allows to model elements of user interfaces in different levels of abstraction without using deep inheritance hierarchies. Nesic and Nyberg [NN17] applied multilevel modeling to data integration in the context of software product lines. In [RRD$^+$18], multilevel modeling was applied by Rodriguez et al. to the modeling of colored Petri nets. In [RDIR19], Rossi et al. presented a multilevel modeling solution to the modeling of IoT applications for the detection of tourism flows, so different aspects and concerns of the system’s architecture can be modeled at different levels. All these works share a common motivation: the need to represent abstract concepts in more than two metamodeling levels. Since the area of multilevel modeling is relatively new, some of these works have also mentioned issues such as the lack of tools.

de Lara et al. present in [dLGC14] a research work focused on “When and how to use multilevel modelling”. The authors mention that “unfortunately, there are scarce applications of multilevel modeling in realistic scenarios [. . .]”. After analyzing a large set of metamodels from different sources, they identified many domains in which a multilevel approach could be more beneficial than a two-level approach, and they also identified a set of patterns where multilevel modeling may bring advantages. Frank also pointed out in [Fra16] that “only little attention has been paid to applying multilevel modeling to particular domains”.

4.1.2 Applications of MDE to GIS

Although most GIS applications are based on a common set of standards, technologies, and assets, they are usually developed “from scratch” with the help of these tools and libraries. This results in low productivity, long time-to-market projects, and high costs, especially in the maintenance and evolution stages. However, it remains clear due to all the exposed above that GIS is a more than adequate field to apply techniques of semi-automatic software development.

Some previous works have applied elements of MDE to GIS development. Regarding modeling and development, Lisboa-Filho et al. [LFSNdVB10, SNF10] proposed a UML profile to support GIS-related concepts in UML conceptual models that was used in [SNF10] to generate spatial database. That UML profile was also used in an MDA architecture to generate the SQL DDL code for spatial databases [JFD+13]. Many GIS standards from ISO, OGC, and INSPIRE include metamodels covering different parts of a GIS. Kutzner [Kut16] addressed the model-driven transformation of geospatial data according to different metamodels that can present differences between them.

In [CLP+17, CLPP17] we have already considered the application of automated software development to the GIS domain and we proposed an architecture and a tool for this purpose combining SPLE and MDE approaches. We analyzed the features of a generic family of GIS products and the components implementing these features, we designed a feature model [BSRC10b] to represent this set of features, and we defined a traditional two-level metamodel to specify how the data model of the products of our family can be described. Afterward, we implemented a tool supporting the automatic generation of GIS products from these models. In [ACL+20], that metamodel was used to define a DSL for GIS development.

Even though our tool and the metamodels it handles are very flexible and complete, this design following the two-level modeling approach has some caveats. First, it forces us to define all possible elements of a GIS in a single metamodel. This does not allow us to reflect in the model common structures in this domain, such as defining entities that refine other entities at higher levels of abstraction. Second, since GIS manage entities with a spatial component in the real world, it is relatively common that some structures appear repeatedly with some adaptations and particularities. Working with a two-level approach does not easily allow us to take any advantage of this scenario. Defining those structures in the metamodel would allow us to use them directly, but they would be difficult to adapt to the particularities of a specific application.

As we will explain in section 4.2, these disadvantages can be addressed by applying multilevel modeling. For example, let us assume we need to develop a GIS that allows a city manager to handle the road networks, the public transportation networks, and also the electricity, water, and telecommunication supply networks. As we will see in the next section, all these networks share the same structure, although they may differ in specific attributes of the network elements. Our proposal shows that applying a multilevel approach allows us to metamodel, at an intermediate level, the most common GIS structures, but allowing us to easily extend and adapt them, so we can use these structures to make simpler models in the lower level.

4.2 Conceptualizing GIS with a Multilevel Modelling Approach

We have identified four scenarios in which multilevel modeling provides a clear advantage over the traditional two-level modeling approach. In each of the scenarios we describe how they are currently modeled in international standards from ISO, OGC, and the EU INSPIRE Directive, then we present our proposal for modeling those scenarios with multilevel modeling, and we end with a discussion of its advantages compared with a two-level approach.

4.2.1 Multilevel notation and patterns

Before describing our proposal, we introduce the notation used in the multilevel models and a set of recurring patterns that will be addressed during the discussion of our multilevel proposals.

There is not a standard language for multilevel modeling yet. Lately, there has been an attempt to identify the most common characteristics of the existing proposals, or the requirements that these proposals target [Fra18]. Tooling support for creating multilevel models is scarce, and the existing prototypes have certain limitations. Given that we have to represent several large models, for convenience we decided to use a standard UML tool to create them.

In this work, we express the multilevel models using the notation presented in [dLG10]. We show an example of the notation in fig. 4.1, inspired by examples from [dLGC14]. We use the symbol "@" to indicate the potency of the meta-classes or clabjects [AK02]. The *potency* of an element represents the number of meta-levels a property needs to be instantiated before we get a plain instance and hence we have to assign it a value [AK02]. If an element does not have an explicit potency indicated, it takes the potency of its container. Each element, class, or relationship that instantiates a higher-level element is underlined, and the instantiated element is indicated after the symbol ":". For our explanations along this section, we will use the concepts potency and classification level indistinctly, and when we refer to a meta-level with a particular potency assigned we use also the symbol "@" in the text (e.g., *meta-level @5* means *the level assigned potency 5*, or *the set of elements with potency 5*). As an example, we describe fig. 4.1, where we can see three meta-levels, assigned to potency 2 to 0. In meta-level @2, or meta-level assigned potency 2, we have a meta-class *Product* without any explicit potency. Therefore, it has potency @2, which is the one assigned to the meta-level. Its attribute *vat* has potency 1. Therefore, it must be assigned a value one meta-level below, when the meta-class *Product* is instantiated into *Publication* and *vat* takes the value "4.0". We can also see that the relationship *published by*, with potency 1, is an instance of the relationship *made by*, with potency 2.

When designing meta-level models there are patterns that appear often, the same way that in traditional two-level modeling there are a well known set of design and architectonic patterns. An effort to identify and describe these patterns was done in [dLGC14], where each pattern is shown with examples and compared to other two-level solutions. These recurring patterns appear on situations where multilevel modeling is adequate, and therefore we will identify them in the example models we present in this section. The patterns are briefly described next, referring to elements shown in fig. 4.1 to illustrate them:

- *Type-object* pattern: modeling types and instances of these types dynamically (e.g., the meta-classes *Publication* or *Book*).
- *Dynamic features* pattern: adding a feature to a dynamic type, and to all the instances of this new type (e.g., the attribute *vat*).

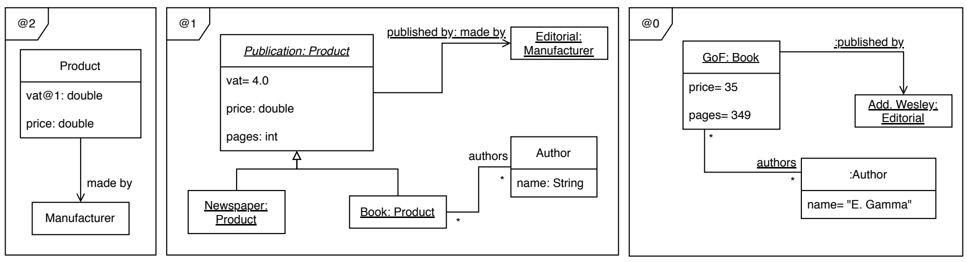


Figure 4.1: Example models showing the notation used for multilevel modeling and common multilevel modeling patterns. Inspired by examples from [dLGC14].

- *Dynamic auxiliary domain concepts* pattern: adding domain related entities that have relationships with existing dynamic types (e.g., the meta-class *Author* and the relationship *authors*).
- *Relation configurator* pattern: allowing the configuration of a relationship (e.g., the relationship *published by*).
- *Element classification* pattern: dynamically create hierarchies of elements, allocating the features that are inherited by the child types (e.g., the hierarchy descending from the abstract meta-class *Publication*).

Finally, the classes that define geographic data types in the international standards are used as UML data types in the models (e.g., 4.3). Just like we can use *String* or *Integer* as the class of an attribute, we use geographic data types such as *Geometry* or *Point.* It may seem that if a class includes an attribute of a geographic type, this should be modeled as a relationship between the class and the class that represents the geographic type. However, we decided not to represent such relationships in that way for three reasons: 1) ISO/OGC UML models consider geographic classes as data types, even though they do not use the stereotype dataType; 2) in the logical and physical levels of a GIS, a geographic class works as a data type: it is not implemented using an association between objects in Java, but as a data type, and it is not represented using a foreign key to a table of geographic values in the database, but in the same row of the database; 3) the geographic attribute of the *SpatialEntity* meta-class is used and redefined in many other models, so if we represent it as a relationship it would worsen the legibility of these models.

4.2.2 Bridging the gap between conceptual and implementation standards in GIS

4.2.2.1 Overview

In recent years, there has been a great effort of standardization in the field of GIS. Two international organizations (ISO/TC 211 and OGC) have defined around a hundred standards for GIS that cover a multitude of aspects (e.g., conceptual models, logical models, physical models, web services). ISO/TC 211, being an organization composed of the standardization agencies of the member countries, has focused on the definition of conceptual standards aimed at providing solutions for general problems. On the other hand, OGC, being an organization composed of companies in the GIS sector, has focused on the definition of implementation standards aimed at solving problems of interoperability between tools and datasets. This has created a gap between the two sets of standards. Although the two sets of standards are focused on the same sector, they are not formally connected and they just have textual references between them. Multilevel modeling would allow to build a bridge between both sets of standards and have a more consistent and reusable domain description.

Consider as an example the standards that define a model for describing the spatial characteristics of geographic entities using vector geometries. The ISO standard *ISO 19107: Geographic Information - Spatial Schema* [Inta] defines primitive data types such as *GM_Point*, *GM_Curve*, and *GM_Surface*, aggregate data types such as *GM_MultiPoint*, *GM_MultiCurve*, and *GM_MultiSurface*, as well as many other data types. The data types are structured in a hierarchy that allows application schemas to use data types from the higher levels of the hierarchy to represent spatial entities whose spatial component can be of any of the data types. The schema described in ISO 19107 is conceptual, and it provides no implementation details. OGC Simple Feature Access (OGC SFA) [Thea], which is also an ISO standard (ISO 19125 [Intc]), offers the most popular implementation of ISO 19107, describing a common architecture for simple feature geometry. In practice, most GIS data models use the spatial data types from OGC SFA, like *Point*, *LineString*, or *Polygon* (some data types from OGC SFA match the names of ISO 19107 without the namespace prefix). Even though ISO and OGC have worked together on the definition of the standards (in fact, OGC SFA is also an ISO standard, ISO 19125 [Intc]), the connection between the standards is limited to an informative annex in ISO 19125 that "identifies similarities and differences" with textual descriptions without formal models (e.g., Annex A ISO 19125 [Intc] states that "MultiPoint in SFA-CA corresponds to GM_MultiPoint in Spatial Schema" but it does not provide a UML model).

4.2.2.2 Multilevel modeling solution

Figure 4.2 shows an excerpt of the metamodels that bridge the gap between the ISO conceptual model and the OGC implementation model. The complete models are included in appendix B (fig. B.1 and fig. B.2). The left part of the figure shows part of the model for meta-level @6 that corresponds to the ISO *GM_Point* data type (a geographic point). The right part shows part of the model for the metal-level @5 that corresponds to the OGC *Point* data type. The OGC data types defined in meta-level @5 instantiate the data types defined by ISO at meta-level @6. We have also defined a class in meta-level @6 (*SpatialEntity*) that we use in the following models (section 4.2.3, section 4.2.4, and section 4.2.5) to describe geographic attributes without specifying the specific type of geographic object until the application schema.

4.2.2.3 Discussion

If the ISO and OGC standards were defined following a multilevel approach, the connection between the models would be explicit and based on a model, rather than implicit and based on a textual explanation. New implementation standards based on the ISO conceptual

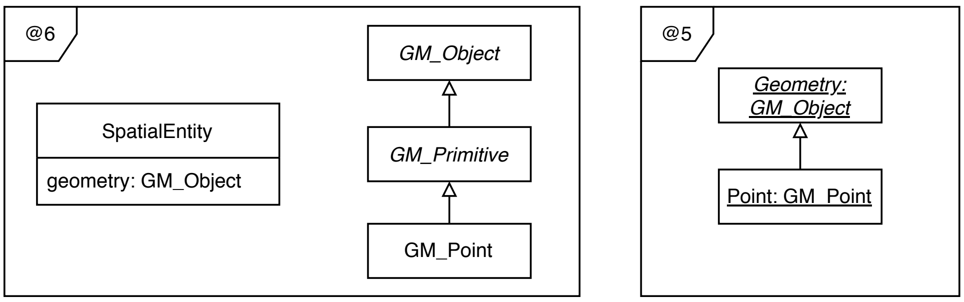


Figure 4.2: Excerpt from the multilevel solution to bridge the ISO conceptual model and the OGC implementation model. Figure B.1 and Figure B.2 in appendix B show the extended models

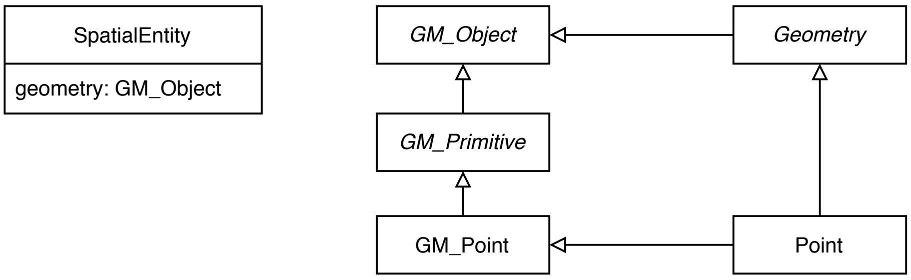


Figure 4.3: Two-level solution to bridge the ISO conceptual model and the OGC implementation model

standard could be defined (e.g., an implementation model that used Bézier curves instead of line segments in the definition of geometries). These new standards would remain connected with the ISO conceptual model, which would allow applying MDE techniques to all ISO and OGC standards (e.g., using model transformation techniques to convert data between the models).

Furthermore, defining a *SpatialEntity* class at meta-level @6 would allow the decision of the specific data type of an attribute to be deferred until the definition of the application data model. This would allow intermediate models (such as those defined by INSPIRE, see section 4.2.3, section 4.2.4, and section 4.2.5) to be independent of the implementation, and hence allowing developers to design applications without selecting the implementation technology.

Figure 4.3 shows a two-level model similar to the one presented in fig. 4.2. The connection between ISO 19107 and OGC-SFA datatypes has to be made through inheritance, which would reduce the readability of the model. Furthermore, multiple inheritances would cause difficulties in languages that do not support it, in addition to all the traditional difficulties of multiple inheritance in object-oriented programming languages. Furthermore, the *SpatialEntity* class would have to be associated with *GM_Object*, and although a particular application might use a subclass of *GM_Object*, it cannot be explicitly reflected in the application schema.

Regarding the complexity of the models, the number of classes in the multilevel models and the 2-level models is the same because the goal is to replicate the same set of data types. However, the number of associations is much lower because in the multilevel models we instantiate the ISO conceptual model data types into OGC conceptual model data types instead of using inheritance relationships. In particular, we avoid 8 inheritances. There are 6 other inheritances that would be avoided, but they are not shown in our models because we have not included the complete ISO 19107 model because it is a 239 page document in which a large number of data types are defined. As an example, *GM_Curve* is the root of a hierarchy of specialization that includes 17 children classes. The classes *LineString*, *Line* and *LinearRing* from the OGC-SFA model would inherit from some of these classes.

4.2.3 Ensuring interoperability in spatial data infrastructures

4.2.3.1 Overview

The European Parliament and the Council of The European Union approved on 2007 the Directive 2007/2/EC establishing an *Infrastructure for Spatial Information in the European Community* (INSPIRE), to create "a European Union spatial data infrastructure for the purposes of EU environmental policies or activities which may have an impact on the environment". This directive is a promising initiative in the management of spatial data in different countries regarding the interoperability and sharing of data.

One of the main parts of INSPIRE is the technical guidelines (called data specifications[5]) that specify common data models to achieve interoperability of spatial data sets and services across Europe. fig. 4.4 describes the organization of these data specifications. The bottom layer of fig. 4.4 contains the 34 data specifications that provide UML application schema for the themes of interest for public administrations regarding resource management and information needed to monitor and define environmental policies. Some example data specifications are Administrative Units, Transport Network, or Production and Industrial Facilities. The middle layer of fig. 4.4 contains the UML application schema defined by INSPIRE that are reused across the 34 data specifications. As an example, INSPIRE defines a generic application schema for Networks that provides basic types that are extended in other data specifications such as the data specification for transport networks (it covers road networks and rail networks among others), or the data specification for Utility and Governmental Services (it covers water networks and electrical networks, among others). The top layer of fig. 4.4 contains UML application schema defined by third parties that are used by INSPIRE data specifications. For example, it contains the ISO standard *ISO 19107: Geographic Information - Spatial Schema* [Inta].

Considering the characteristics of INSPIRE we have mentioned, this is a clear example of a problem where multilevel modeling is well-suited. One of the drawbacks of INSPIRE is its complexity. Across the 34 technical guidelines, we can find many elements that share a set of common attributes and relationships. Sometimes these common elements are not reflected in the technical guidelines, which can lead to repeating the same structures

[5]`https://inspire.ec.europa.eu/data-specifications/2892`

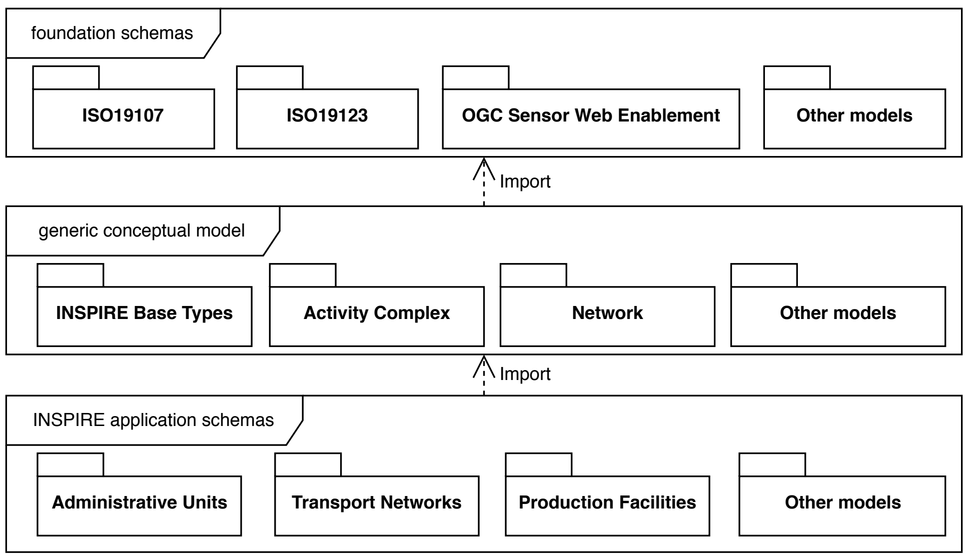


Figure 4.4: Overview of the INSPIRE Data Specifications. The top layer are ISO/OGC international standards, the middle layer are INSPIRE common models, the bottom layer contains the 34 INSPIRE data specifications.

and patterns in different domains. In other cases, those commonalities are identified and considered in the technical guidelines, but through complex and deep inheritance hierarchies that can be difficult to specialize and adapt to the particularities of a specific country.

Consider as an example the application domain of territory administration. Its main responsibility is representing the boundaries of spaces and territories, both rural and urban. Typically, the administration of the territory divides the geographic space into administrative units, that can be composed of other administrative units, and so on. For example, Spain is divided into 17 *autonomous communities* and 2 *autonomous cities*. Each autonomous community is further divided into *provinces* that are divided into *municipalities*. Autonomous cities are not divided into provinces and they contain a single municipality. On the other hand, Portugal is divided into 18 *districts* and 2 *autonomous regions*, both are divided into *municipalities*, which are themselves divided into *parishes*. Hence, each country has its own structure for its territory, with its own names, levels and restrictions.

To support all this variability, INSPIRE has defined a generic model[6] that can be applied to all of them, independently of their particularities, shown in fig. 4.5.

The class called *AdministrativeUnit* represents any part of the territory, from a whole country to a small village. The administrative hierarchy level is stored with a generic enumerated, where values go from *1stOrder* (country) to *6thOrder* (smallest administrative level of a country). Other attributes are the name (or names), the INSPIRE identification

[6]INSPIRE Data Specification on Administrative Units – Technical Guidelines: `https://inspire.ec.europa.eu/Themes/114/2892`

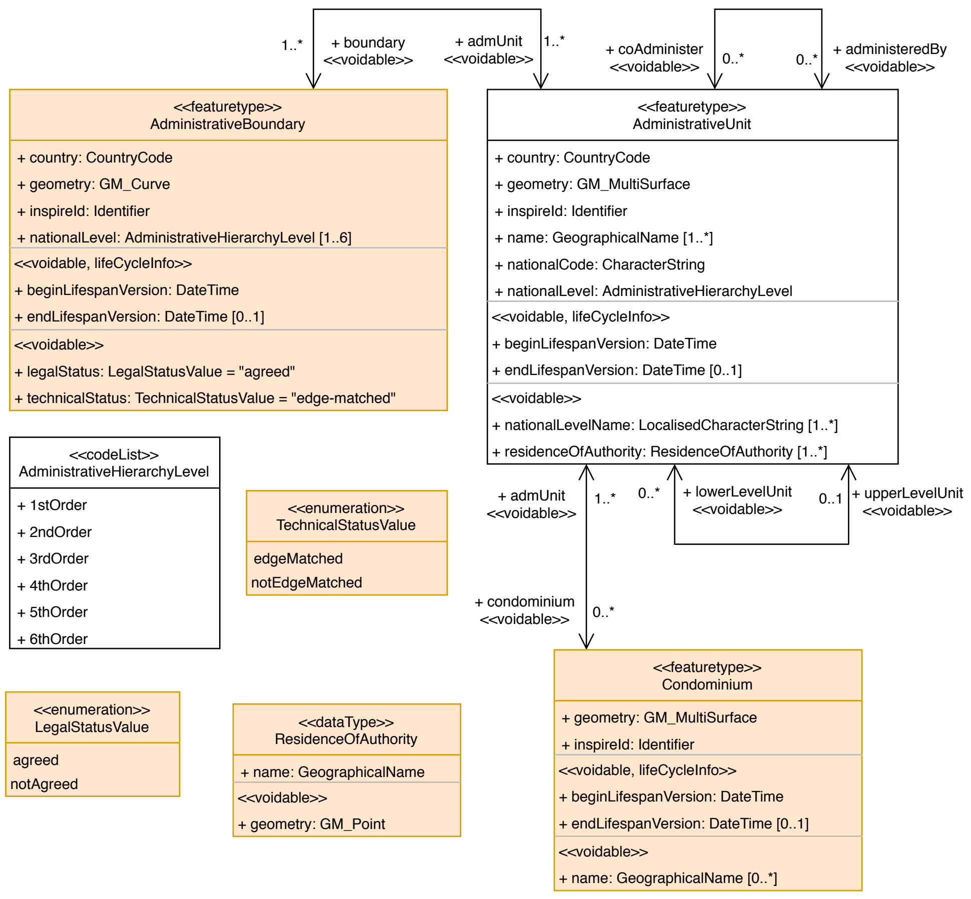


Figure 4.5: INSPIRE Administrative Units Overview, from `https://inspire.ec.europa.eu/data-model/approved/r4618-ir/html/index.htm?goto=2:1:2:1:7106`

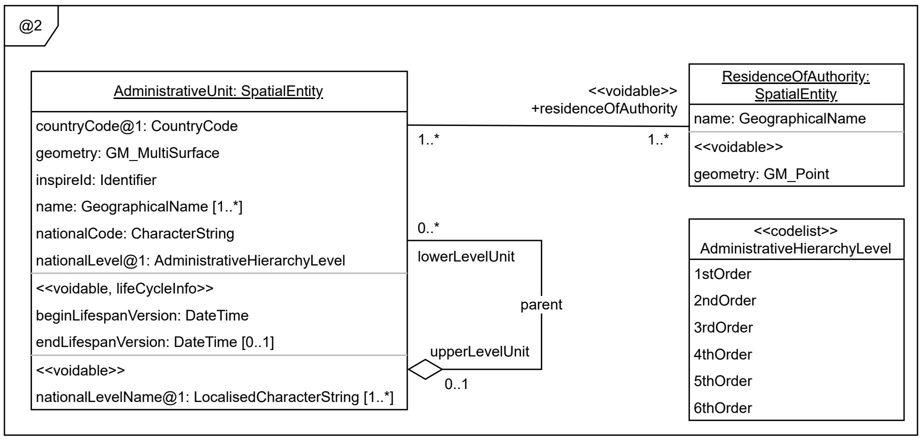

@2
AdministrativeUnit: SpatialEntity
countryCode@1: CountryCode
geometry: GM_MultiSurface
inspireId: Identifier
name: GeographicalName [1..*]
nationalCode: CharacterString
nationalLevel@1: AdministrativeHierarchyLevel
<<voidable, lifeCycleInfo>>
beginLifespanVersion: DateTime
endLifespanVersion: DateTime [0..1]
<<voidable>>
nationalLevelName@1: LocalisedCharacterString [1..*]
<<voidable>>
+residenceOfAuthority
1..*
1..*
0..*
lowerLevelUnit
parent
upperLevelUnit
0..1
ResidenceOfAuthority: SpatialEntity
name: GeographicalName
<<voidable>>
geometry: GM_Point
<<codelist>>
AdministrativeHierarchyLevel
1stOrder
2ndOrder
3rdOrder
4thOrder
5thOrder
6thOrder

Figure 4.6: Modeling multilevel Territory Administration - Meta-level @2

(a unique identifier within all Europe), and the geometry representing the surface of the territory. This class also stores information about the country to which it belongs, and an identifier within this country. Finally, there are some optional attributes to represent the version of each instance of the class, to store the *residence of authority* (usually, a capital city), or the name of the administration level of the instance within the country (for example, in Spain the administrative units of 3rd level are called *Provincias*). The *ResidenceOfAuthority* has its own representation, which is very simple, just the name of the place and its geometry. Each administrative unit can aggregate a series of administrative units of a lower level. For example, a country and the regions of this country are related, being the former the *upperLevelUnit* and the latter the *lowerLevelUnit* of the relationship we can see in fig. 4.5.

Given that the INSPIRE data specifications are abstract, each member country of the European Union is expected to adapt them to their particularities. Even though the model in fig. 4.5 can be used to represent the hierarchical division of the territory of a country, it has some drawbacks. First, the semantics of the administrative division of a country are missing. For instance, it is possible that objects from an upper level (e.g., an autonomous community in Spain, a 2nd order division) aggregate objects from the wrong lower level (e.g., municipalities in Spain, a 4th order division). Second, a member country may require that objects from each level of the administrative division have additional attributes. For example, autonomous communities and municipalities in Spain have specific legislation but provinces do not. Hence, the classes for the 2nd and 4th level in Spain require attributes describing the legislation but the class for the 3rd level does not require the attribute.

4.2.3.2 Multilevel modeling solution

In order to solve these drawbacks, we use multilevel modeling. We define a meta-level @2 in fig. 4.6 that is similar to the one defined by INSPIRE. This meta-level @2 model can be instantiated in each member country in such a way that it describes the semantics of its

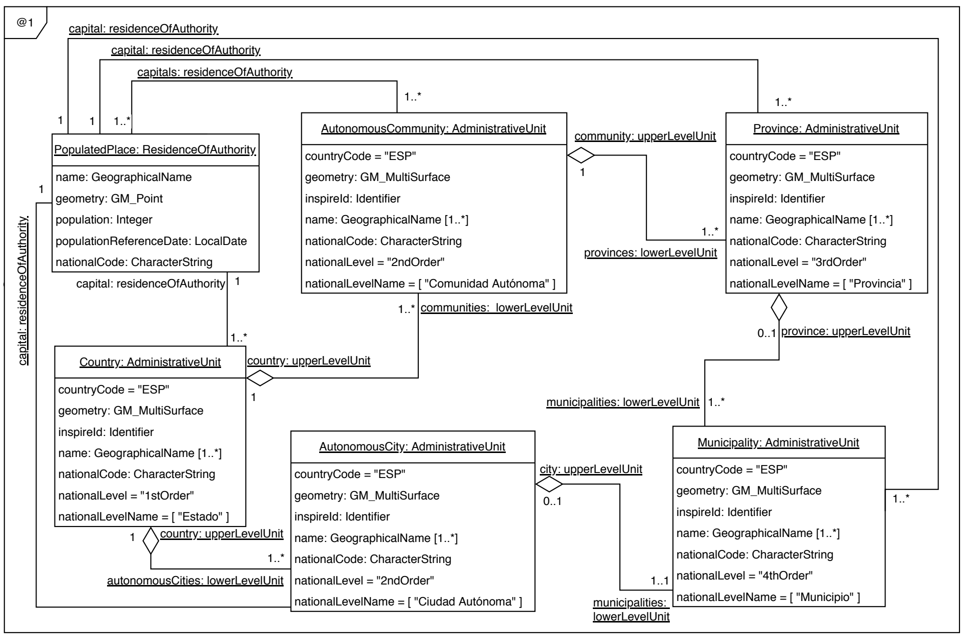


Figure 4.7: Modeling multilevel Territory Administration - Meta-level @1 for Spain

administrative division.

In figs. 4.7 and 4.8 we show the model for the meta-level @1 of two particular countries: Spain and Portugal. Being both territorial structures very similar, there are certain particularities in the administration of these countries that are specified in these levels. Spain (fig. 4.7) is composed by *AutonomousCommunities*, that are composed by *Provinces*, and these ones by *Municipalities*. There are also *AutonomousCities*, composed each one by a single *Municipality*. Regarding the residence of authority linked to the administrative units, all have exactly one *PopulatedPlace* (that represents a city or village) as capital, except in the case of *AutonomousCommunities* because some of them can have more than one capital (e.g., the capital of the Canary Islands is shared by Santa Cruz de Tenerife and Las Palmas de Gran Canaria).

The territorial division of Portugal (fig. 4.8) is a bit different. The first level of the administrative division (2nd order from the EU point of view) consists of *Districts* and *Autonomous Regions* (i.e. Azores and Madeira). Both of them are composed by *Municipalities*, which are themselves composed by *Parishes*.

The particularities of each country are explicit in the meta-level @1 metamodels by redefining the relationships between administrative units and residences of authority. Of course, any extra attribute required for modeling the administrative divisions of these countries could be added at this level (e.g., we have added a few attributes to *PopulatedPlace*), and many attributes are instantiated at this level, such as *countryCode* or

@1
capital: residenceOfAuthority
capital: residenceOfAuthority
capitals: residenceOfAuthority
PopulatedPlace: ResidenceOfAuthority
name: GeographicalName
geometry: GM_Point
population: Integer
populationReferenceDate: LocalDate
nationalCode: CharacterString
AutonomousCommunity: AdministrativeUnit
countryCode = "ESP"
geometry: GM_MultiSurface
inspireId: Identifier
name: GeographicalName [1..*]
nationalCode: CharacterString
nationalLevel = "2ndOrder"
nationalLevelName = ["Comunidad Autónoma"]
community: upperLevelUnit
provinces: lowerLevelUnit
Province: AdministrativeUnit
countryCode = "ESP"
geometry: GM_MultiSurface
inspireId: Identifier
name: GeographicalName [1..*]
nationalCode: CharacterString
nationalLevel = "3rdOrder"
nationalLevelName = ["Provincia"]
capital: residenceOfAuthority
communities: lowerLevelUnit
province: upperLevelUnit
Country: AdministrativeUnit
countryCode = "ESP"
geometry: GM_MultiSurface
inspireId: Identifier
name: GeographicalName [1..*]
nationalCode: CharacterString
nationalLevel = "1stOrder"
nationalLevelName = ["Estado"]
country: upperLevelUnit
municipalities: lowerLevelUnit
AutonomousCity: AdministrativeUnit
countryCode = "ESP"
geometry: GM_MultiSurface
inspireId: Identifier
name: GeographicalName [1..*]
nationalCode: CharacterString
nationalLevel = "2ndOrder"
nationalLevelName = ["Ciudad Autónoma"]
city: upperLevelUnit
Municipality: AdministrativeUnit
countryCode = "ESP"
geometry: GM_MultiSurface
inspireId: Identifier
name: GeographicalName [1..*]
nationalCode: CharacterString
nationalLevel = "4thOrder"
nationalLevelName = ["Municipio"]
country: upperLevelUnit
autonomousCities: lowerLevelUnit
municipalities: lowerLevelUnit
capital: residenceOfAuthority

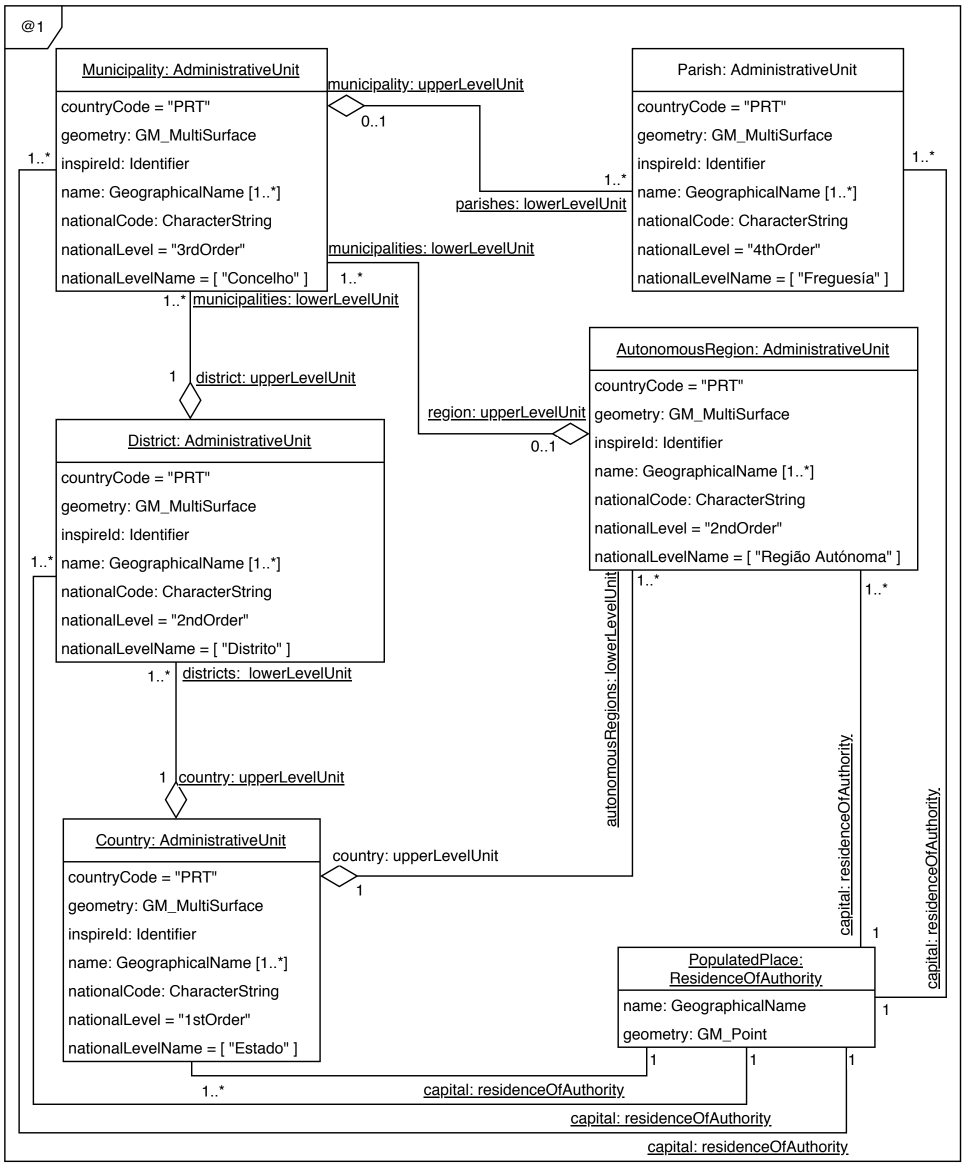

@1
Municipality: AdministrativeUnit
countryCode = "PRT"
geometry: GM_MultiSurface
inspireId: Identifier
name: GeographicalName [1..*]
nationalCode: CharacterString
nationalLevel = "3rdOrder"
nationalLevelName = ["Concelho"]
Parish: AdministrativeUnit
countryCode = "PRT"
geometry: GM_MultiSurface
inspireId: Identifier
name: GeographicalName [1..*]
nationalCode: CharacterString
nationalLevel = "4thOrder"
nationalLevelName = ["Freguesía"]
municipality: upperLevelUnit
0..1
1..*
parishes: lowerLevelUnit
municipalities: lowerLevelUnit
1..*
1..* municipalities: lowerLevelUnit
1 district: upperLevelUnit
District: AdministrativeUnit
countryCode = "PRT"
geometry: GM_MultiSurface
inspireId: Identifier
name: GeographicalName [1..*]
nationalCode: CharacterString
nationalLevel = "2ndOrder"
nationalLevelName = ["Distrito"]
AutonomousRegion: AdministrativeUnit
countryCode = "PRT"
geometry: GM_MultiSurface
inspireId: Identifier
name: GeographicalName [1..*]
nationalCode: CharacterString
nationalLevel = "2ndOrder"
nationalLevelName = ["Região Autónoma"]
region: upperLevelUnit
0..1
1..* districts: lowerLevelUnit
1 country: upperLevelUnit
autonomousRegions: lowerLevelUnit
1..*
1..*
capital: residenceOfAuthority
capital: residenceOfAuthority
Country: AdministrativeUnit
countryCode = "PRT"
geometry: GM_MultiSurface
inspireId: Identifier
name: GeographicalName [1..*]
nationalCode: CharacterString
nationalLevel = "1stOrder"
nationalLevelName = ["Estado"]
country: upperLevelUnit
1
PopulatedPlace: ResidenceOfAuthority
name: GeographicalName
geometry: GM_Point
1
1
1 1 1
1..*
1..*
1..*
capital: residenceOfAuthority
capital: residenceOfAuthority
capital: residenceOfAuthority

Figure 4.8: Modeling multilevel Territory Administration - Meta-level @1 for Portugal

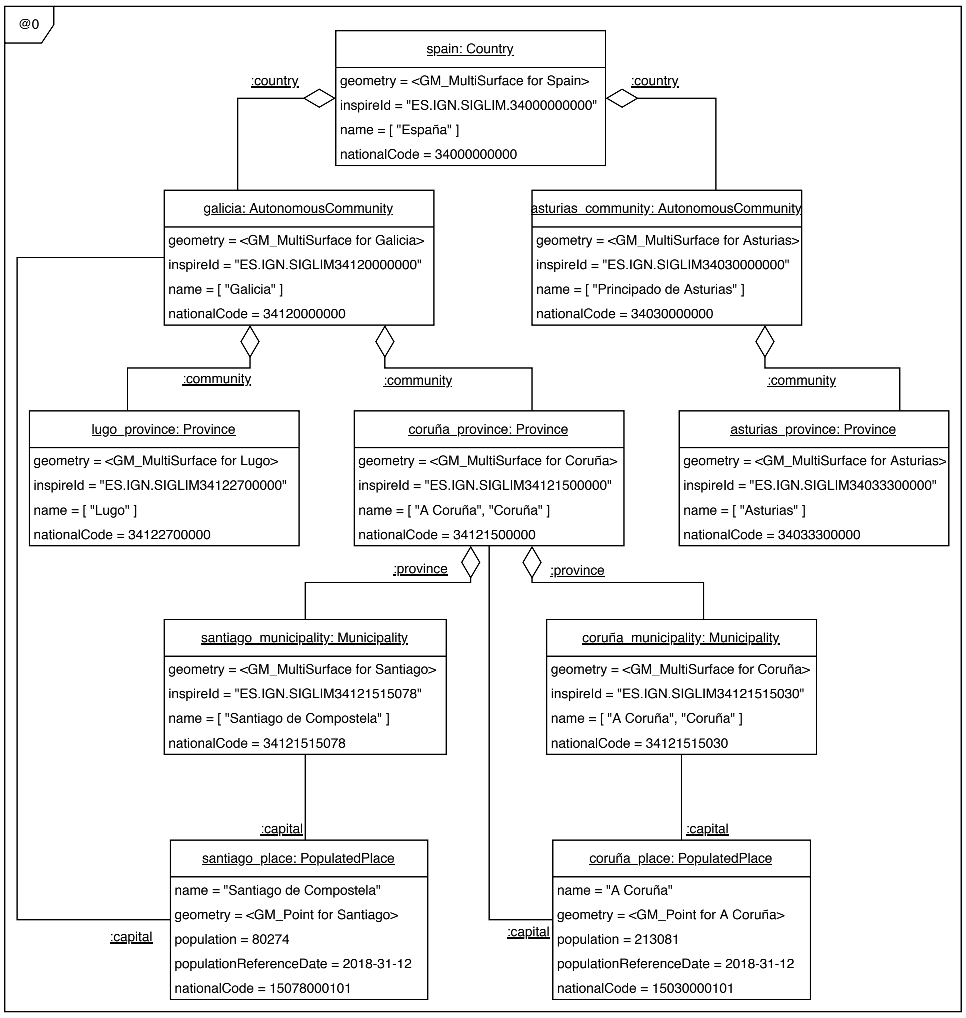

@0
spain: Country
geometry = <GM_MultiSurface for Spain>
inspireId = "ES.IGN.SIGLIM.34000000000"
name = ["España"]
nationalCode = 34000000000
:country
:country
galicia: AutonomousCommunity
geometry = <GM_MultiSurface for Galicia>
inspireId = "ES.IGN.SIGLIM34120000000"
name = ["Galicia"]
nationalCode = 34120000000
asturias_community: AutonomousCommunity
geometry = <GM_MultiSurface for Asturias>
inspireId = "ES.IGN.SIGLIM34030000000"
name = ["Principado de Asturias"]
nationalCode = 34030000000
:community
:community
:community
lugo_province: Province
geometry = <GM_MultiSurface for Lugo>
inspireId = "ES.IGN.SIGLIM34122700000"
name = ["Lugo"]
nationalCode = 34122700000
coruña_province: Province
geometry = <GM_MultiSurface for Coruña>
inspireId = "ES.IGN.SIGLIM34121500000"
name = ["A Coruña", "Coruña"]
nationalCode = 34121500000
asturias_province: Province
geometry = <GM_MultiSurface for Asturias>
inspireId = "ES.IGN.SIGLIM34033300000"
name = ["Asturias"]
nationalCode = 34033300000
:province
:province
santiago_municipality: Municipality
geometry = <GM_MultiSurface for Santiago>
inspireId = "ES.IGN.SIGLIM34121515078"
name = ["Santiago de Compostela"]
nationalCode = 34121515078
coruña_municipality: Municipality
geometry = <GM_MultiSurface for Coruña>
inspireId = "ES.IGN.SIGLIM34121515030"
name = ["A Coruña", "Coruña"]
nationalCode = 34121515030
:capital
:capital
santiago_place: PopulatedPlace
name = "Santiago de Compostela"
geometry = <GM_Point for Santiago>
population = 80274
populationReferenceDate = 2018-31-12
nationalCode = 15078000101
coruña_place: PopulatedPlace
name = "A Coruña"
geometry = <GM_Point for A Coruña>
population = 213081
populationReferenceDate = 2018-31-12
nationalCode = 15030000101
:capital
:capital

Figure 4.9: Modeling multilevel Territory Administration - Example of Level 0 for Spain

nationalLevelName, simplifying meta-level @0 models. In fig. 4.9 there is an example of the meta-level @0 model of an application based on the metamodel for Spain.

4.2.3.3 Discussion

The main advantage of applying multilevel modeling in this scenario is that the model of meta-level @2 can be instantiated in such a way that it can capture the semantics of each member country. Hence, the application schema of each member country forbids that objects from an upper level (e.g., an autonomous community in Spain, a 2nd order division) aggregate objects from the wrong lower level (e.g., municipalities in Spain, a 4th order division). The same goal could have been achieved in a traditional two-level approach using inheritance, but it would require a complex inheritance hierarchy and the redefinition of the association between administrative units defined in the INSPIRE data specification. This solution would be less reusable and flexible than the multilevel solution that we propose.

Another advantage is that member countries may now easily add additional attributes to specific levels of the administrative division modifying the meta-level @1 as desired. Again, the same goal could have been achieved using an inheritance hierarchy, but the resulting model would not be as clear.

Finally, the advantages described in section 4.2.2.3 also apply in this scenario. First, the models of the member countries would be explicitly and formally connected, and hence data transformation techniques could be used to achieve interoperability of the applications. Second, using the *SpatialEntity* class from meta-level @6 would make INSPIRE Data Specifications independent of the implementation and it would still be easy to select a data type from the implementation model in the application schema.

In the example models described we can find several common patterns of multilevel modeling (see section 4.2.1):

- *Type-object* pattern: e.g., *AdministrativeUnit* is instantiated into different types in lower meta-levels, such as *Country* or *Municipality* (see fig. 4.7).
- *Dynamic features* pattern: e.g., the attributes included in meta-classes of the meta-level @1, such as *population* or *nationalCode* in *PopulatedPlace* (see fig. 4.7). Also, almost every instance of *SpatialEntity* in all the examples include new attributes, since the meta-class *SpatialEntity* is very generic. For example, *inspireId*, *name* or *countryCode* in *AdministrativeUnit* of the meta-level @2 (see fig. 4.6).
- *Relation configurator* pattern: both the relationships *parent* and *residenceOfAuthority* defined in the meta-level @2 (see fig. 4.6) are configured or redefined in the meta-level @1, changing the cardinality and the end classes in every case. For example, in the case of the association between *Province* and *AutonomousCommunity* (see fig. 4.7).

Regarding the complexity of the models, the number of classes is the same in the INSPIRE models and in our proposal. However, the INSPIRE models would require many inheritance associations (i.e., one for each administrative division of each member country) that are represented as instantiations in our models. Furthermore, the INSPIRE models would not easily represent the semantics of the administrative divisions of each member country because UML does not provide simple notation to specify that an association is specialized in a inheritance hierarchy (i.e., constraints must be used).

As a conclusion, the solution based on multilevel modeling is more expressive, flexible and simple. It also ensures interoperability between member countries because it ensures that the model of each country remains formally connected to the INSPIRE model while allowing the addition of country-specific semantics.

4.2.4 Modeling common GIS structures

4.2.4.1 Overview

Spatial networks are one of the most common data model structures in GIS. Many domains require modeling and processing different types of networks. Two of the most common domains are transport networks, such as those representing roads, railways, or flight routes, and resource distribution networks, such as electricity supply, telecommunications, or water supply networks.

From the most abstract point of view, a network is composed of nodes and edges. When a GIS is used to model networks, both nodes and edges of a spatial network are spatial entities because they represent real-world geographic features. Each node has a location, which is usually a point in the space (a *GM_Point* geometry). The edges of the network are defined by the two nodes they connect. If the edge is directed, one node plays the role of the *source*, and the other plays the role of the *target*. In most cases, edges are defined in the space by a line or curve (a *GM_Curve* geometry). Even though the data model for spatial networks is clearly understood (it was already defined in [G9̈4]), each GIS application schema has to redefine the same classes for networks composed of edges and nodes instead of referencing the spatial network definition as a common structure.

The idea of reusing the model that defines spatial networks was applied by the designers of the INSPIRE data specifications. Given that spatial networks are required by three data specifications in INSPIRE (namely, transport networks, hydrography, and utility and government services), they have defined a generic network model as part of its base models[7]. Then, each data specification extends the classes in the Generic Network Model to define a common set of base classes for transport networks[8], hidrography[9], and utility networks[10]. Finally, each data specification defines classes for specific network types extending the classes of the common model (e.g., the specification for transport networks defines classes for road railway, air, water, and cable transport networks, and the specification for utility networks defines classes for electricity, oil-gas-chemicals, water, sewer, and thermal networks). The result is a set of quite complex models with a very deep inheritance hierarchy that makes the model quite difficult to understand.

4.2.4.2 Multilevel modeling solution

Figure 5.2 shows a simple example of a multilevel solution for modeling networks with four levels. The upper level (not shown) consists of the meta-level @6 of spatial entities

[7]INSPIRE Data Specifications – Base Models – Generic Network Model: `https://inspire.ec.europa.eu/documents/inspire-data-specifications-%E2%80%93-base-models-%E2%80%93-generic-network-model`

[8]INSPIRE Data Specification on Transport Networks – Technical Guidelines: `https://inspire.ec.europa.eu/Themes/115/2892`

[9]INSPIRE Data Specification on Hidrography – Technical Guidelines: `https://inspire.ec.europa.eu/Themes/116/2892`

[10]INSPIRE Data Specification on Utility and Government Services – Technical Guidelines: `https://inspire.ec.europa.eu/Themes/136/2892`

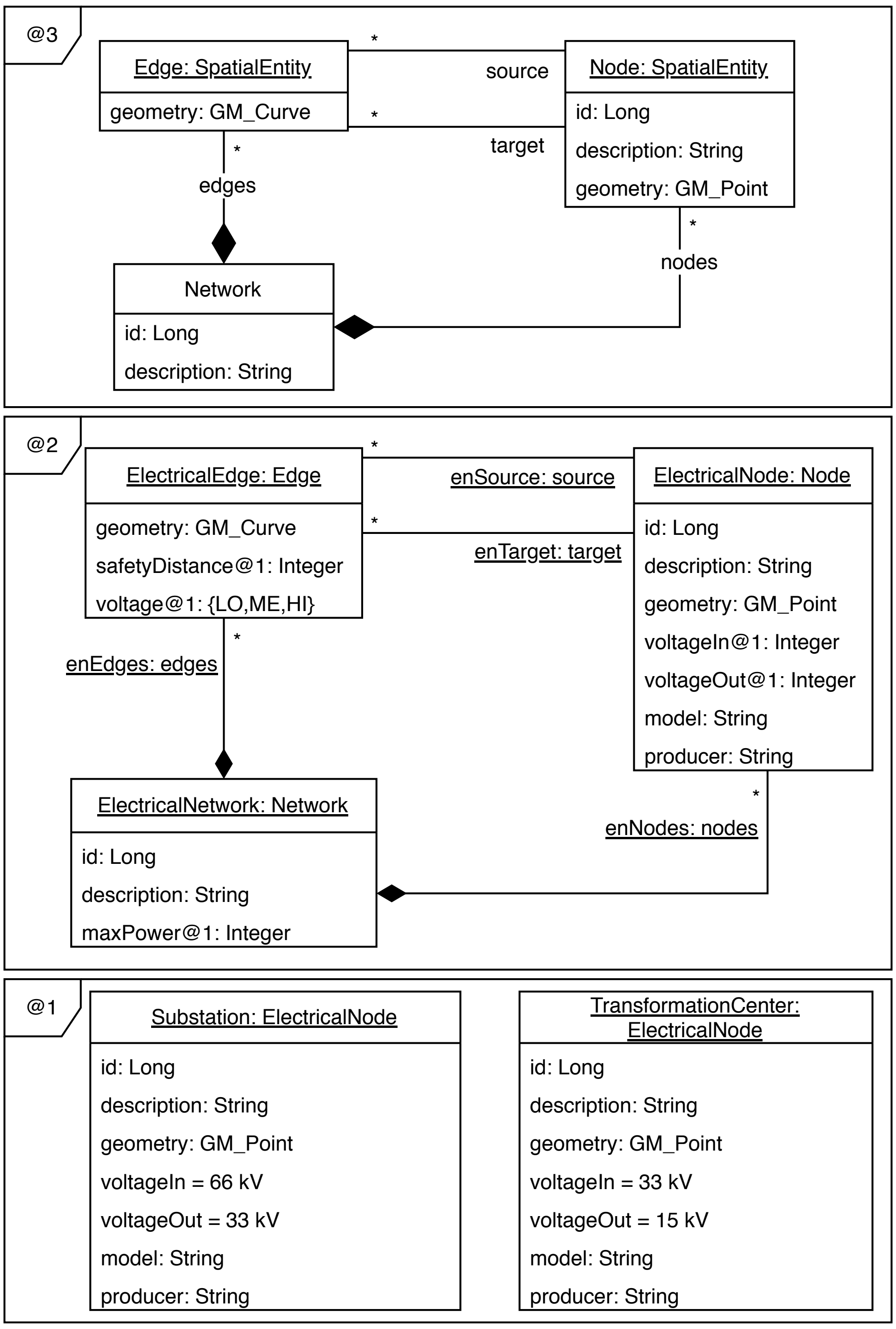


Figure 4.10: Modeling multilevel Spatial Networks - A simple example

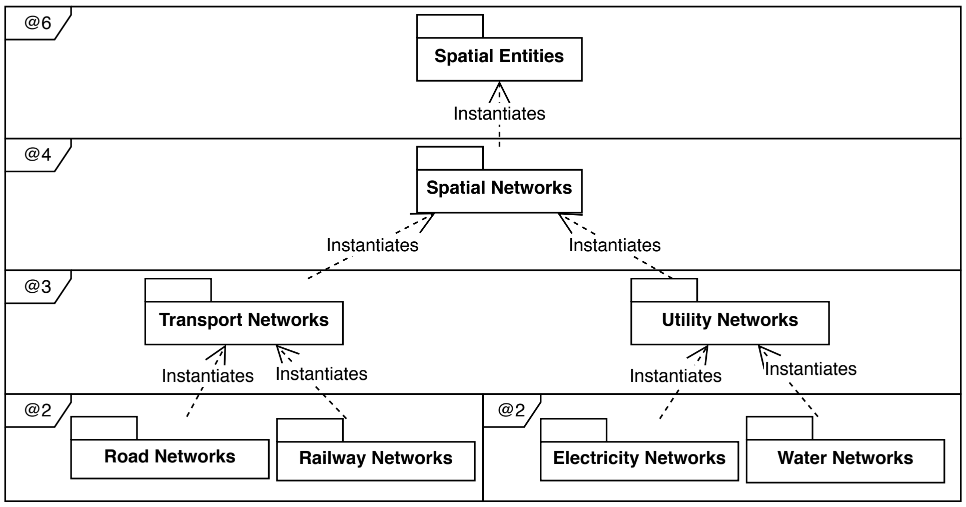


Figure 4.11: Modeling multilevel Spatial Networks - Overview

defined in fig. 4.2. The meta-level @3 is used to model the generic structure for a network, independently of the nature of either the nodes or edges. *Node*s are spatial entities for which the geometry is a point. Besides, all nodes must have an identifier and a description. *Edge*s are spatial entities too, but their geometry is a line. In addition, edges have two associated nodes (source and target). All these attributes are instantiated in meta-level @0.

The meta-level @2 shows the model of a specific type of network, an *Electricity Supply Network*. As we can see in the figure, the edges need to store data regarding the voltage they transport (which can be low, medium, or high) and the safety distance they must keep to buildings or trees. For each node, we must know the input and output voltages (which can be different, as it happens in the case of transformation centers). Other attributes to store for each electric node can be the model or the producer.

The design we have presented is extended in the next level, in which we define specific classes for specific types of nodes of the network that instantiate some of the attributes. For example, we define a *TransformationCenter* which is a special type of node that transforms medium-voltage electricity into low-voltage electricity, and a class *ElectricalSubstation*, which transforms high-voltage electricity into medium-voltage electricity. In this case, these classes instantiate the attributes *voltageIn* and *voltageOut*, since it is not necessary to do it in the meta-level @0.

Figure 4.11 shows the overview of our multilevel modeling approach to the INSPIRE models. The topmost level (i.e. meta-level @6) is the level of ISO 19107 and the class *SpatialEntity* described in section 4.2.2. The meta-level @4 corresponds to the generic network model defined by INSPIRE. The meta-level @3 corresponds to the common data models defined in INSPIRE for transport networks and utility networks. Finally, the meta-level @2 corresponds to the data models defined by INSPIRE for road and railway networks in the data specification for transport networks, and electricity and water networks in the data specification for utility networks. The INSPIRE data specifications include additional data models for networks that we have not included here for the sake of clarity (e.g., water transport networks, or sewer networks). The meta-level @1 is also left undefined in this paper because it should be defined by each member country with its specific requirements.

Figure 4.12 shows the meta-level @4 of our multilevel metamodel for modeling spatial networks that replicates the INSPIRE Generic Network Model[11]. A *Network* is composed by *NetworkElement*s, which can be *Node*s and *Link*s (edges), both of them being *SpatialEntities* with geometries. A *NetworkElement* can also be a set of *Link*s (to represent collections of related edges) or a sequence of *Link*s (to represent ordered collections of related edges). *NetworkProperty* can be used to reference a collection of *NetworkElement*s to apply properties to sections of the network. The reference can be applied to the whole *NetworkElement* or a part of it using a point and an offset, or using an offset and a length (i.e., the traditional concept of linear referencing in GIS). This is a design that allows, for example, to specify in a road network that the maximum speed for the first half of a road link is different than the one for the second half).

Figure B.4 (in appendix B) shows the meta-level @3 model corresponding to the INSPIRE model for transport networks[12] (see fig. B.3 in appendix B), adapted to multilevel approach. As we can see, it redefines some meta-classes from the previous metamodel (fig. 4.12), adding the specific elements of the transport networks such as the *typeOfTransport*, an attribute that defines the network nature that is instantiated in the next meta-level (@2), or the properties *MaintenanceAuthority*, *TrafficFlowDirection* and *OwnerAuthority*.

The INSPIRE specifications define a more specific model for each kind of transport network (see figs. B.5 and B.7 in appendix B). We have represented them as the metamodels of the meta-level @2, shown in figs. B.6 and B.8 (in appendix B). These models instantiate meta-classes of meta-level @3 as concrete meta-classes of level @2 (e.g., a *Road* class, a *ERoad* class, or a *RailwayLine* as an instance of *TransportLinkSet*). They also add new attributes when necessary (e.g., the European route number of an *ERoad* or the railway line code of a *RailwayLine*).

We have followed a similar approach for the case of utility networks[13] (see fig. B.9 in appendix B). The model at meta-level @3 (fig. B.10, in appendix B) defines the common set of classes used in INSPIRE for utility networks, namely an *UtilityLinkSet* as an instantiation of a *LinkSet* of the meta-level @4, and classes to represent cables, pipes and ducts as specializations of an *UtilityLinkSet*. The model at meta-level @3 also defines a class to represent appurtenances of the utility network as an instance of the a *Node* from meta-level @4. Then, the model at meta-level @3 for water networks (fig. B.12, which replicates the INSPIRE data specification of fig. B.11, both in appendix B) instantiates the classes from meta-level @2 adding additional attributes to better describe the objects (e.g., the pipe diameter or the pressure). The same occurs with the meta-level @2 model for electricity networks (fig. B.14, which replicates the INSPIRE data specification of fig. B.13,both in

[11] `INSPIREDataSpecifications\T1\textendashBaseModels\T1\textendashGenericNetworkModel: https://inspire.ec.europa.eu/documents/inspire-data-specifications-%E2%80%93-base-models-%E2%80%93-generic-network-model`

[12] INSPIRE Data Specification on Transport Networks – Technical Guidelines: `https://inspire.ec.europa.eu/Themes/115/2892`

[13] INSPIRE Data Specification on Utility and Government Services – Technical Guidelines: `https://inspire.ec.europa.eu/Themes/136/2892`

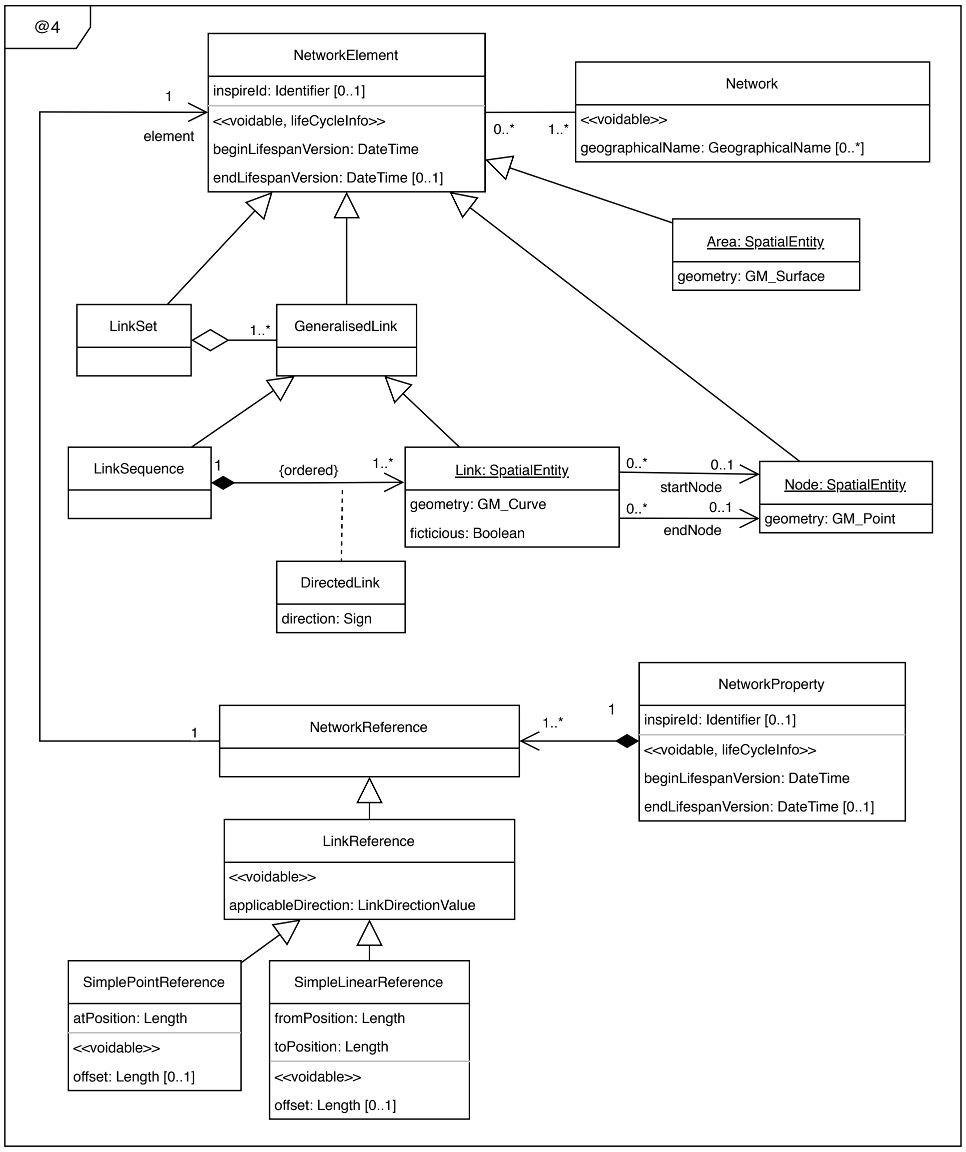


Figure 4.12: Modeling multilevel Spatial Networks - Meta-level @4

appendix B) that adds attributes to describe information such as the operating voltage or the nominal voltage of an electricity cable.

The proposal stops at meta-level @2 because it corresponds with the INSPIRE specification. A member country is expected to define a level @1 model that takes into consideration the specific requirements for its networks. Furthermore, if we take into account that the electricity networks of a country are built and operated by different companies, it would be possible to define an additional level below the INSPIRE level (that is, raising all the levels from fig. 4.11 one level up), adding a new meta-level @2 with elements like those used in fig. 5.2. This way, each company that operates an electricity network could define a meta-level @1 model compliant with @2 but taking into consideration the specific needs and functionalities of its information system.

4.2.4.3 Discussion

The advantages of multilevel modeling in this scenario are similar to those presented in section 4.2.2 and section 4.2.3: the multilevel model is simpler, it is more flexible, it ensures interoperability between INSPIRE member countries, and it can be easily extended to additional domains. Regarding the complexity of the models, the number of classes is similar because we have tried to replicate the models in the INSPIRE data specifications. However, the number of associations is much lower. In fact, the UML models in the INSPIRE data specification for transport networks uses non-standard UML notation to avoid cluttering the diagram with multiple inheritances (e.g., the class *TransportArea* inherits both from *NetworkArea* and *TransportObject*, but this is represented as a small italic text above the UML stereotype instead of displaying the parent classes and the associations).

The following common patterns of multilevel modeling appear in the example models for networks (see section 4.2.1):

- *Type-object* pattern: e.g., the different types that instantiate *Node* or *Edge* in lower meta-levels, such as *TransportNode* or *Appurtenance* (see figs. B.4 and B.10 in appendix B).
- *Dynamic features* pattern: e.g., the attributes added to meta-level @2 *ElectricalNode*, such as *model*, *voltageIn* or *voltageOut* (see fig. 5.2).
- *Dynamic auxiliary domain* pattern: e.g., in fig. 5.2, there is a new meta-class in meta-level @3, *Network*, and a new association between it and instances of *SpatialEntity*. More examples of this pattern appear in the more complex example shown in fig. 4.12, where instances of SpatialEntity have different relationships with domain-related elements such as *LinkSequence* or *Network*.
- *Element classification* pattern: e.g., the hierarchy descending from the meta-class *RailwayNode*, with its subclasses *RailwayYardNode* and *RailwayStationNode*, which are all of them instances of *TransportNode* from the superior meta-level (see figs. B.4 and B.8 in appendix B).

As a conclusion, the multilevel models are general enough to be used outside the INSPIRE domain and to be applied in any domain that needs the definition of spatial networks. Hence, the multilevel model can be considered and used as a common structure.

4.2.5 Using common structures in unrelated domains

4.2.5.1 Overview

In the previous Section, we have applied multilevel modeling in a domain that uses a common structure in the field of GIS. There are other application domains that are related but that are not usually modeled using the same common structure.

The INSPIRE data specifications provide an example for this scenario. INSPIRE defines some data specifications in the domain of facilities management (e.g., environmental management facilities, agricultural and aquaculture facilities, production and industrial facilities, and buildings). In this domain, it is very common to have a hierarchy of facilities that contain lower-level facilities. For example, a manufacturing plant is usually divided into different facilities. Each facility may consist of different buildings and installations, each one may be in turn divided into different parts. Another example may be an agricultural installation divided into different units with different objectives. Unlike the approach taken with networks, the authors of the INSPIRE data specifications did not consider to provide a generic model for hierarchies of facilities. Therefore, each data specification takes a different approach to model this hierarchy.

4.2.5.2 Multilevel modeling solution

Figure 4.13 presents a multilevel solution to the problem that allows us to define a hierarchy of facilities using a composite design pattern that is then instantiated in different models that are able to capture the specific semantics of the INSPIRE data specifications.

The meta-level @4 (fig. 4.13) of our solution defines a generic composite pattern to represent hierarchies of facilities using as base class of the composite the *ActivityComplex* class defined by INSPIRE as part of its base models[14] (see fig. B.15 in appendix B). This class includes an identifier, a geometry (which we include instantiating the *SpatialEntity* class form meta-level @6) and one or several thematic identifiers and functions. It also includes some voidable attributes related to life-cycle and validity information of the instances, which can be used at the object level (meta-level @0) for convenience if the actual application requires them. Then, we define a composite pattern of facilities inheriting from *ActivityComplex* that can be used by models in lower meta-levels to represent the hierarchy of facilities for each specific domain.

Figure B.17 (see appendix B) shows our solution to model environmental management facilities that are used to handle environmental material flows, such as waste or wastewater flows, which is based on the INSPIRE data specification on utility and government services[15] (see fig. B.16 in appendix B). The meta-class *EnvironmentalManagementFacility* instantiates *ComplexFacility* and redefines its relation since one of these facilities can manage a set of facilities itself. Some attributes of this domain are added, such as a *facilityDescription*, or the *serviceHours*.

[14]INSPIRE Data Specifications – Base Models – Activity Complex: `https://inspire.ec.europa.eu/documents/inspire-data-specifications-%E2%80%93-base-models-%E2%80%93-activity-complex`

[15]INSPIRE Data Specification on Utility and Government Services – Technical Guidelines: `https://inspire.ec.europa.eu/Themes/136/2892`

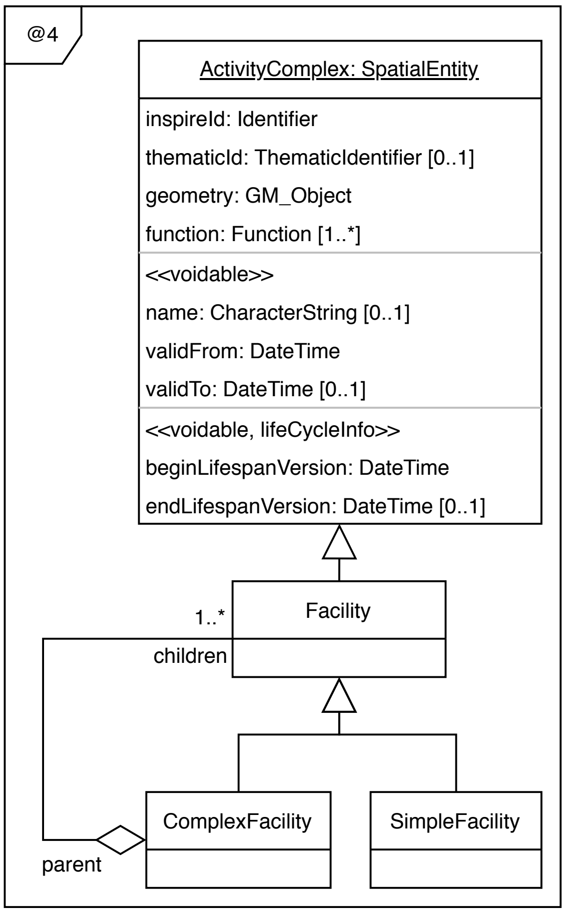

@4
ActivityComplex: SpatialEntity
inspireId: Identifier
thematicId: ThematicIdentifier [0..1]
geometry: GM_Object
function: Function [1..*]
<<voidable>>
name: CharacterString [0..1]
validFrom: DateTime
validTo: DateTime [0..1]
<<voidable, lifeCycleInfo>>
beginLifespanVersion: DateTime
endLifespanVersion: DateTime [0..1]
Facility
1..*
children
ComplexFacility
SimpleFacility
parent

Figure 4.13: Modeling multilevel Facilities Management - Meta-level @4.

In fig. B.19 (see appendix B) we have applied the pattern defined in fig. 4.13 to the specification of agricultural facilities[16] (see fig. B.18 in appendix B). The meta-class *Holding* represents the whole area and the infrastructures within it under the control of an operator to perform agricultural activities. Each *Holding* is composed by individual *Sites*. *Holding* instantiates the meta-class *ComplexFacility*, while *Site* instantiates *SimpleFacility*, being this a very straightforward example of the *ActivityComplex* composite.

A much more complex metamodel is the one for production facilities, as we can see in fig. B.21 (see appendix B). The data specification for this domain by INSPIRE[17] (see fig. B.20 in appendix B) defines a class *ProductionSite* representing the surface where one or several *ProductionFacilities* are located. Each *ProductionFacility* is composed itself by a set of *ProductionPlots* (land or water portions destined to functional purposes), *ProductionBuildings* (artificial constructions within the production facility), and *ProductionInstallations*. The latter are composed themselves by *ProductionInstallationParts*, which are single engineered facilities that perform specific functionalities.

The last example is related to the representation of buildings in INSPIRE. The INSPIRE specification for this domain[18] (see fig. B.22 in appendix B) is a bit more complex than the previous examples because it contemplates two different models to represent buildings: 2D or 3D. Therefore, the data specification defines in a first data model the generic attributes of constructions and buildings, which are then specialized in a 2D model by classes that represent buildings using 2D geometries and in a 3D model by classes that represent buildings using 3D geometries with different levels of detail. In our proposal (fig. B.23, see appendix B), we have defined a model at meta-level @3 to represent the generic attributes of constructions (i.e., class *AbstractConstruction*) and buildings (i.e., class *AbstractBuilding*). Then, we have represented the composition of building parts into buildings instantiating the classes *SimpleFacility* and *ComplexFacility* from meta-level@4 respectively. Our proposal for the 2D version of the data specification is shown in fig. B.24 (see appendix B). The class *Building* is instantiated into a class *Building2D* and the class *BuildingPart* is instantiated into a class *BuildingPart2D*. Both are associated with a class *BuildingGeometry2D*.

4.2.5.3 Discussion

The main advantage of our solution is that we apply a well-known design pattern in meta-level @4 (i.e., a composite pattern, see fig. 4.13) that is instantiated in the models of four different application domains. The use of the design pattern allows software engineers to take advantage of all its advantages (e.g., reusability and flexibility), while multilevel modeling allows software engineers to express the specific restrictions of each application domain.

The authors of the INSPIRE data specifications (almost) applied the composite pattern in the definition of the data specification for administrative units (fig. 4.5). Hence, the model in the data specification is flexible enough to accommodate the specific administrative divisions of all member countries, but using a two-level solution prevents each member

[16]INSPIRE Data Specification on Agricultural and Aquaculture Facilities – Technical Guidelines: `https://inspire.ec.europa.eu/Themes/137/2892`

[17]INSPIRE Data Specification on Production and Industrial Facilities – Technical Guidelines: `https://inspire.ec.europa.eu/Themes/121/2892`

[18]INSPIRE Data Specification on Buildings – Technical Guidelines: `https://inspire.ec.europa.eu/Themes/126/2892`

country from defining specific restrictions while conforming to the model in the INSPIRE data specification. However, the authors of the INSPIRE data specifications related to the management of facilities decided not to use a design pattern because they considered that representing the precise semantics of each domain was more important than using common and well-known structures. This decision makes the models more difficult to understand.

Our solution has the advantages of both alternatives. On the one hand, the model at meta-level @4 uses a composite pattern that provides flexibility and reusability. On the other hand, each of the models of each domain captures the semantics of the domain, in particular:

- The meta-level @2 for environmental management facilities does not define any specific restrictions on the composition, just like the data specification does.
- The meta-level @2 for agricultural facilities restricts the composition to two levels (i.e., holdings and sites) retaining the semantics of the data specification.
- The meta-level @2 for production facilities defines a hierarchy (i.e., nested containment) with four levels, just like the data specification does.
- Our proposal for buildings shows two advantages. First, the composite pattern of meta-level @4 is reused while keeping the semantics of the INSPIRE data specification. Second, while the INSPIRE data specification must use a UML constraint to represent that the part of a 2D building has to be a 2D building part (because the inheritance hierarchy would allow any building part to be part of any building), our proposal explicitly represents the constraint by redefining the association in meta-level @2.

Regarding common patterns of multilevel modeling (see section 4.2.1), the following appear in the examples of this section:

- *Type-object* pattern: e.g., the different types that instantiate *ComplexFacility* or *SimpleFacility* in fig. B.21 (see appendix B).
- *Dynamic features* pattern: besides the attributes added in the meta-class *Activity-Complex*, that instantiates *SpatialEntity* (see fig. 4.13), we have new attributes for example in *EnvironmentalManagementFacility* with respect to *ComplexFacility* (see fig. B.17 in appendix B).
- *Dynamic auxiliary domain* pattern: e.g., new associations are created between instances of *Building* and *BuildingPart*, and a new meta-class that instantiates *SpatialEntity* (see fig. B.24 in appendix B).
- *Relation configurator* pattern: e.g., in fig. B.17 (see appendix B), the association between

  Facility and its subclass *ComplexFacility* is redefined since *EnvironmentalManage-mentFacility* can be composed of instances of the same class.

The number of classes in these modes is similar to the models in the INSPIRE data specifications because we have tried to replicate them. Regarding the number of associations, our model does not require inheritance associations to *ActivityComplex*, thus reducing the complexity. The number of associations between the classes is not reduced in our models, but considering that we have included a composite design pattern, we can say that we have improved reusability and flexibility without adding complexity.

4.3 Discussion and evaluation

In this section, we evaluate the proposal presented in section 4.2 with respect to the requirements we described in the introduction of this chapter for the modeling solution: (R1) Scope, (R2) Reuse of common structures, (R3) Flexibility, (R4) Realistic, and (R5) Generality. Also, we summarize the multilevel metamodeling patterns that occur in the solution and the advantages that we have identified. Finally, we summarize the advantages of the multilevel modeling solutions with respect to their two-level counterparts.

Scope (R1), flexibility (R3) and generality (R5) As we explained in previous sections, working with one metamodel level requires balancing the trade-off between the scope of the metamodel and its flexibility. Adding common elements and structures to the metamodel would enforce all the models to use those elements and common structures as they were defined and updating them would require updating the metamodel.

As an example, in the case of networks with a spatial component, in a two-level metamodeling solution, all our models would have to conform to the network definition at the metamodel level and adding new features or admitting potential modifications on that network definition would force us to redefine the metamodel and, probably, add unnecessary complexity to it. Trying to consider all those particularities in one meta-model would be far from flexible. Moreover, the adaptations needed for two different applications could be contradictory, which would pose a problem to the usability of the metamodel. This is the reason why metamodels such as the one presented in [CLP+17, CLPP17, ACL+20] leaves out of the metamodel some elements of the domain that are interesting.

Something similar happens in the case of territory administration. In the metamodel level, we may define that a hierarchical decomposition of the territory follows a composite pattern. While this is the general case, the specific legal context of two countries may specify a more specific structure. For example, the territory can be structured in autonomous communities, provinces, and municipalities in Spain, but it may follow a slightly different schema in other countries. Trying to reflect all those specificities in the same metamodel would lead to an artificially complex solution. We can find similar problems in the scenario of facility management.

A multilevel modeling solution allows us to define common elements that may be necessary for different applications while having the flexibility and generality to adapt them to the particularities of each project in other metamodel levels.

The solution presented in section 4.2 defines at each level elements at a specific level of abstraction that can be refined or adapted in metamodels at lower levels. For example, in the case of networks we were able to first define a generic network structure, then refine it to the case of transportation networks, and then adapt it again to the particular case of road and railway networks. In territory administration, we considered a scope that can be adapted to the particular case of any country, and in the case of facilities management, we were able to consider different types of facilities (environmental, agricultural, industrial, and buildings). Therefore, the solution based on multilevel modeling allowed us to address a wider scope. In the multilevel solution, extending the scope of the metamodel does not necessarily imply less flexibility. Therefore, in the multilevel solution, the flexibility and the generality of the metamodel is not determined by the scope of the metamodel (R1, R3 and R5).

Scope (R1), reuse of common structures (R2), and generality (R5) The reuse of common structures may not be relevant in other application domains, but it is especially convenient in GIS. The common information structures we have modeled emerge naturally from the information managed in these systems and its organization in the real world. In the four scenarios we considered in our proposal, we were able to identify those information structures and to model them in a general way that was then adapted to the particularities of specific cases. More specifically, our solution covers the following common structures of the GIS domain:

- *ISO conceptual model and OGC implementation model:* we propose in our solution to apply multilevel modeling to the definition of international standards in GIS. This is a very ambitious proposal since ISO and OGC have proposed 81 and 70 standards each (although not all are related to each other, and there is overlap between the standards). However, it cannot be denied that carrying out this task would meet the requirements of scope, reuse of common structures and generality.
- *Territory administration:* our solution models a general decomposition of the territory for administration purposes (just like the INSPIRE solution does) but allows us to specialize it to the particular territory administration of different countries (in our case, Spain and Portugal).
- *Networks:* from an abstract definition of a network, our solution considers the cases of transportation networks (in our case, road and railway networks), and also utility networks (in our case, electricity and water pipes networks). The solution could be easily extended to consider other types of networks.
- *Facilities:* as in the previous examples, an abstract facility information structure was then refined for managing environmental, agricultural, industrial, and building facilities.

Scope (R1) and Realistic (R4) We consider the solution we have presented realistic (R4) and covers a wide scope (R1) based on existing proposals for GIS that are currently being used in the industry. One of our goals in this work was to create a model solution according to real existing proposals for GIS modeling. Thus, we considered the INSPIRE Directive, a very complete set of models that cover the most typical application areas of GIS. INSPIRE defines 34 data specifications, and our goal was not to model INSPIRE completely. Therefore, we focused on selecting representative examples that show in a real example the benefits of a solution with multiple meta-modeling levels.

As we have seen in section 4.1, the INSPIRE data definitions support a large set of concepts and many of the relationships among them. However, the models of INSPIRE can be quite complex. The multilevel models we presented in section 4.2 provide a much simpler solution, with each level concerning one level of abstraction that can be easily adapted to specific needs of applications at lower levels.

Considering the purpose of INSPIRE and that it is thought to be used in different countries, one of the benefits we appreciate in a multilevel modeling solution in GIS is that it allows us to instantiate attributes, operations, and relationships at different levels. This is particularly important in complex model structures such as the one proposed by INSPIRE.

One of the advantages of multilevel modeling is the capability of deferring instantiation. This is very adequate to our context since certain elements of the models need to be defined depending on the particularities of the application context. For GIS applications based

Patterns	GIS Scenarios		
	Administration	Networks	Facilities
Type-object	x	x	
Dynamic features	x	x	x
Dynamic auxiliary domain		x	x
Relation configurator	x	x	x
Element classification		x	

Table 4.1: Occurrence of multilevel metamodeling patterns in the scenarios.

on INSPIRE specifications, the application context depends not only on the requirements of a specific application but also on the country. With our multilevel approach, the particularities of the country can be expressed in a meta-level lower than INSPIRE, but there is still a place for expressing the actual application requirements in the data model of the application, in meta-level @1.

Multilevel metamodeling patterns We analyzed the presence of the different multilevel metamodeling patterns described in [dLGC14]: type-object, dynamic features, dynamic auxiliary domain, relation configurator, and element classification. Table 4.1 shows the patterns that occur in each of the scenarios, we are using all the patterns at some point of the solution, and all the scenarios use at least two multilevel metamodeling patterns.

Summary of the advantages

The advantages of multilevel modeling in the GIS domain can be summarized as follows:

- Many organizations are defining models for many aspects of GIS that are strongly related between them. However, these models are completely independent and they are disconnected. If a multilevel modeling approach were used, the models would be formally connected enabling many advantages related to MDE (e.g., applying model transformation techniques).
- Multilevel modeling reduces inheritance in the models. This improves the readability of the models and reduces the probability of requiring multiple inheritance and hence avoiding its problems.
- Interoperability can be improved using multilevel modeling because the organizations that define standards can propose metamodels that can later be instantiated in a more precise metamodel at a lower-level standard organization. The INSPIRE directive is a paradigmatic example in which the European Union would define a metamodel for each of the themes of the European spatial data infrastructure, that would, in turn, be instantiated in a new metamodel in each of the member countries, that would finally be instantiated in an application schema (or even in a new metamodel at a lower-level administrative division of the member country).

- Well-known common structures of GIS applications could be proposed as metamodels that could later be instantiated in the metamodel of specific GIS development tools to be finally instantiated in specific application schemas.
- Software engineering design patterns (e.g., the composite or the strategy design patterns) could be applied to application domains that are currently unrelated to transfer the benefits of the design patterns to the application domain. This would add flexibility to the application domains and it would reduce the complexity of the models and the learning curve of engineers and developers, facilitating the use of standard models instead of ad-hoc solutions for each problem.
- Some modeling constraints can be modeled as first-class entities in multilevel modeling instead of being external expressions in a different language or simple annotations.

4.4 Conclusions

Geographic information systems manage entities with a geospatial component that plays a central role in the system. Even if two GIS applications have different functional scopes, they will share a set of common concepts, data types for representing geometries, spatial structures (such as territory decompositions or spatial networks), and a set of technologies based on international standards published by different organizations, such as ISO or OGC. These characteristics make GIS a suitable application domain for MDE.

In this work, we have addressed the modeling of GIS following a multilevel modeling approach. More specifically, we have presented a multilevel modeling solution for GIS considering different scenarios: harmonization of basic conceptual and implementation models from ISO and OGC, territory administration, spatial networks, and facilities management. These scenarios have been selected from the European Union's INSPIRE Directive, which defines a set of models for information regarding resource and environmental management so that EU member countries can follow them to ensure interoperability. In each of these scenarios, we have shown how typical elements and structures present in many GIS applications can be modeled in abstract levels to be refined and instantiated at lower levels. We have used these examples to show how multilevel modeling can be applied to bridge the gap between conceptual and implementation standards in GIS, ensuring interoperability in spatial data infrastructures, modeling common GIS patterns, and applying common structures to unrelated domains.

Based on that previous experience and the analysis presented in the discussion sections of this work, and although the set of models presented in this research does not pretend to be exhaustive and is just a part of all the potential elements included in a GIS, we consider that it shows that the application of multilevel modeling in this domain can lead to simpler, more flexible, and more expressive solutions when compared with a two-level approach. The multilevel solution allows us to define metamodels with a larger scope and richness that can be later adapted. It also allows us to model common information structures that appear repeatedly in GIS in a way that allows us to redefine or adapt them to the particular needs of an application. The models defined by INSPIRE provide a solution following a traditional modeling approach, but extending and adapting those models to the particular needs of each country or organization would not always be possible, or it would imply extending inheritance hierarchies that are already very complex. Also, the solution we have presented is based on existing standard models for GIS, which shows that multilevel modeling can lead to a good solution in a realistic view of the domain of geographic information systems. Furthermore, by using INSPIRE models to create a multi-level model for GIS, we have shown that international standards for information systems are a promising application domain for multilevel modeling approaches.

The scope of this work does not include aspects related to model transformation, code generation, or other implementation aspects, which remain as future work.

Chapter 5

Mutation Testing for Geographic Information Systems

In this chapter, we address the mutation-based testing of *geographic information systems* (GIS), by proposing a set of mutation operators that address the typical errors that can be made with the technologies used to develop these systems. The main feature of GIS is that they manage entities with a spatial component, typically represented as a point, line, polygon, or as a variant of these basic geometries. As we will see in Section 2.1.1, the processes for capturing, storing, processing, and visualizing these entities are based on very specific technologies. If we want to apply mutation-based testing to GIS in an effective way, we must be able to inject in the SUT meaningful errors related to the specific technologies used in GIS development. Thus, in this chapter, we present a set of mutation operators that reproduce common programming errors in the development of Geographic Information Systems (GIS).

To obtain the set of operators that we present, we analyzed the most used free software technologies in the development of GIS, and for each of the layers and technologies operators were defined that reproduce errors that we consider likely during programming. Also, we present how these operators have been implemented using aspect-oriented programming, and an experimental evaluation where they were tested on real applications: a desktop GIS software for land management, and an Android mobile application for configuring touristic routes from a list of points of interest.

The rest of this chapter is organized as follows: In Section 3.1, we describe the work related to this area. In Sections 2.1.1 and 5.2, we analyze the most used technologies in the development of GIS and present a set of mutation operators that respond to errors that may occur during the development of this type of applications. Section 5.3 describes the process by which the operators are applied to the SUT to generate the mutants. In Section 5.4, we present a case study that describes how these operators are implemented and validates their use by reproducing these errors in real GIS. Finally, in Section 5.5 we present our conclusions.

5.1 Fundamentals concepts

5.1.1 Mutation

Mutation Testing is an error-based testing technique. It consists of making changes to a program at the source or bytecode level[1] and involves the construction of test data designed to discover these changes [Woo93].

This technique has been used to validate sets of test cases. Your goal is to find errors in a SUT in such a way that if a set of test cases does not find them, it is probably poorly designed.

These errors are artificially injected into the SUT by mutation operators and are called mutations. Mutations are a copy of the original program under test that has had semantic or syntactic changes made without preventing the program from compiling correctly, thus generating a faulty version of the original program.

Table 5.1 shows as an example a function that performs an arithmetic operation on two numbers and three mutants generated for said operation. These mutants arise from the syntactic modification of the arithmetic operator addition (+) by the applicable equivalent subtraction (-), multiplication (*) and division (/) operators.

Function version	**Implementation**
Original	int sum(int a, int b){ return a+b; }
Mutant 1	int sum(int a, int b){ return a-b; }
Mutant 2	int sum(int a, int b){ return a*b; }
Mutant 3	int sum(int a, int b){ return a/b; }

Table 5.1: Example of a function and its mutated versions

5.1.2 Mutation Operator

The generation of the mutants from the original program is carried out by applying mutation operators. Each operator introduces a certain type of error into the SUT, these failures must be replicas of common errors that programmers may commit unintentionally. There are types of operators classified according to the contexts for which they were implemented:

- Traditional: They apply to practically any programming language.
- Dependent on the Paradigm: They are oriented to different programming paradigms, such as Object-Oriented Programming.
- Language-specific: They are defined for a particular programming language.

[1]Intermediate code between the source code and the machine code. It is generated from the compilation process and is usually treated as a binary file that contains an executable program. Its name originates from the fact that each opcode is generally one byte in length.

- Technology Oriented: They are specifically designed for a specific technology.

Tables 5.2 and 5.3 describe some of the Traditional mutation operators specific to the Object Oriented Paradigm (OOP).

Mutation Operator	**Description**
ABS (absolute value)	Substitution of a variable by its absolute value
ACR (array reference for constant replacement)	Replacing a variable reference to an array by a constant
AOR (arithmetic operator replacement)	Substitution of an arithmetic operator
CRP (constant replacement)	Substituting the value of a constant
ROR (relational operator replacement)	Substitution of a relational operator
RSR (return statement replacement)	Return statement replacement
SDL (statement deletion)	Deleting a sentence
UOI (unary operator insertion)	Unary operator insert

Table 5.2: Traditional Mutation Operators

5.1.3 Aspect-Oriented Programming

Aspect-Oriented Programming (POA) is a paradigm that attempts to formalize and represent the transversal elements to the entire system, calling them Aspects. It emerged from research conducted at the Xerox PARC Research Center during the 1980s and 1990s [KLM$^+$97].

A **Appearance** is a piece of code that implements a common interest and modifies the behaviour of those areas of the code related to the purpose of the appearance. These areas of the code whose behaviour is captured are called **joinpoints** (or endpoint). The code that implements the change that must be executed when a joinpoint is captured is **advice**. The advice code can be executed before (**before**), after (**after**) or instead of the (**around**) joinpoint code. These modifiers and others make up the **pointcut** (or cut-off point) and indicate when during the joinpoint execution the advice code should be executed. Through these elements, it is possible to alter the behaviour of the original code without modifying it. It also allows you to specify with a fine level of granularity the context of the application behaviour alteration.

In POA, it is possible to define joinpoints on the execution that you want to capture in a program, these represent the places where the aspects add their behaviour. Table 5.4 lists the keywords to indicate the type of capture that the pointcut must do followed by the identification information of the method, constructor or another element to capture.

The pointcut can also include operators, wildcards, and other keywords to specify exactly which joinpoints should be trapped.

Operator	**Description**
AMC (access modifier change)	Access modifier replacement
AOC (argument order change)	Changing the order of arguments passed in a method call
CRT (compatible reference type replacement)	Replacing a reference to an instance of a class with a reference to an instance of a compatible class
EHC (exception handling change)	Changing an exception handling statement to a statement that propagates the exception, and vice versa
EHR (exception handling removal)	Removing a Exception Handling Statement
HFA (hiding field variable addition)	Adding in the subclass a variable with the same name as a variable of its superclass
MIR (method invocation replacement)	Replacing a call to a method with a call to another version of the same method
OMR (overriding method removal)	Removal in subclass redefining of a method defined in a superclass
POC (parameter order change)	Changing the order of parameters in a method declaration
SMC (static modifier change)	Adding or removing the static modifier

Table 5.3: Mutation Operators for OOP

- **Operators:**
  - ! negation.
  - || OR logic.
  - && AND logic.
  - () parenthesis.
- **Wildcard characters:**
  - * asterisk, to refer to anything.
  - .. colon, to replace a single item..
  - . a period, to refer to sub-packages.
  - + plus sign, to refer to subtypes.
- **Keywords:**

Element to capture	**Keyword**
Method calls	call(method signature)
Method execution	execution(method signature)
Constructor calls	call new(constructor signature)
Initializer execution	initialization(constructor signature)
Constructor execution	execution new(constructor signature)
Static initializer execution	static initialization(type name)
Field reading	get(field signature)
Assignment of value to field	set(field signature)
Execution of the exception handler	handler(exception to catch)

Table 5.4: Keywords used in pointcut capture types

- *target* refers to the object on which the trapped code will be executed: that is, the object that executes the joinpoint.
- *args* refers to the arguments of the trapped method.

Aspect-Oriented Programming has broad expressiveness for writing specific mutation operators. For example, it is possible to express situations like the following:

Before calling any method within the calculate method, whose names begin with subtraction or addition, that the first parameter is of type double, and that returns a double, the divide operation is executed.

```
public privileged aspect AspectArithmeticOperation
{
    double before(ArithmeticOperation operation, double x, double y):
    withincode (void ArithmeticOperation.calculate() )

    call (double operation.subtract*(double, *)) || call (double operation.sum*(double, *))
    && target(operation)
    && args(x, y)
    {
        operation.divide;
    }
}
```

Listing 5.1: Example of expressiveness aspect

Interlacing an aspect file with a program is done through the process of **weaving** or **interwoven**. This process is responsible for adequately combining the original program code, that is, the joinpoints, with the code that implements the change to be introduced, in this case, the advice so that, when the system is up and running, the advice and joinpoint run together. Interweaving can be static or dynamic, static interweaving handles those elements that can be determined before the program begins its execution, usually at compile-time, while dynamic interweaving occurs when the SUT is at execution time.

To make use of aspect-oriented programming it is necessary to:

1. **Base language:** A general-purpose language where basic functionalities are defined.
2. **Aspect-oriented language:** A language for programming aspects.
3. **A weaver:** A program in charge of weaving that combines the program and the aspects.

5.1.4 Mutation Operators with Aspects

Rather than manually manipulating the source code of a program to modify its behaviour to reproduce bugs, with POA this can be accomplished without directly altering the original source code. For this, through different elements, an aspect is written that intercepts the original program, causing it to be modified. This abstraction in which the writing of a program behaviour change is specified, simulating an error, represents a mutation operator [Pol14].

Suppose we have the ArithmeticOperation Class 5.2, which contains a set of methods for basic arithmetic operations. We want to apply the AOC (Argument Order Change) mutation operator to the divide method, which exchanges the parameter values so that if divide (10, 2) is written, divide (2, 10) is executed. You want to modify the execution of the original program so that it behaves as if it were a programming error where the position of the two arguments had been exchanged. In this case, the *around* clause is used, which replaces the body of a method with the code included in the *advice*. Fragment 5.3 shows the code of the aspect that contains the *pointcut* that catches the call and the code of the *advice* that performs the modification to this method of the Arithmetic Operation class. In the *advice* 3 instructions have been added for illustrative purposes to carry out the exchange of the parameter values using a temporary variable and then a call to the function *proceed (m, x, y) has been included)*, which produces a call to the original method.

```
package Aspects;
public class ArithmeticOperation {

    public double sum(double x, double y) {
        return x+y;
    }
    public double subtract(double x, double y) {
        return x-y;
    }
    public double multiply(double x, double y) {
        return x*y;
    }
    public double divide(double x, double y) {
        return x/y;
    }
}
```

Listing 5.2: Arithmetic operation class

```
public privileged aspect AspectArithmeticOperation
{
    double around(ArithmeticOperation operation, double x, double y):
```

```
    call (double operation.divide(double, double))
        && target(operation)
        && args(x, y)
    {
        double aux=x;
                x=y;
                y=aux;
                return proceed(operation, x, y);
    }
}
```

Listing 5.3: An aspect that implements the AOC operator to intercept the divide method of the Arithmetic Operation class

Observe the code of the aspect presented in Fragment 5.3:

- In line 3 the clause *around* replaces the body of the method with the code included in the advice.
- The arguments passed are: (1) the type of the object on which the operation is executed, in this case the object *operation* of the class *Arithmetic Operation* and (2) the types and names of the parameters of the method to be captured.
- After the colon symbol (:), line 4 indicates that the call will be caught with the keyword *call* to the operation indicated as an argument of the *call* itself; that is, the call to *doubleArithmeticOperation.divide (double, double)* will be trapped. After *call* on lines 5 and 6 we identify the role of the elements that we pass as arguments to the *around*: the *target* is the object on which the operation is executed, and the arguments they are x and y.
- Starting from line 7 in the advice, write the code to be executed when the operation *divide (..)* is called, which in this case will be to exchange the values of the method's parameters.

When interweaving the original program 5.2 with the Aspect 5.3 a different result will be executed, for this case, instead of dividing the values (x, y) in that order, the aspect will interchange them, introducing in its execution an error that does not exist in the original class.

5.2 A collection of mutation operators for GIS

In order to identify a set of potential mutation operators for GIS, we systematically analyzed the full stack of technologies we could use on its development. There are many approaches to GIS development. In our work, we have focused on open-source technologies based on the standards published by the Open Geospatial Consortium[2] and ISO. In this Section, we based the typical architecture that GIS application usually present and the most widely used open-source technologies for develop them. These technologies were analyzed to identify possible errors that may occur during the process of implementing a GIS.

In this section we present a collection of mutation operators for GIS based on the analysis of the technologies we have described in the Section 2.1.1 , and on our experience

[2]http://www.ogc.com

using them in real projects. Each mutation operator tries to reproduce an error that we consider a developer could easily make in a real project. We tried to come up with a collection as complete as possible. However, it is possible that other researchers consider that other errors can be frequent and are not included in our collection. As we will see in the next section (which focus on the implementation of the operators), the collection could be easily extended.

From the analysis we have done on the different layers and technologies that support the typical architecture of GIS applications, in this Section we present a set of mutation operators that has been defined to reproduce errors that the programmer can make in the different layers of a GIS during their development.

5.2.1 Operators on connectivity between user interface and service layer

This sub-section presents mutation operators that affect the connectivity between the service layer and the view layer. These operators introduce errors in methods for obtaining geometries and maps from REST and WFS services, altering parameters on the calls to the services, causing service drops, or overly expensive queries that can cause a *timeout*.

5.2.1.1 ChangeCoordSys:

This operator exchanges the coordinate system of a geometry so that it does not match the coordinate system that it is using in the user interface. It simulates the error of not checking that the coordinate system is correct (to change it if necessary) when a GIS works with multiple data sources. The error is introduced by directly modifying the coordinate system of a geometry when recovering the wrapping of the figure (Listing 5.5).

```
Geometry geometry = feature.getGeometry();
```

Listing 5.4: Captured original code

```
public class ChangeCoordSys extends Operator {

    @Override
    public String getCode(String code) {
        code="pGeometry1.setSRID(-pGeometry1.getSRID());";
        return code;
    }
}
```

Listing 5.5: Operator ChangeCoordSys

5.2.1.2 ExpandVisualRange:

This operator allows to extend the extension of a polygon of Java Topology Suite that represents the range of visualization allowed in the user interface. Thus, the system may allow the user to enter values of geometries that are outside the originally allowed range. The error is entered directly by expanding the polygon. This operator captures the invocation to the method em GeometryFactory.toGeometry (envelope) and replaces it with em GeometryFactory.toGeometry (envelope.expandBy (3.0)) simulating that the user has selected geographical limits not allowed (Listing 5.7).

```
Polygon polygon = JTS.toGeometry(envelope);
```

Listing 5.6: Captured original code

```
public class ExpandVisualRange extends Operator {

    @Override
    public String getOperatorCode(String code) {
        code="args[0]=pEnvelope1.expandBy(3.0);";
        return code;
    }
}
```

Listing 5.7: Operator ExpandVisualRange

5.2.1.3 WMSDoesntRespond:

This operator simulates the situation in which a WMS map server does not respond. This situation is simulated by issuing an exception directly when it comes to accessing the service. This operator captures the invocation to the method *createGetMapRequest()*, and to simulate that the map server is down, the sentence is added: *throw new ServiceException("WMS does not respond")* that throws a service availability exception (Listing 5.9).

```
GetMapRequest request = wms.createGetMapRequest();
GetMapResponse response = (GetMapResponse) wms.issueRequest(request);
```

Listing 5.8: Captured original code

```
public class WMSDoesntRespond extends Operator {

    @Override
    public String getOperatorCode(String code) {
        code="throw new ServiceException(\"WMS does not respond\");";
        return code;
    }
}
```

Listing 5.9: Operator WMSDoesntRespond

5.2.1.4 CostlyWFS:

This operator introduces a waiting time that can produce a timeout when accessing a WFS service. Simulates the actual situation in which the spatial database, by its size, or by the type of query, may take time to respond. This operator captures the instantiation of an object of type *FeatureCollection* using the method *source.getFeatures(..)* and replaces it with the invocation to the method *GeomUtils.createHugeWFSResponse(..)* (Listing 5.11).

```
FeatureCollection<SimpleFeatureType, SimpleFeature> objeto_feature;
objeto_feature = source.getFeatures(param);
```

Listing 5.10: Captured original code

```
public class CostlyWFS extends Operator {

    @Override
    public String getOperatorCode(String code) {
        code=return GeomUtils.createHugeWFSResponse (param);
        return code;
    }
}
```

Listing 5.11: Operator CostlyWFS

5.2.2 Internal processing errors

Mutation operators that affect operations that are performed in the data processing layer. These operators introduce errors in the processing of geometries in the service layer, transforming geometries, coordinate systems, or disturbing the result of operations and topological constraints between geometries.

5.2.2.1 BooleanPolygonConstraint

This operator disturbs the result of topological constraint operations that can be performed between a polygon and another geometry (ie: *disjoint, touches, overlaps, within, contains, crosses, equals*). It captures the previous operations and alters the geometry that is passed as a parameter by adding to it a new point that passes through the centroid of the original geometry (Listing 5.13).

```
@PolygonAndPolygon
Method(Polygon x, Polygon y);
```

Listing 5.12: Captured original code

```
public class BooleanPolygonConstraint extends Operator {

    @Override
    public String getOperatorCode(String code) {
        code="com.vividsolutions.jts.geom.Coordinate[] coordinates =
      pGeometry1.getCoordinates();\n" +
        "\t\t\t\tcoordinates[0]=
        pGeometry1.getCentroid().getCoordinate();\n" +
        "\t\t\t\tcoordinates[coordinates.length-1]=
        pGeometry1.getCentroid().getCoordinate();\n" +
        "\t\t\t\tpGeometry1=
        new com.vividsolutions.jts.geom.
        GeometryFactory().createPolygon(coordinates);\n" +
        "\t\t\t\targs[0]=pGeometry1;";
        return code;
    }
}
```

Listing 5.13: Operator BooleanPolygonConstraint

5.2.2.2 RESTToGeometry:

This operator set to null a received geometry from a REST service when trying to convert to a geometry of the Geometry class. this operator identifies the line *Geometry geo = feature.getGeometry ()* and replaces it with *Geometry geo = null* (Fragments 5.14 and 5.15).

```
Geometry geo = feature.getGeometry();
```

Listing 5.14: Captured original code

```
public class RESTToGeometry extends Operator {

    @Override
    public String getOperatorCode(String code) {
        code="return null;";
        return code;
    }
}
```

Listing 5.15: Operator RESTToGeometry

5.2.3 Interaction with the spatial database:

Mutation operators that simulate errors that can occur when interacting with a spatial database, such as the use of operations that are not allowed in the current version of the database , violations of topological restrictions, system crashes or *timeouts* for operations that are too complex.

5.2.3.1 CantConnectPostgreSQL:

This operator simulates the situation in which it cannot connect to the server of spatial databases, throwing an exception when it comes to establishing the connection. This operator captures the invocation to the *DriverManager.getConnection (...)* method and inserts the throwing of an exception type *throw new SQLException ("Connection cannot be established")* just before being captured, conditioning to the connection never being established (Listing 5.17).

```
DriverManager.getConnection(arguments)
```

Listing 5.16: Captured original code

```
public class CantConnectPostgreSQL extends Operator {

    @Override
    public String getOperatorCode(String code) {
        code="throw new SQLException(Connection cannot be established)";
        return code;
    }
}
```

Listing 5.17: Operator CantConnectPostgreSQL

5.2.3.2 ForceQueryTimeout:

When sending a query to the spatial database, this operator introduces an excessive waiting time, which simulates a complex query that can force a timeout. The way in which the operator replicates this error is by capturing the query launch using the method *s.executeQuery (SQL)*, where "s" is an object of the class *Statement* or one of its subclasses. Once this method is captured, the operator adds the statement *s.setQueryTimeout(1)* after the execution of the captured method (Listing 5.19).

```
statement.executeQuery(query);
```

Listing 5.18: Captured original code

```
public class ForceQueryTimeout extends Operator {

    @Override
    public String getOperatorCode(String code) {
        code="java.sql.Statement statement=(java.sql.Statement) joinPoint;\n";
        code="statement.setQueryTimeout(1);\n";
        return code;
    }
}
```

Listing 5.19: Operator ForceQueryTimeout

5.3 Mutation operator generation process

To generate the mutation operators we define an approach, which we present in Figure 5.1. This approach is composed of two processes represented as gray rectangles: (1) formalization of operators from the identified errors an (2) creation of operators as aspects files. The files on which these two processes work or that are obtained as a result are represented as rectangles of white color. This approach has been implemented in a prototype, shown in Figure 5.2.

Formalization of mutation operators: The first step to apply the mutation is to define the mutation operators that you want to use to generate the mutants. For the creation of the operators, it will be based on the identification and definition of the errors

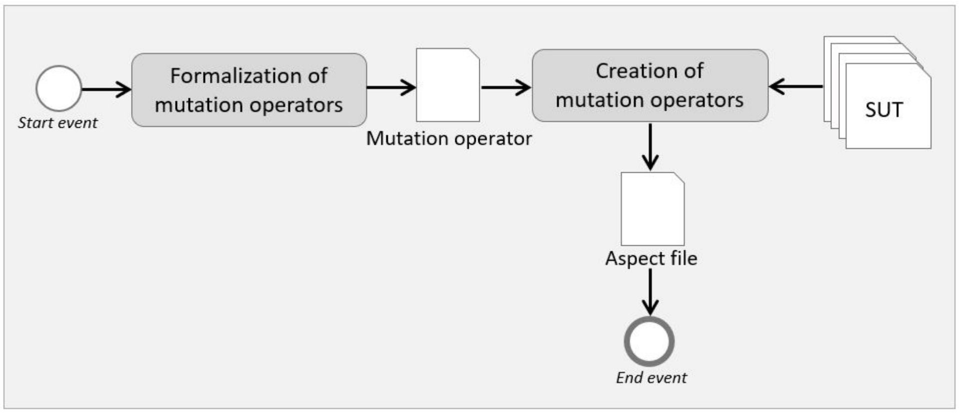
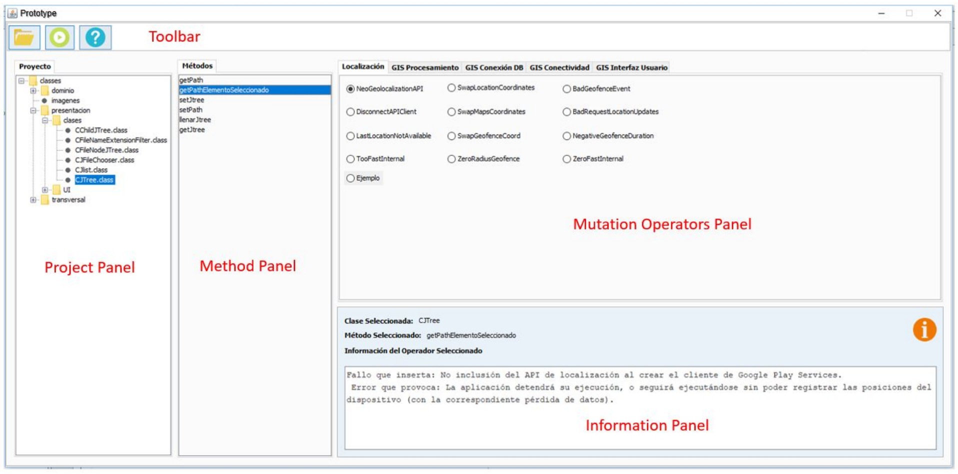

Start event
Formalization of mutation operators
Mutation operator
Creation of mutation operators
SUT
Aspect file
End event

Figure 5.1: Business Process Workflow, generation of mutation operators



[✓ NeoGeolocalizationAPI]

Clase Seleccionada: CJTree

Método Seleccionado: getPathElementoSeleccionado

Información del Operador Seleccionado

```
Fallo que inserta: No inclusión del API de localización al crear el cliente de Google Play Services.
 Error que provoca: La aplicación detendrá su ejecución, o seguirá ejecutándose sin poder registrar las posiciones del
dispositivo (con la correspondiente pérdida de datos).
```

Figure 5.2: Prototype Interface, generation of mutation operators

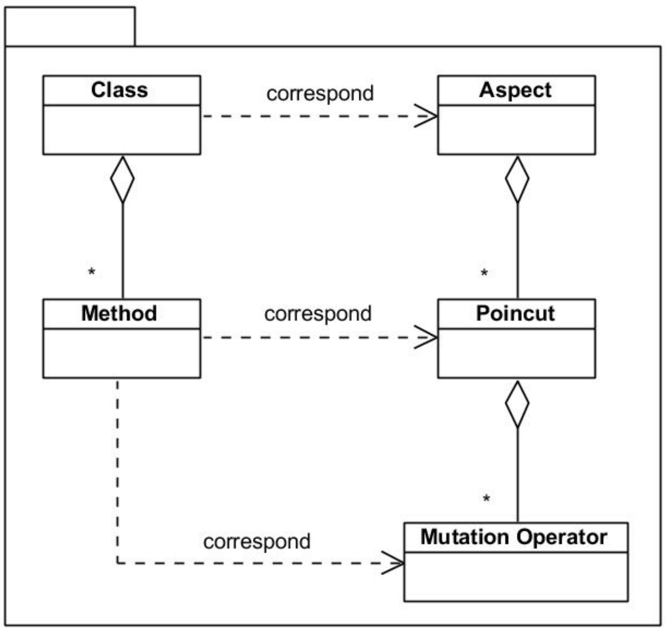


Figure 5.3: Aspect file structure

to subsequently define the elements that make up the operator based on this information. These operators will be used to generate the aspects file.

Creation of mutation operators: In this stage, the physical creation of the mutation operator is performed. The operator is physically represented by a file of aspects. This aspect file will contain as many poincuts as methods to be mutated (Figure 5.3) and will be generated from a template containing tokens or special marks. These tokens are replaced by code snippets that correspond to two elements: (1) the information of the methods of the SUT, captured by the Java *Reflection API* and (2) the definition of the mutation operators that have already been formalized in the previous stage. The file of aspects that is generated from this information (SUT source code and operators), is possible through the use of AspectJ libraries to interweave it with the original SUT and generate the mutated versions of the system.

The implementation of these operators is focused on the way in which the change is introduced in the SUT. The definition of the mutation operator will be used to create the physical operator as an aspect file, which will capture the call to a particular method of the SUT and alter its behavior. These operators are specializations of the abstract class Operator (Figure 5.4). Some of the methods more relevant that structure the Operator are:

- **getName**: Mutation operator name.
- **getDescription**:Description of the function performed by the mutation operator.
- **getCode**: Code that modifies the behavior of the SUT method to mutate.

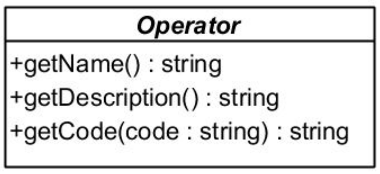


Figure 5.4: The Operator Class

The code fragments 5.20 and 5.21 show at a conceptual level the template to generate the aspect files and the Fragment 5.22 we show how this template is applied to an example of source code. The symbols between sharp (#) represent the tokens that must be replaced to generate the aspect. The following tokens are defined in the templates:

- IMPORTS, This token will be replaced by the corresponding *import* statements for the aspect to compile.
- CLASS_NAME, This token will be replaced by the class name for which the aspect is being created.
- POINTCUTS, This token will be replaced by the syntax of a *poincuts* AspectJ. A poincuts will allow you to capture the source code and modify its behavior.
- MOMENT, This token will be replaced by *before, after or around* as appropriate.
- CAPTURE_TYPE. This token will be replaced by one of the keywords *call or execution.*
- OBJECT_NAME. This token will be replaced by the name of the method to be captured.
- ARGUMENTS. This token will be replaced by arguments name of the method to be captured.
- PROCEED. This token will be replaced by the original code of the method on which the pointcut intervenes.
- OPERATORS. This token will be replaced by the error code that will modify the method.

```
#IMPORTS#

public aspect Operadores#CLASS_NAME# {
#POINTCUTS#
}
```

Listing 5.20: Aspect pointcut template

```
#RETURN_TYPE# #MOMENT# (#TARGET_TYPE# obj #PARAMETERS_TYPE_AND_NAME_LIST#) : target(obj)
     &&args(# PARAMETERS_LIST_NAMES_FOR_ARGS#) && #CAPTURE_TYPE# (#RETURN_TYPE#
     #TARGET_TYPE#.#OPERATION# (# PARAMETERS_LIST_TYPES#)){

int operadorActual = director.getCurrentOperator();
```

```
    if (operadorActual == ORIGINAL) {
         #PROCEED#
    }
    if (director.getCurrentJoinpoint()!=this)
        #PROCEED#
    else {
        #OPERADORES#
    }
}
```

Listing 5.21: Aspect file template

```
#RETURN_TYPE# #MOMENT# (#TARGET_TYPE# obj #PARAMETERS_TYPE_AND_NAME_LIST#) : target(obj) &&
     args(#PARAMETERS_LIST_NAMES_FOR_ARGS#) && #CAPTURE_TYPE# (#RETURN_TYPE#
     #TARGET_TYPE#.#OPERATION#(#PARAMETERS_LIST_TYPES#))
--------------------------------------------------------
void around(mypackage.MyClass object, double parameter) : target(object) && args(parameter) &&
     call(void mypackage.MyClass.MyMethod(double))
```

Listing 5.22: Header of poincuts, example of replacement of tokens

To generate the mutation operator, which is physically an aspect file, we have defined an algorithm, which we present as a pseudocode in the Algorithm 1.

```
for each mutable class $C$ do
    aspectsTemplate = read template aspects file;
    aspectCode = replaceTokens(aspectsTemplate, C);
    for each method $m \in C$ do
        pointcutTemplate = read template pointcut file;
        poincutCode = replaceTokens(pointcutTemplate, m);
        for each mutation operator op do
            if op is applicable to m then
                errorTemplate = read error template;
                operatorCode = replaceTokens(errorTemplate, op);
                poincutCode = poincutCode U operatorCode;
            end
        end
        aspectCode = aspectCode U poincutCode;
    end
    saveFile(aspectCode);
end
```

Algorithm 1: Pseudocode algorithm to generate the mutation operator.

5.4 Application examples

To test the mutation operators, two real-use GIS were used. These applications (Figures 5.5 and 5.7) were applied to different layers of their architectures by operators to simulate errors that could have occurred during their development process. Below is a case for each application that describes the errors that simulate the operators *BooleanPolygonConstraint* and *ChangeCoordSys*.

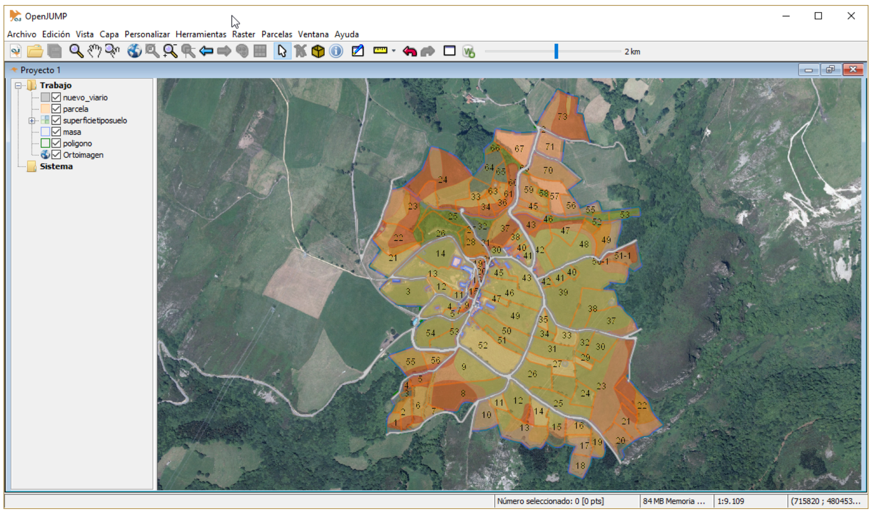


Figure 5.5: Land Reparcelling App

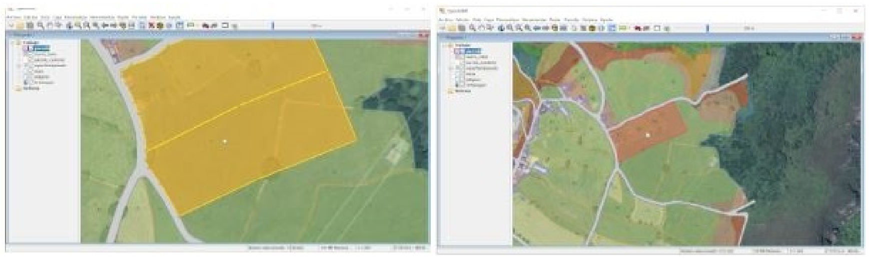


Figure 5.6: Original and mutant application.

5.4.1 Land Reparcelling App

This application is a simplified version of a Land reparcelling system. Land reparcelling processes are applied in regions where land ownership is very fragmented among different owners. The objective of the land consolidation is to reunify the lands of an owner to facilitate their exploitation. This application was developed using JTS tool for processing spatial objects, and PostgreSQL with PostGIS as a spatial database management system.

Figure 5.6 shows a screenshot of two selected adjacent parcels that meet conditions to be merged. In this case, the mutation operator *BooleanPolygonConstraint* was applied. This operator introduces errors in the processing of geometries in the service layer of the application, manipulating the result of operations that contemplate checking different topological constraints between geometries such as intersection or overlap. Specifically, the result of the merge operation between the two polygons has been affected. This error causes the user to incorrectly visualize the resulting geometry that should be drawn on the

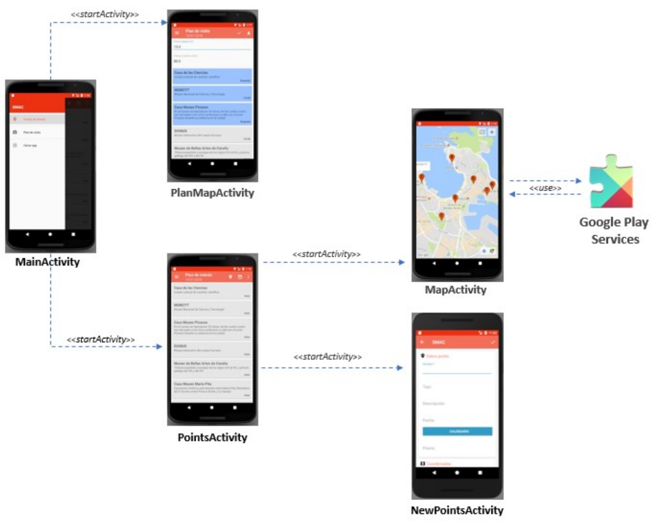


Figure 5.7: Interest Point App.

interface after the operation applied to the two initial geometries.

5.4.2 Interest Point App

This is a mobile technology GIS application that allows you to register places of interest for the user and track their entry/exit points. From this information, the application records the routes through which the user has traveled and generates a visit plan. For its operation, it uses Google's Localization Service. Figure 5.8 shows the different interaction screens (Activities) of this application.

A fundamental aspect in the development of mobile applications is the ability of the application to geolocate the user's position through the phone's resources. This technology allows applications to always know the user's position, adapting their behavior accordingly. Search sites Interest or nearby users are very common examples of the use of geolocation technology.

The Android operating system allows application developers to get notifications of when a user enters or leaves an area of interest. These areas of interest are called *Geofences*. A *Geofence* is determined by a geographical location expressed in terms of latitude and length, and a radius around said location. Thus, any application with the necessary permissions can request the operating system to notify the moment in which the person enters or leaves one of those areas of interest. These functionalities are mainly used for presence control. Although creating *Geofences* and receiving notifications is relatively simple, it is also a bug-prone code. By creating a *Geofence* with an erroneous location from its central location, the device will receive incorrect location notifications regarding the user's location and will move it to erroneous input/output zones. To simulate this error, the mutation

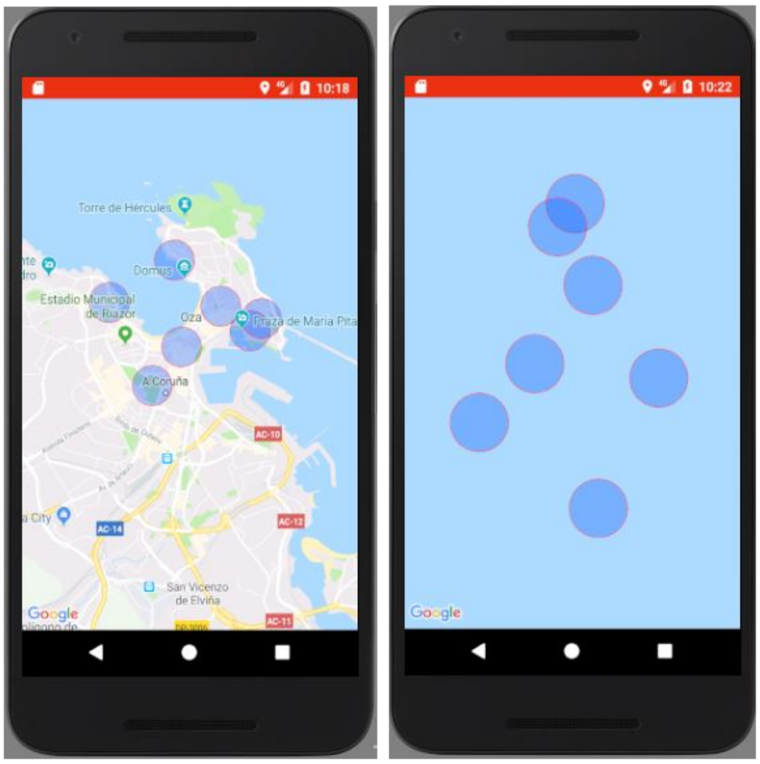


Figure 5.8: Original and mutant application.

operator *ChangeCoordSys* has been applied, which exchanges the location coordinates of the *Geofences*. As a result, the user will see the *Geofences* drawn in incorrect areas in the application's map viewer.

5.5 Conclusions

Based on a review of the state of the art on mutation operators, the existence of mutation operators both general purpose and specific to some technologies and languages has been verified, however, when simulating errors in particular domains is required, these operators are not adequate. We objective has been to define new mutation operators specific to the domain of GIS-based applications, which currently have no history in the scientific literature. We have also proposed a way to generate them and we have tested their applicability on two GIS systems of real use.

Chapter 6

Conclusions and Future Work

The implementation of GIS applications does not start strictly from scratch but is built in most cases applying techniques that cover in a sequential way the different stages of the traditional software development life cycle. There are various technologies and tools that support this process for all components of your architecture. Apart from these technologies, if it is required to develop applications for specific domains, it is necessary to write source code from scratch, which fits its functional scope. Although there can be great domain differences between these types of applications, there are a set of common elements that make them similar. In this research we have focused on these common characteristics, so the main contribution of this thesis focuses on the investigation of automated software development techniques applicable to the GIS domain.

We have researched about a definition of a declarative domain-specific language for the development of GIS. This language allows a user without deep programming knowledge can specify and generate a basic system, only specifying the system by defining its entities, with their properties and relationships, maps, and layers. We have developed a case study in which we use two sample projects of different sizes (application for the management of points of interest and application for the management of civil infrastructures) to evaluate the software products generated from the specifications in GIS- DSL. From our analysis we have concluded that the amount of LOC that is generated is high, therefore this gives us an idea of the rate of code generation that we can achieve, and the savings in terms of productivity that we could obtain in larger projects, with tens or hundreds of entities. As future work we propose to be carried out a more extensive evaluation, involving the generation of real applications.

On the other hand, we have research how to apply the multilevel modeling approach to GIS-based applications in different real-world scenarios. In each of these scenarios, we have shown how typical elements and structures present in many GIS applications can be modeled in abstract levels to be refined and instantiated at lower levels. We have identifies several benefits in terms of simplicity, expressiveness, and flexibility versus two-level modeling. As future work, it is proposed to include aspects related to the transformation of models, code generation or other implementation aspects.

Finally, our lasted contribution is on the application and automation of the mutation testing technique in the GIS domain. In the scientific literature, there are definitions of mutation operators both general purpose and specific for some technologies and languages, however, when it is required to simulate errors in particular domains, these operators are not suitable. We have defined a set of mutation operators specific to the domain of GIS-based applications. We have also proposed a workflow to generate it and generate the mutated system versions. Finally, we have tested their applicability on two GIS systems of real use. The approach we have proposed to specify and generate mutation operators is dependent on a language of aspects related to the language of the SUT. As future work, it is proposed to carry out a generic specification that can be independent of the technology.

Appendix A

Publications and other research results

Publications

Journals

- Alvarado, S. H., Cortiñas, A., Luaces, M. R., Pedreira, O., Places, A. S. (2022). Multilevel modeling of geographic information systems based on international standards. Software and Systems Modeling, 21(2), 623-666.
- Alvarado, S. H., Cortiñas, A., Luaces, M. R., Pedreira, O., Places, Á. S. (2020). Developing Web-based Geographic Information Systems with a DSL: Proposal and Case Study. J. Web Eng., 19(2), 167-194.

International conferences

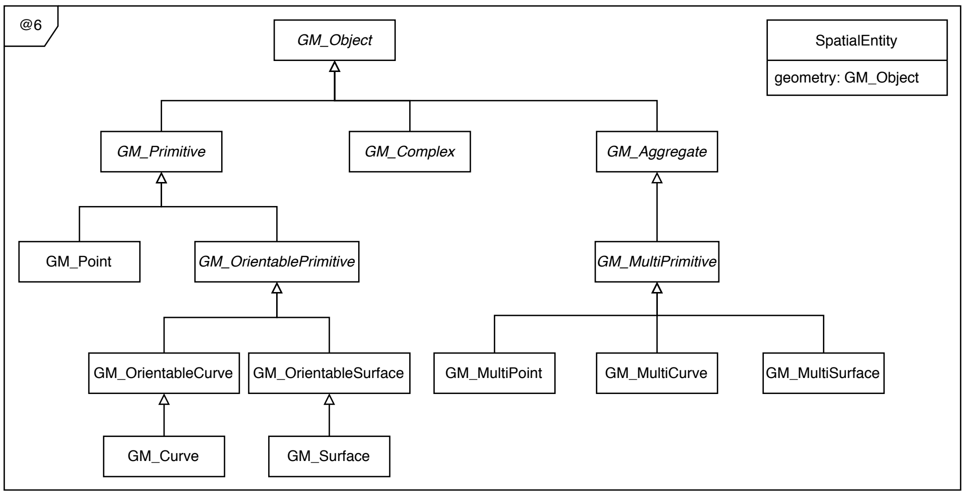
- Alvarado, S., Cortiñas, A., Luaces, M., Pedreira, O., Places, A. (2019, September). A Domain Specific Language for Web-based GIS. In Proceedings of the 15th International Conference on Web Information Systems and Technologies (pp. 462-469).
- Alvarado, S. H., Cortinas, A., Luaces, M. R., Pedreira, O., Places, A. S. (2019, September). Applying multilevel modeling to the development of geographic information systems. In 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C) (pp. 128-133). IEEE.
- Usaola, M. P., Rojas, G., Rodriguez, I., Hernandez, S. (2017, March). An architecture for the development of mutation operators. In 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) (pp. 143-148). IEEE.

National conferences

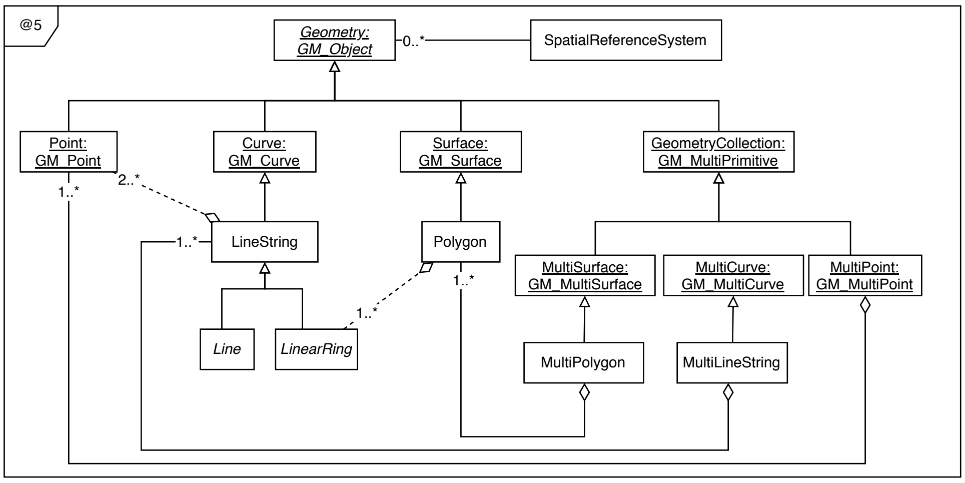
- Alvarado, S. H. (2019). Design of Mutation Operators for Testing Geographic Information Systems. Multidisciplinary Digital Publishing Institute Proceedings, 21(1), 43.
- Alvarado, S. H., de Guzmán, I. G. R., Luaces, M. R., Pedreira, O., Places, Á. S., Polo, M. Definición de Operadores de Mutación para Sistemas de Información Geográfica.

Appendix B

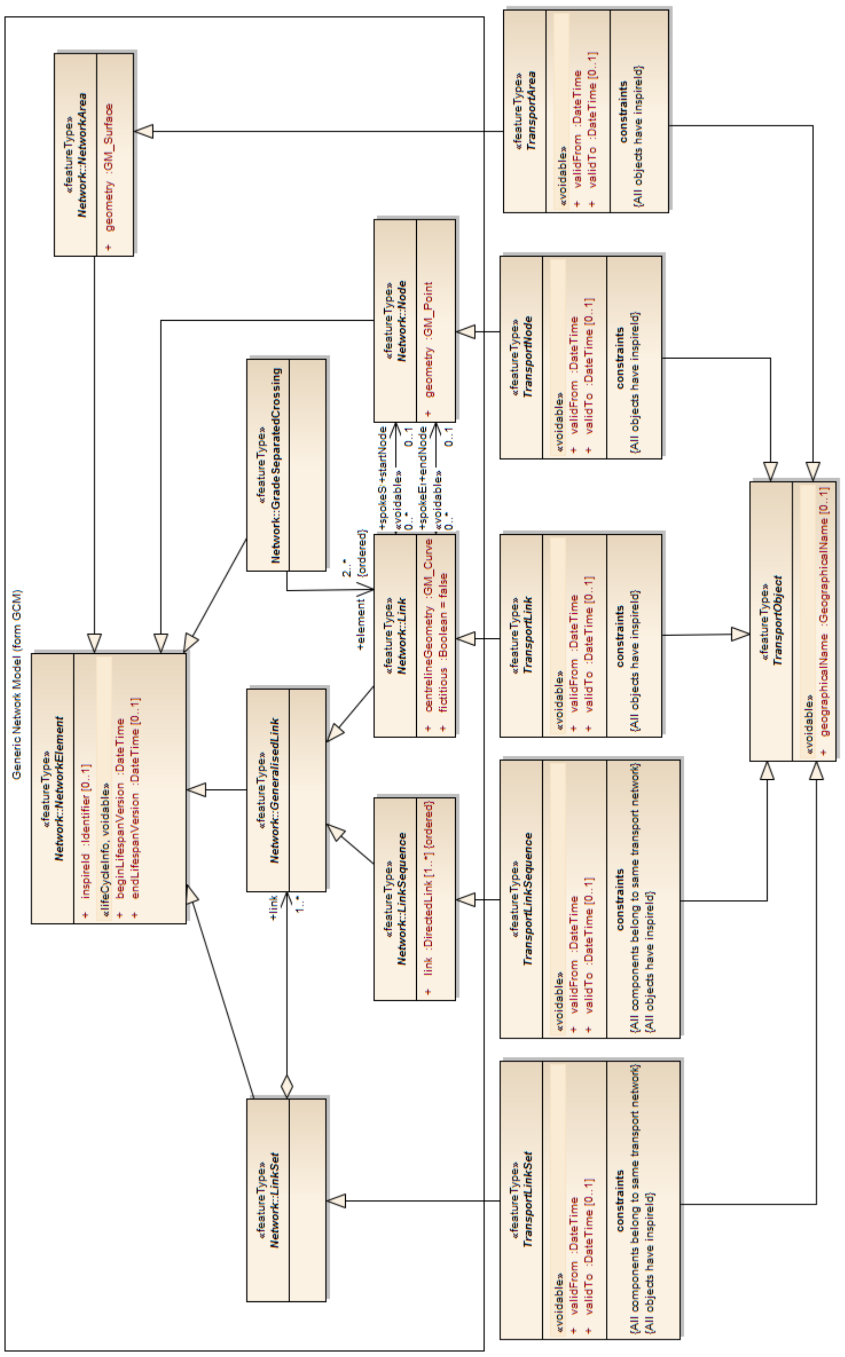
Complete models of Multilevel Modeling Chapter



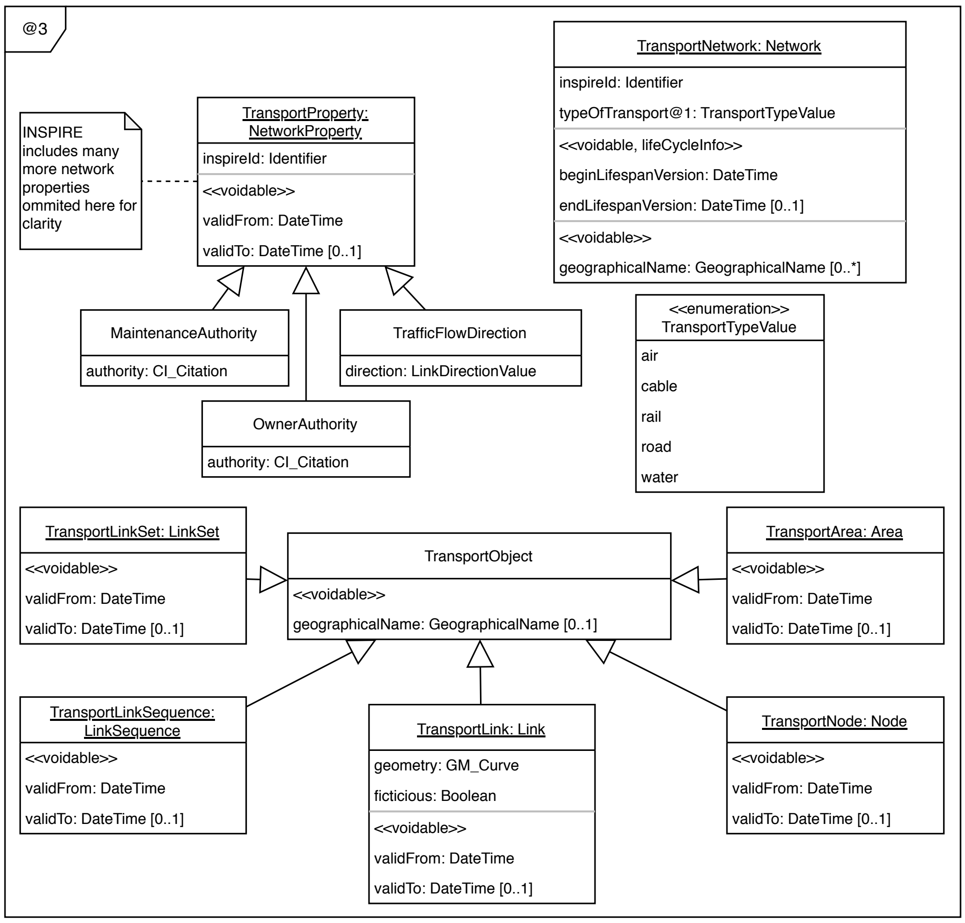
▲ **Figure B.1:** ISO 19107: Geographic Information - Spatial Schema



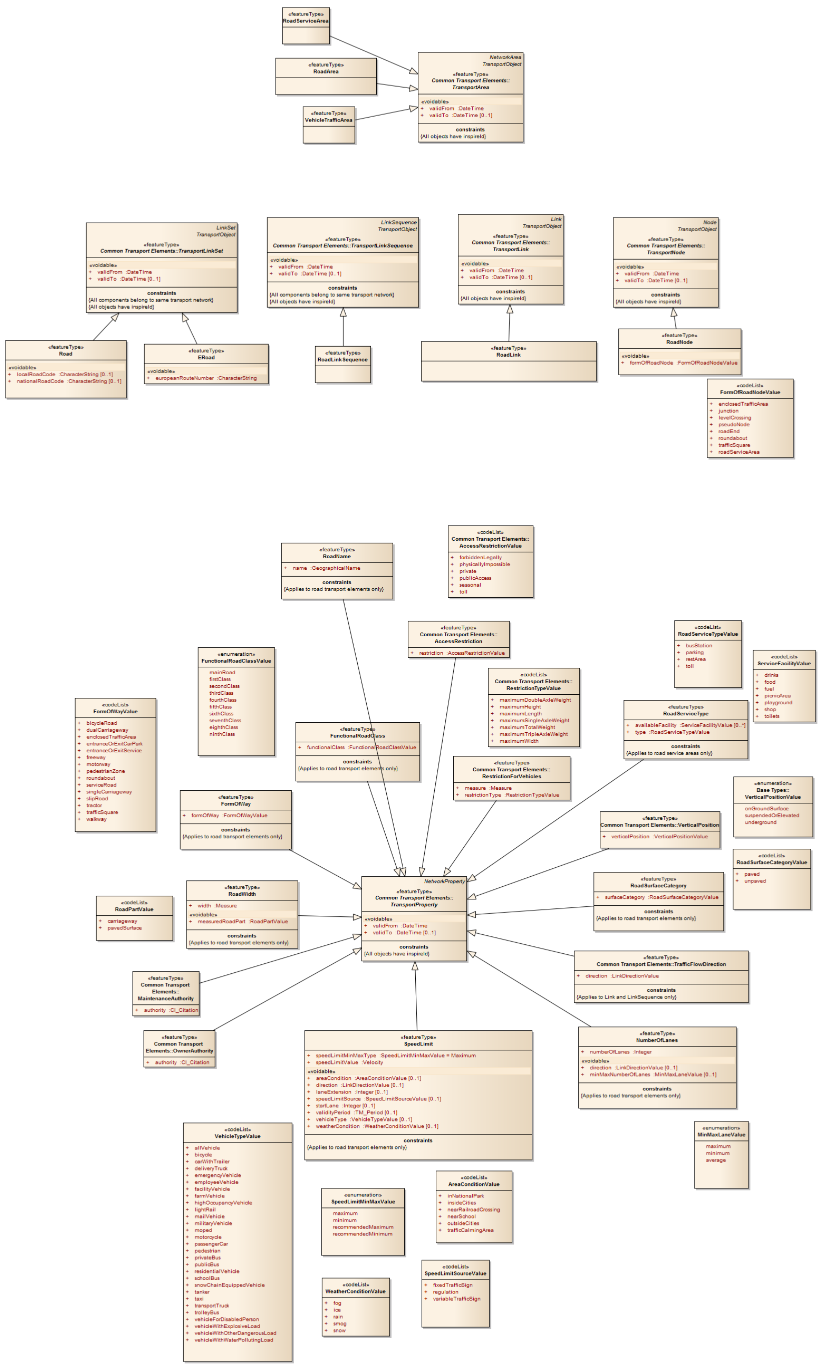
▲ **Figure B.2:** OGC Simple Feature Access (OGC SFA)


Generic Network Model (form GCM)
«featureType» Network::NetworkArea
+ geometry :GM_Surface
«featureType» Network::NetworkElement
+ inspireId :Identifier [0..1]
«lifeCycleInfo, voidable»
+ beginLifespanVersion :DateTime
+ endLifespanVersion :DateTime [0..1]
«featureType» Network::GradeSeparatedCrossing
«featureType» Network::GeneralisedLink
«featureType» Network::LinkSet
«featureType» Network::Node
+ geometry :GM_Point
«featureType» Network::Link
+ centrelineGeometry :GM_Curve
+ fictitious :Boolean = false
«featureType» Network::LinkSequence
+ link :DirectedLink [1..*] {ordered}
+spokeStart +startNode
+spokeEnd +endNode
«voidable»
0..*
0..1
+element
2..*
{ordered}
+link
1..*
«featureType» TransportArea
«featureType» TransportNode
«featureType» TransportLink
«featureType» TransportLinkSequence
«featureType» TransportLinkSet
«voidable»
+ validFrom :DateTime
+ validTo :DateTime [0..1]
constraints
{All objects have inspireId}
{All components belong to same transport network}
«featureType» TransportObject
+ geographicalName :GeographicalName [0..1]

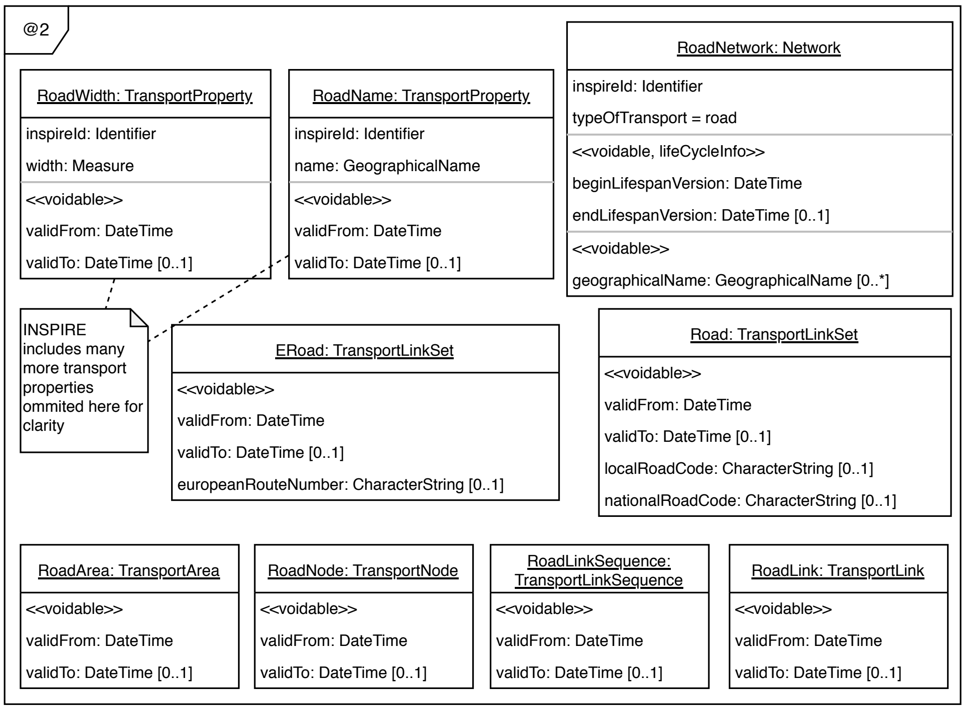
▲ **Figure B.3:** INSPIRE Network Base Model and Common Transport Elements Overview, from `https://inspire.ec.europa.eu/data-model/approved/r4618-ir/html/index.htm?goto=2:1:9:6:7590`



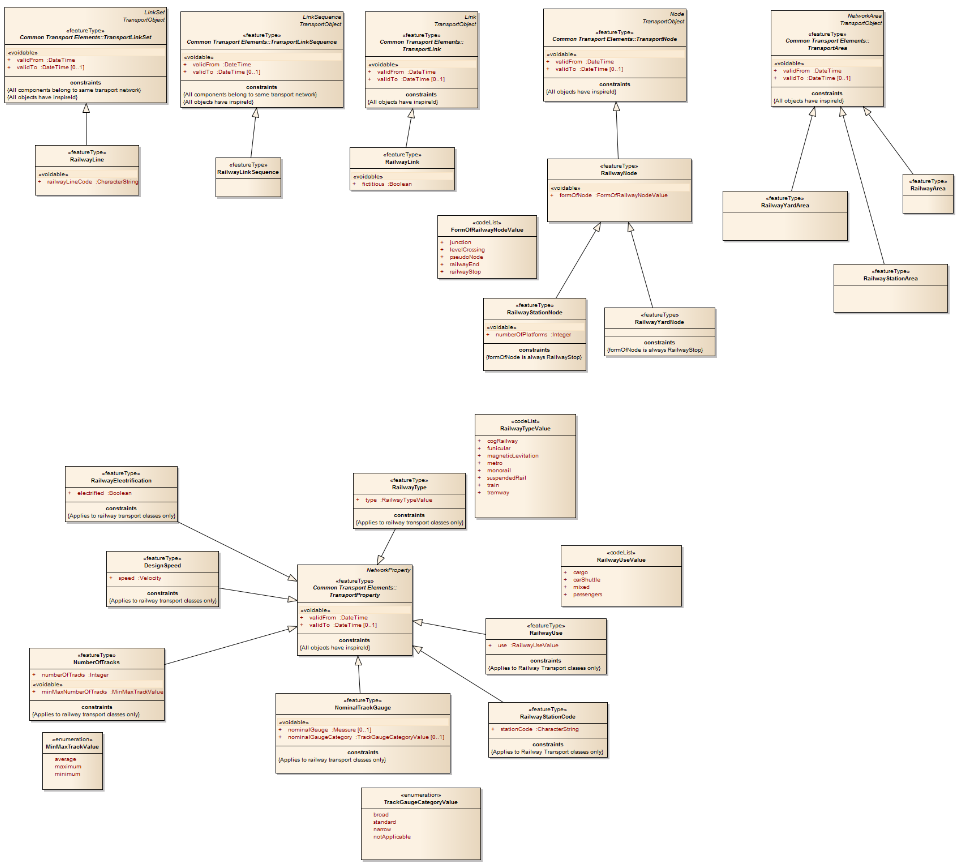
▲ **Figure B.4:** Modeling multilevel Spatial Networks - Meta-level @3 for transportation networks


RoadServiceArea
RoadArea
VehicleTrafficArea
Common Transport Elements::TransportArea
Common Transport Elements::TransportLinkSet
Common Transport Elements::TransportLinkSequence
Common Transport Elements::TransportLink
Common Transport Elements::TransportNode
Road
ERoad
RoadLinkSequence
RoadLink
RoadNode
FormOfRoadNodeValue
RoadName
Common Transport Elements::AccessRestrictionValue
Common Transport Elements::AccessRestriction
RoadServiceTypeValue
ServiceFacilityValue
FunctionalRoadClassValue
Common Transport Elements::RestrictionTypeValue
FormOfWayValue
FunctionalRoadClass
RoadServiceType
Common Transport Elements::RestrictionForVehicles
FormOfWay
Base Types::VerticalPositionValue
Common Transport Elements::VerticalPosition
RoadSurfaceCategoryValue
RoadPartValue
RoadWidth
Common Transport Elements::TransportProperty
RoadSurfaceCategory
Common Transport Elements::MaintenanceAuthority
Common Transport Elements::TrafficFlowDirection
Common Transport Elements::OwnerAuthority
SpeedLimit
NumberOfLanes
MinMaxLaneValue
VehicleTypeValue
AreaConditionValue
SpeedLimitMinMaxValue
SpeedLimitSourceValue
WeatherConditionValue

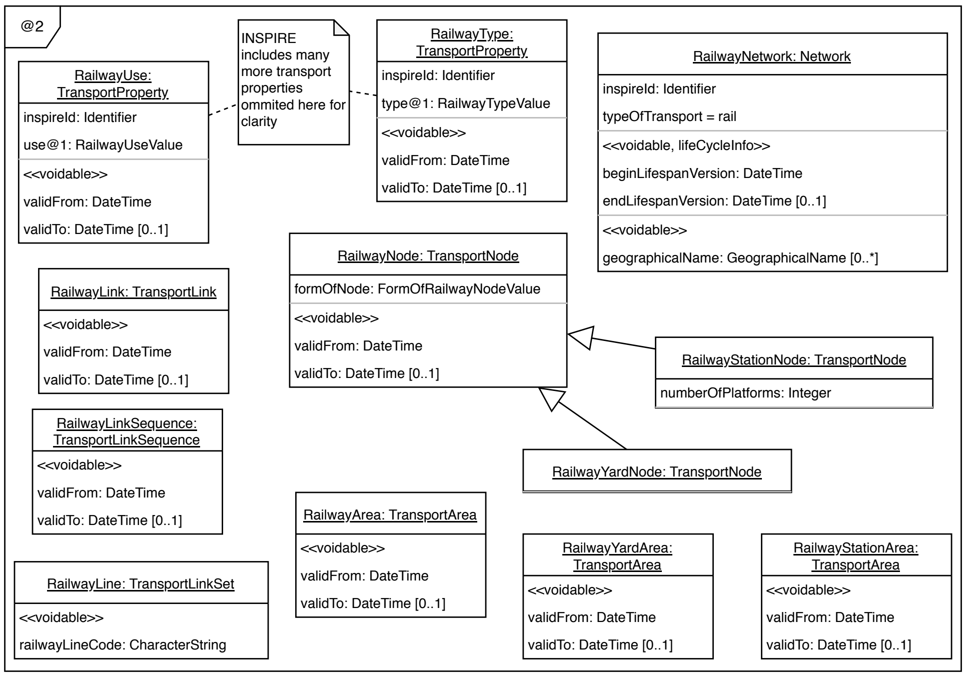
▲ **Figure B.5:** INSPIRE Road Transport Network, from `https://inspire.ec.europa.eu/data-model/approved/r4618-ir/html/index.htm?goto=2:1:9:7:7627`



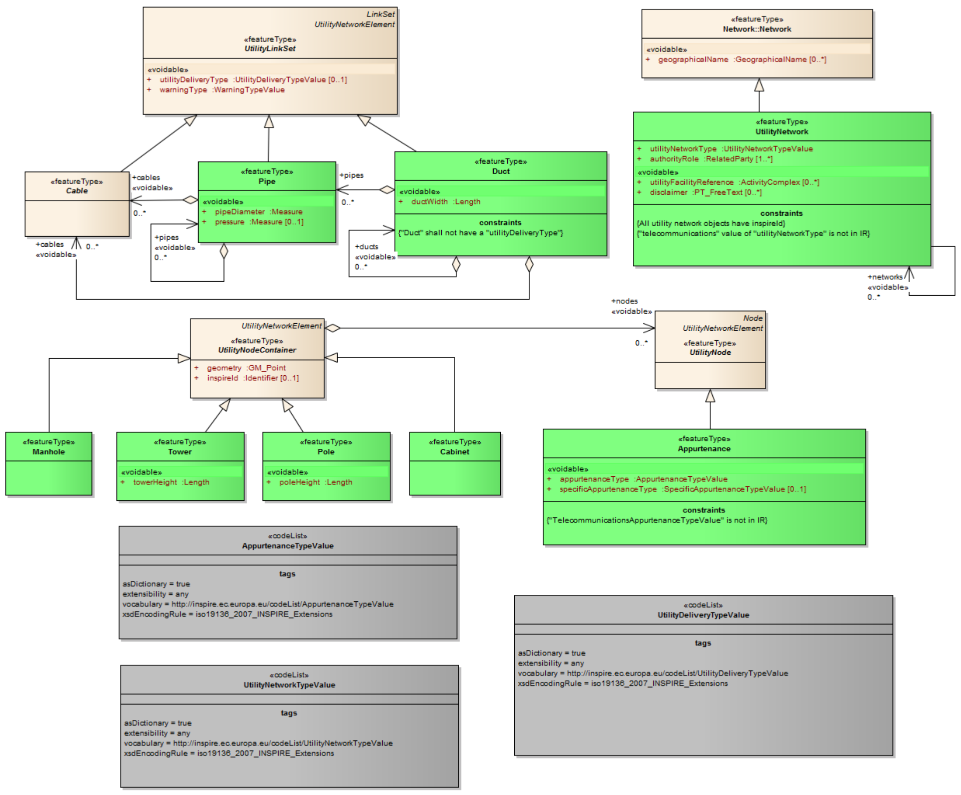
▲ **Figure B.6:** Modeling multilevel Spatial Networks - Meta-level @2 for roads networks



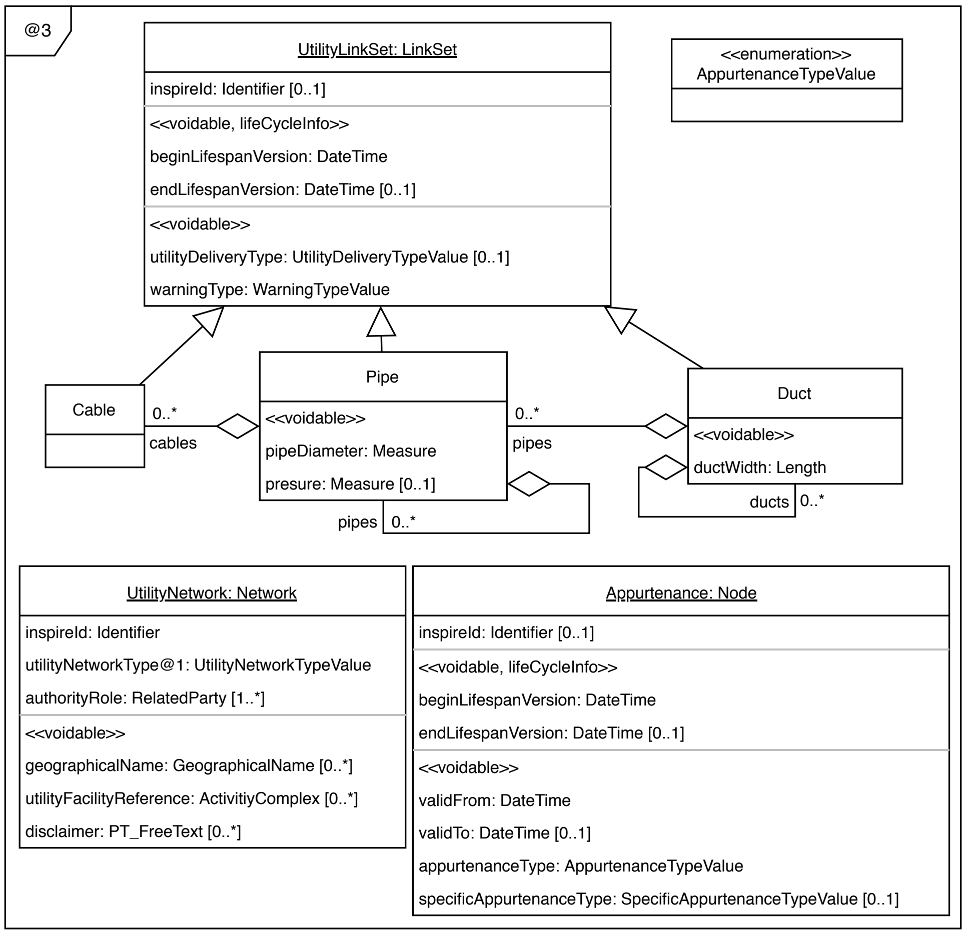
▲ **Figure B.7:** INSPIRE Railway Transport Network, from `https://inspire.ec.europa.eu/data-model/approved/r4618-ir/html/index.htm?goto=2:1:9:4:7508`


@2
INSPIRE includes many more transport properties ommited here for clarity
RailwayUse: TransportProperty
inspireId: Identifier
use@1: RailwayUseValue
<<voidable>>
validFrom: DateTime
validTo: DateTime [0..1]
RailwayType: TransportProperty
inspireId: Identifier
type@1: RailwayTypeValue
<<voidable>>
validFrom: DateTime
validTo: DateTime [0..1]
RailwayNetwork: Network
inspireId: Identifier
typeOfTransport = rail
<<voidable, lifeCycleInfo>>
beginLifespanVersion: DateTime
endLifespanVersion: DateTime [0..1]
<<voidable>>
geographicalName: GeographicalName [0..*]
RailwayNode: TransportNode
formOfNode: FormOfRailwayNodeValue
<<voidable>>
validFrom: DateTime
validTo: DateTime [0..1]
RailwayLink: TransportLink
<<voidable>>
validFrom: DateTime
validTo: DateTime [0..1]
RailwayStationNode: TransportNode
numberOfPlatforms: Integer
RailwayLinkSequence: TransportLinkSequence
<<voidable>>
validFrom: DateTime
validTo: DateTime [0..1]
RailwayYardNode: TransportNode
RailwayArea: TransportArea
<<voidable>>
validFrom: DateTime
validTo: DateTime [0..1]
RailwayLine: TransportLinkSet
<<voidable>>
railwayLineCode: CharacterString
RailwayYardArea: TransportArea
<<voidable>>
validFrom: DateTime
validTo: DateTime [0..1]
RailwayStationArea: TransportArea
<<voidable>>
validFrom: DateTime
validTo: DateTime [0..1]

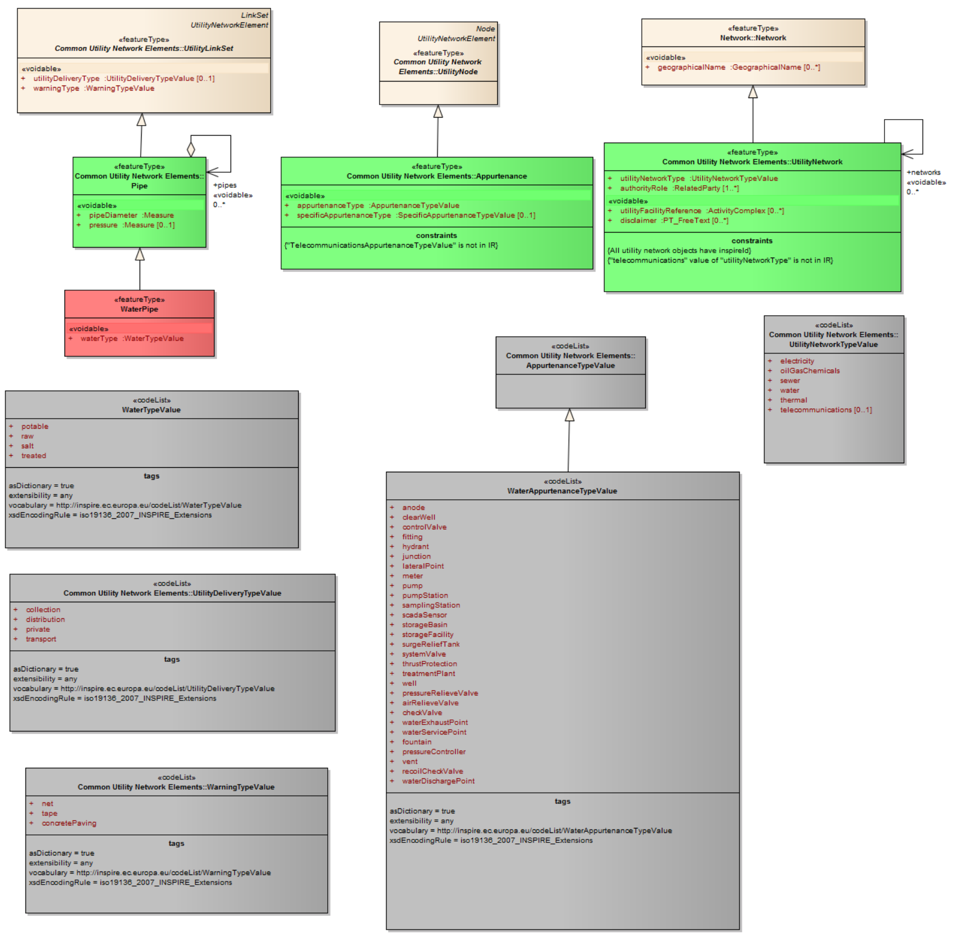
▲ **Figure B.8:** Modeling multilevel Spatial Networks - Meta-level @2 for railway networks


LinkSet
UtilityNetworkElement
«featureType»
UtilityLinkSet
«voidable»
+ utilityDeliveryType :UtilityDeliveryTypeValue [0..1]
+ warningType :WarningTypeValue
«featureType»
Network::Network
«voidable»
+ geographicalName :GeographicalName [0..*]
«featureType»
UtilityNetwork
+ utilityNetworkType :UtilityNetworkTypeValue
+ authorityRole :RelatedParty [1..*]
«voidable»
+ utilityFacilityReference :ActivityComplex [0..*]
+ disclaimer :PT_FreeText [0..*]
constraints
{All utility network objects have inspireId}
{"telecommunications" value of "utilityNetworkType" is not in IR}
+networks
«voidable»
0..*
«featureType»
Cable
+cables
«voidable»
0..*
+cables
«voidable»
0..*
«featureType»
Pipe
«voidable»
+ pipeDiameter :Measure
+ pressure :Measure [0..1]
+pipes
«voidable»
0..*
+pipes
0..*
«featureType»
Duct
«voidable»
+ ductWidth :Length
constraints
{"Duct" shall not have a "utilityDeliveryType"}
+ducts
«voidable»
0..*
UtilityNetworkElement
«featureType»
UtilityNodeContainer
+ geometry :GM_Point
+ inspireId :Identifier [0..1]
+nodes
«voidable»
0..*
Node
UtilityNetworkElement
«featureType»
UtilityNode
«featureType»
Manhole
«featureType»
Tower
«voidable»
+ towerHeight :Length
«featureType»
Pole
«voidable»
+ poleHeight :Length
«featureType»
Cabinet
«featureType»
Appurtenance
«voidable»
+ appurtenanceType :AppurtenanceTypeValue
+ specificAppurtenanceType :SpecificAppurtenanceTypeValue [0..1]
constraints
{"TelecommunicationsAppurtenanceTypeValue" is not in IR}
«codeList»
AppurtenanceTypeValue
tags
asDictionary = true
extensibility = any
vocabulary = http://inspire.ec.europa.eu/codeList/AppurtenanceTypeValue
xsdEncodingRule = iso19136_2007_INSPIRE_Extensions
«codeList»
UtilityDeliveryTypeValue
tags
asDictionary = true
extensibility = any
vocabulary = http://inspire.ec.europa.eu/codeList/UtilityDeliveryTypeValue
xsdEncodingRule = iso19136_2007_INSPIRE_Extensions
«codeList»
UtilityNetworkTypeValue
tags
asDictionary = true
extensibility = any
vocabulary = http://inspire.ec.europa.eu/codeList/UtilityNetworkTypeValue
xsdEncodingRule = iso19136_2007_INSPIRE_Extensions

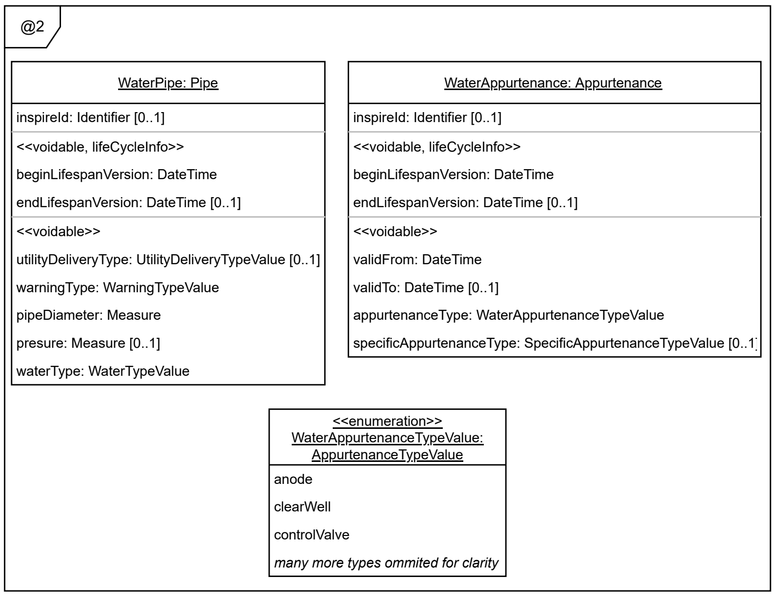
▲ **Figure B.9:** INSPIRE Common Utility Network Elements, from `https://inspire.ec.europa.eu/data-model/approved/r4618-ir/html/index.htm?goto=2:3:20:3:1:8887`


@3
UtilityLinkSet: LinkSet
inspireId: Identifier [0..1]
<<voidable, lifeCycleInfo>>
beginLifespanVersion: DateTime
endLifespanVersion: DateTime [0..1]
<<voidable>>
utilityDeliveryType: UtilityDeliveryTypeValue [0..1]
warningType: WarningTypeValue
<<enumeration>>
AppurtenanceTypeValue
Cable
Pipe
<<voidable>>
pipeDiameter: Measure
presure: Measure [0..1]
Duct
<<voidable>>
ductWidth: Length
0..*
cables
0..*
pipes
pipes
0..*
ducts
0..*
UtilityNetwork: Network
inspireId: Identifier
utilityNetworkType@1: UtilityNetworkTypeValue
authorityRole: RelatedParty [1..*]
<<voidable>>
geographicalName: GeographicalName [0..*]
utilityFacilityReference: ActivitiyComplex [0..*]
disclaimer: PT_FreeText [0..*]
Appurtenance: Node
inspireId: Identifier [0..1]
<<voidable, lifeCycleInfo>>
beginLifespanVersion: DateTime
endLifespanVersion: DateTime [0..1]
<<voidable>>
validFrom: DateTime
validTo: DateTime [0..1]
appurtenanceType: AppurtenanceTypeValue
specificAppurtenanceType: SpecificAppurtenanceTypeValue [0..1]

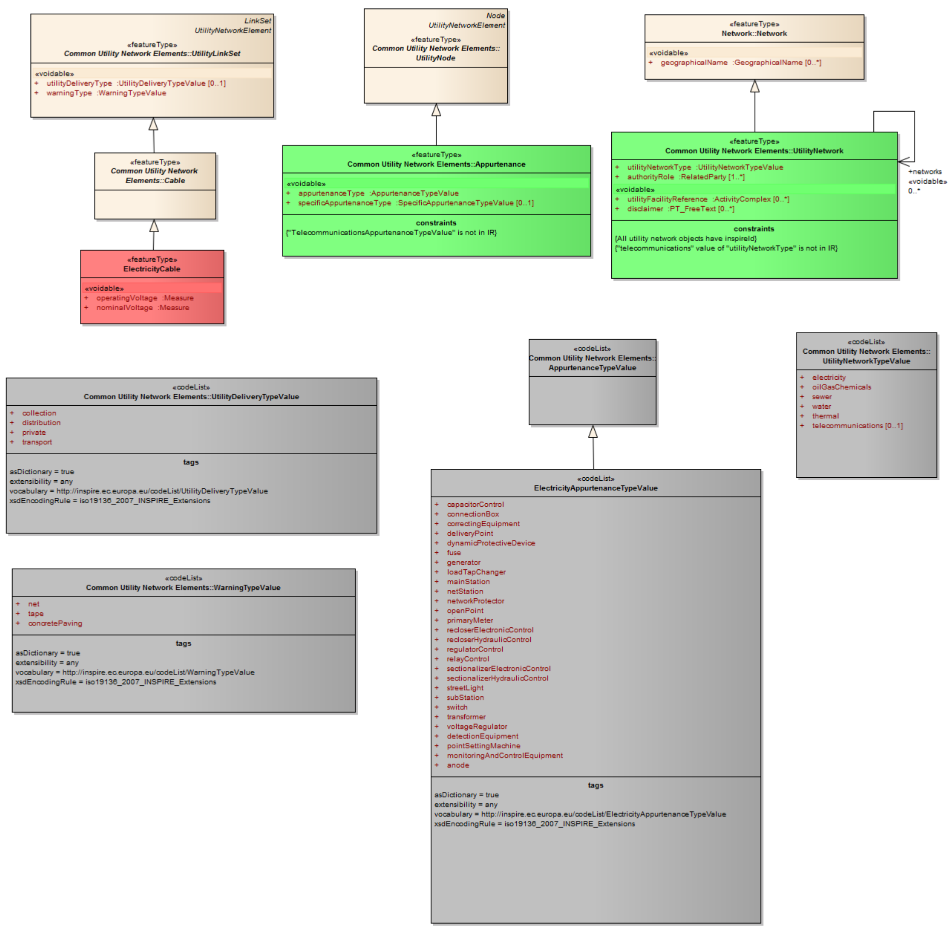
▲ **Figure B.10:** Modeling multilevel Spatial Networks - Meta-level @3 for utility networks


LinkSet
UtilityNetworkElement
«featureType»
Common Utility Network Elements::UtilityLinkSet
«voidable»
+ utilityDeliveryType :UtilityDeliveryTypeValue [0..1]
+ warningType :WarningTypeValue
Node
UtilityNetworkElement
«featureType»
Common Utility Network Elements::UtilityNode
«featureType»
Network::Network
«voidable»
+ geographicalName :GeographicalName [0..*]
«featureType»
Common Utility Network Elements::Pipe
«voidable»
+ pipeDiameter :Measure
+ pressure :Measure [0..1]
+pipes
«voidable»
0..*
«featureType»
Common Utility Network Elements::Appurtenance
«voidable»
+ appurtenanceType :AppurtenanceTypeValue
+ specificAppurtenanceType :SpecificAppurtenanceTypeValue [0..1]
constraints
{"TelecommunicationsAppurtenanceTypeValue" is not in IR}
«featureType»
Common Utility Network Elements::UtilityNetwork
+ utilityNetworkType :UtilityNetworkTypeValue
+ authorityRole :RelatedParty [1..*]
«voidable»
+ utilityFacilityReference :ActivityComplex [0..*]
+ disclaimer :PT_FreeText [0..*]
constraints
{All utility network objects have inspireId}
{"telecommunications" value of "utilityNetworkType" is not in IR}
+networks
«voidable»
0..*
«featureType»
WaterPipe
«voidable»
+ waterType :WaterTypeValue
«codeList»
Common Utility Network Elements::AppurtenanceTypeValue
«codeList»
Common Utility Network Elements::UtilityNetworkTypeValue
+ electricity
+ oilGasChemicals
+ sewer
+ water
+ thermal
+ telecommunications [0..1]
«codeList»
WaterTypeValue
+ potable
+ raw
+ salt
+ treated
tags
asDictionary = true
extensibility = any
vocabulary = http://inspire.ec.europa.eu/codeList/WaterTypeValue
xsdEncodingRule = iso19136_2007_INSPIRE_Extensions
«codeList»
WaterAppurtenanceTypeValue
+ anode
+ clearWell
+ controlValve
+ fitting
+ hydrant
+ junction
+ lateralPoint
+ meter
+ pump
+ pumpStation
+ samplingStation
+ scadaSensor
+ storageBasin
+ storageFacility
+ surgeReliefTank
+ systemValve
+ thrustProtection
+ treatmentPlant
+ well
+ pressureRelieveValve
+ airRelieveValve
+ checkValve
+ waterExhaustPoint
+ waterServicePoint
+ fountain
+ pressureController
+ vent
+ recoilCheckValve
+ waterDischargePoint
tags
asDictionary = true
extensibility = any
vocabulary = http://inspire.ec.europa.eu/codeList/WaterAppurtenanceTypeValue
xsdEncodingRule = iso19136_2007_INSPIRE_Extensions
«codeList»
Common Utility Network Elements::UtilityDeliveryTypeValue
+ collection
+ distribution
+ private
+ transport
tags
asDictionary = true
extensibility = any
vocabulary = http://inspire.ec.europa.eu/codeList/UtilityDeliveryTypeValue
xsdEncodingRule = iso19136_2007_INSPIRE_Extensions
«codeList»
Common Utility Network Elements::WarningTypeValue
+ net
+ tape
+ concretePaving
tags
asDictionary = true
extensibility = any
vocabulary = http://inspire.ec.europa.eu/codeList/WarningTypeValue
xsdEncodingRule = iso19136_2007_INSPIRE_Extensions

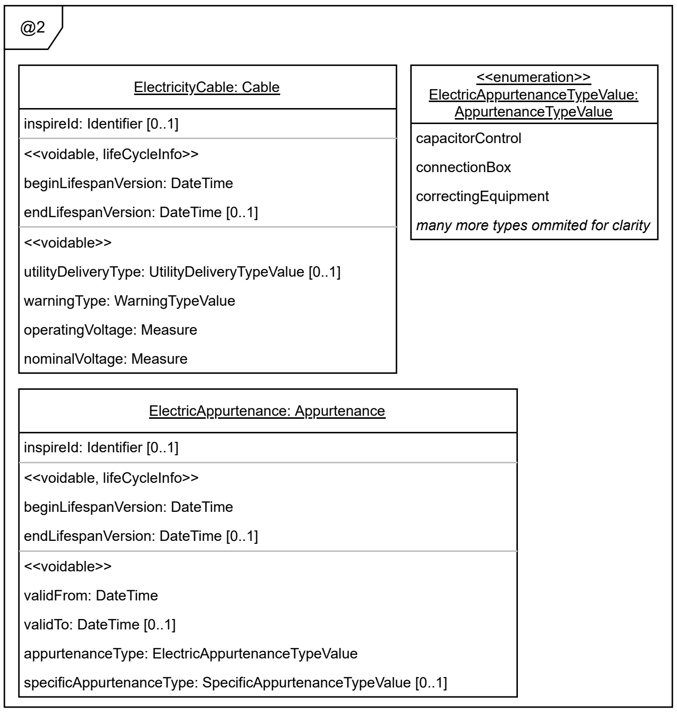
▲ **Figure B.11:** INSPIRE Water Network, from `https://inspire.ec.europa.eu/data-model/approved/r4618-ir/html/index.htm?goto=2:3:20:3:7:8933`


@2
WaterPipe: Pipe
inspireId: Identifier [0..1]
<<voidable, lifeCycleInfo>>
beginLifespanVersion: DateTime
endLifespanVersion: DateTime [0..1]
<<voidable>>
utilityDeliveryType: UtilityDeliveryTypeValue [0..1]
warningType: WarningTypeValue
pipeDiameter: Measure
presure: Measure [0..1]
waterType: WaterTypeValue
WaterAppurtenance: Appurtenance
inspireId: Identifier [0..1]
<<voidable, lifeCycleInfo>>
beginLifespanVersion: DateTime
endLifespanVersion: DateTime [0..1]
<<voidable>>
validFrom: DateTime
validTo: DateTime [0..1]
appurtenanceType: WaterAppurtenanceTypeValue
specificAppurtenanceType: SpecificAppurtenanceTypeValue [0..1]
<<enumeration>>
WaterAppurtenanceTypeValue:
AppurtenanceTypeValue
anode
clearWell
controlValve
many more types ommited for clarity

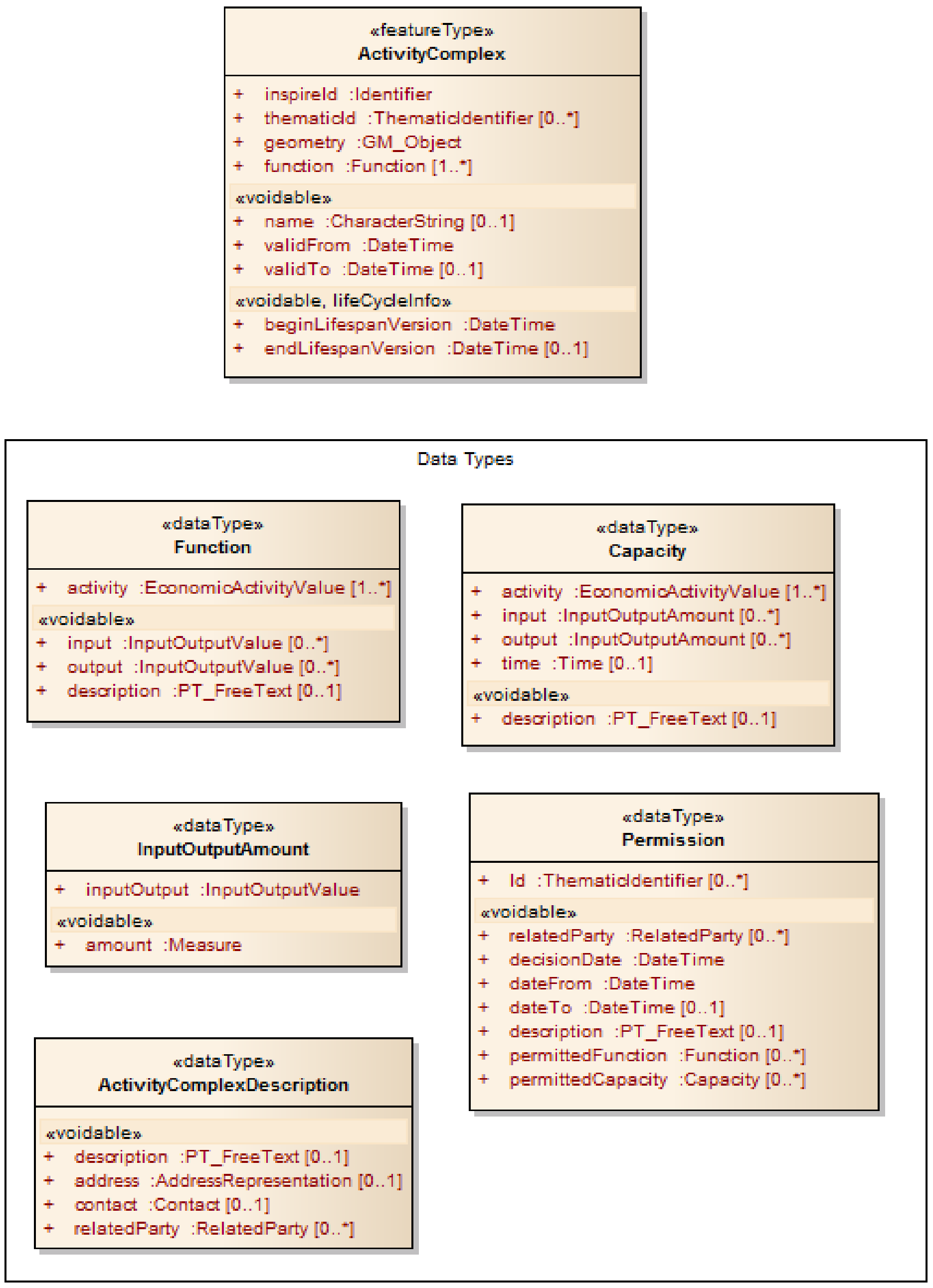
▲ **Figure B.12:** Modeling multilevel Spatial Networks - Meta-level @2 for water pipes networks


LinkSet
UtilityNetworkElement
«featureType»
Common Utility Network Elements::UtilityLinkSet
«voidable»
+ utilityDeliveryType :UtilityDeliveryTypeValue [0..1]
+ warningType :WarningTypeValue
Node
UtilityNetworkElement
«featureType»
Common Utility Network Elements::UtilityNode
«featureType»
Network::Network
«voidable»
+ geographicalName :GeographicalName [0..*]
«featureType»
Common Utility Network Elements::Cable
«featureType»
Common Utility Network Elements::Appurtenance
«voidable»
+ appurtenanceType :AppurtenanceTypeValue
+ specificAppurtenanceType :SpecificAppurtenanceTypeValue [0..1]
constraints
{"TelecommunicationsAppurtenanceTypeValue" is not in IR}
«featureType»
Common Utility Network Elements::UtilityNetwork
+ utilityNetworkType :UtilityNetworkTypeValue
+ authorityRole :RelatedParty [1..*]
«voidable»
+ utilityFacilityReference :ActivityComplex [0..*]
+ disclaimer :PT_FreeText [0..*]
constraints
{All utility network objects have inspireId}
{"telecommunications" value of "utilityNetworkType" is not in IR}
+networks
«voidable»
0..*
«featureType»
ElectricityCable
«voidable»
+ operatingVoltage :Measure
+ nominalVoltage :Measure
«codeList»
Common Utility Network Elements::AppurtenanceTypeValue
«codeList»
Common Utility Network Elements::UtilityNetworkTypeValue
+ electricity
+ oilGasChemicals
+ sewer
+ water
+ thermal
+ telecommunications [0..1]
«codeList»
Common Utility Network Elements::UtilityDeliveryTypeValue
+ collection
+ distribution
+ private
+ transport
tags
asDictionary = true
extensibility = any
vocabulary = http://inspire.ec.europa.eu/codeList/UtilityDeliveryTypeValue
xsdEncodingRule = iso19136_2007_INSPIRE_Extensions
«codeList»
ElectricityAppurtenanceTypeValue
+ capacitorControl
+ connectionBox
+ correctingEquipment
+ deliveryPoint
+ dynamicProtectiveDevice
+ fuse
+ generator
+ loadTapChanger
+ mainStation
+ netStation
+ networkProtector
+ openPoint
+ primaryMeter
+ recloserElectronicControl
+ recloserHydraulicControl
+ regulatorControl
+ relayControl
+ sectionalizerElectronicControl
+ sectionalizerHydraulicControl
+ streetLight
+ subStation
+ switch
+ transformer
+ voltageRegulator
+ detectionEquipment
+ pointSettingMachine
+ monitoringAndControlEquipment
+ anode
tags
asDictionary = true
extensibility = any
vocabulary = http://inspire.ec.europa.eu/codeList/ElectricityAppurtenanceTypeValue
xsdEncodingRule = iso19136_2007_INSPIRE_Extensions
«codeList»
Common Utility Network Elements::WarningTypeValue
+ net
+ tape
+ concretePaving
tags
asDictionary = true
extensibility = any
vocabulary = http://inspire.ec.europa.eu/codeList/WarningTypeValue
xsdEncodingRule = iso19136_2007_INSPIRE_Extensions

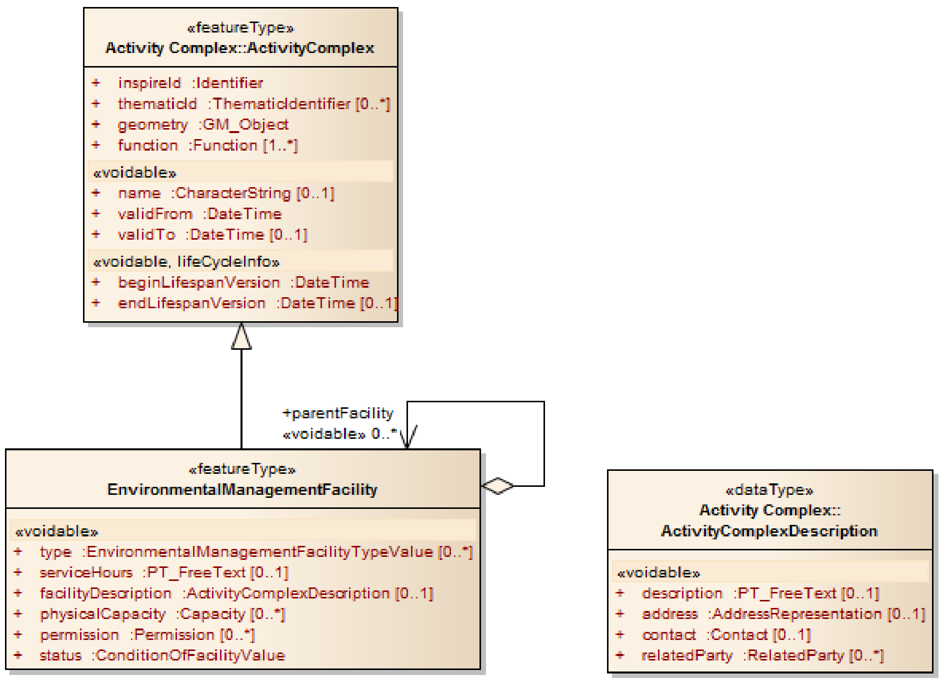
▲ **Figure B.13:** INSPIRE Electricity Network, from `https://inspire.ec.europa.eu/data-model/approved/r4618-ir/html/index.htm?goto=2:3:20:3:2:8910`


@2
ElectricityCable: Cable
inspireId: Identifier [0..1]
<<voidable, lifeCycleInfo>>
beginLifespanVersion: DateTime
endLifespanVersion: DateTime [0..1]
<<voidable>>
utilityDeliveryType: UtilityDeliveryTypeValue [0..1]
warningType: WarningTypeValue
operatingVoltage: Measure
nominalVoltage: Measure
<<enumeration>>
ElectricAppurtenanceTypeValue: AppurtenanceTypeValue
capacitorControl
connectionBox
correctingEquipment
many more types ommited for clarity
ElectricAppurtenance: Appurtenance
inspireId: Identifier [0..1]
<<voidable, lifeCycleInfo>>
beginLifespanVersion: DateTime
endLifespanVersion: DateTime [0..1]
<<voidable>>
validFrom: DateTime
validTo: DateTime [0..1]
appurtenanceType: ElectricAppurtenanceTypeValue
specificAppurtenanceType: SpecificAppurtenanceTypeValue [0..1]

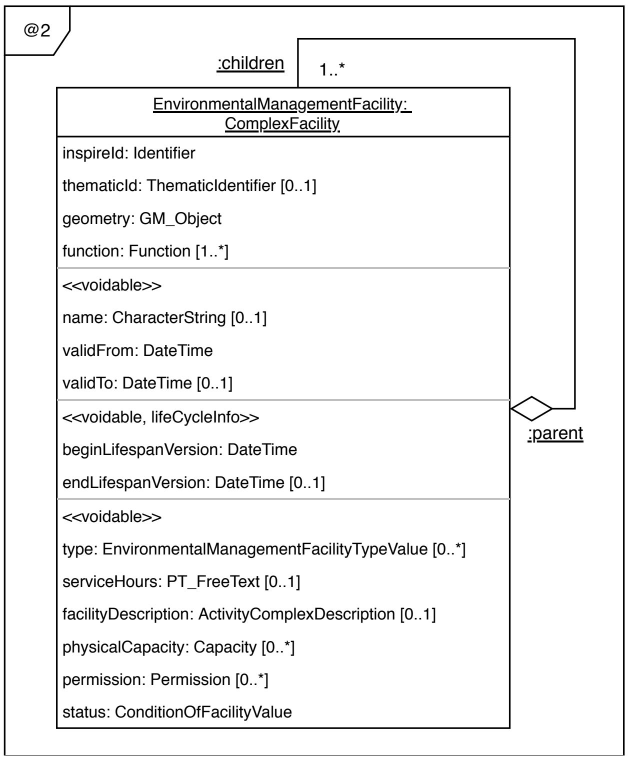
▲ **Figure B.14:** Modeling multilevel Spatial Networks - Meta-level @2 for electricity networks


«featureType»
ActivityComplex
+ inspireId :Identifier
+ thematicId :ThematicIdentifier [0..*]
+ geometry :GM_Object
+ function :Function [1..*]
«voidable»
+ name :CharacterString [0..1]
+ validFrom :DateTime
+ validTo :DateTime [0..1]
«voidable, lifeCycleInfo»
+ beginLifespanVersion :DateTime
+ endLifespanVersion :DateTime [0..1]
Data Types
«dataType»
Function
+ activity :EconomicActivityValue [1..*]
«voidable»
+ input :InputOutputValue [0..*]
+ output :InputOutputValue [0..*]
+ description :PT_FreeText [0..1]
«dataType»
Capacity
+ activity :EconomicActivityValue [1..*]
+ input :InputOutputAmount [0..*]
+ output :InputOutputAmount [0..*]
+ time :Time [0..1]
«voidable»
+ description :PT_FreeText [0..1]
«dataType»
InputOutputAmount
+ inputOutput :InputOutputValue
«voidable»
+ amount :Measure
«dataType»
Permission
+ Id :ThematicIdentifier [0..*]
«voidable»
+ relatedParty :RelatedParty [0..*]
+ decisionDate :DateTime
+ dateFrom :DateTime
+ dateTo :DateTime [0..1]
+ description :PT_FreeText [0..1]
+ permittedFunction :Function [0..*]
+ permittedCapacity :Capacity [0..*]
«dataType»
ActivityComplexDescription
«voidable»
+ description :PT_FreeText [0..1]
+ address :AddressRepresentation [0..1]
+ contact :Contact [0..1]
+ relatedParty :RelatedParty [0..*]

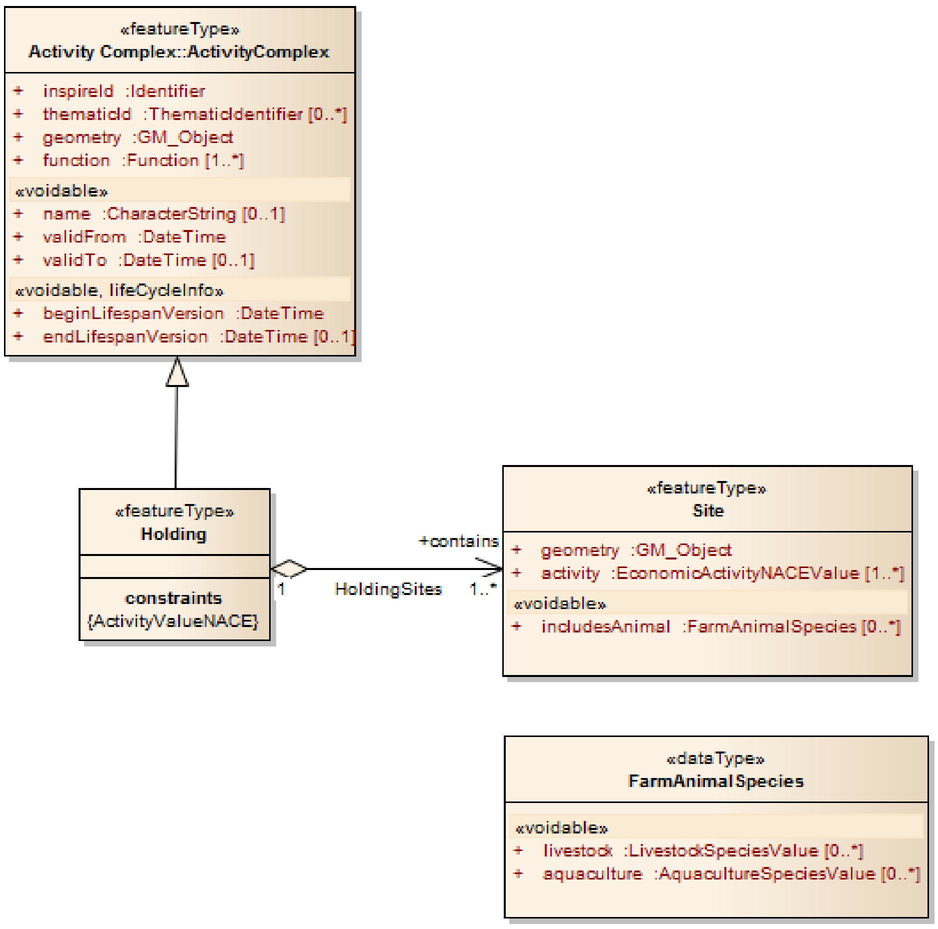
▲ **Figure B.15:** INSPIRE Activity Complex Base Model, from `https://inspire.ec.europa.eu/data-model/approved/r4618-ir/html/index.htm?goto=3:1:4:1:8990`


«featureType»
Activity Complex::ActivityComplex
+ inspireId :Identifier
+ thematicId :ThematicIdentifier [0..*]
+ geometry :GM_Object
+ function :Function [1..*]
«voidable»
+ name :CharacterString [0..1]
+ validFrom :DateTime
+ validTo :DateTime [0..1]
«voidable, lifeCycleInfo»
+ beginLifespanVersion :DateTime
+ endLifespanVersion :DateTime [0..1]
+parentFacility
«voidable» 0..*
«featureType»
EnvironmentalManagementFacility
«voidable»
+ type :EnvironmentalManagementFacilityTypeValue [0..*]
+ serviceHours :PT_FreeText [0..1]
+ facilityDescription :ActivityComplexDescription [0..1]
+ physicalCapacity :Capacity [0..*]
+ permission :Permission [0..*]
+ status :ConditionOfFacilityValue
«dataType»
Activity Complex::
ActivityComplexDescription
«voidable»
+ description :PT_FreeText [0..1]
+ address :AddressRepresentation [0..1]
+ contact :Contact [0..1]
+ relatedParty :RelatedParty [0..*]

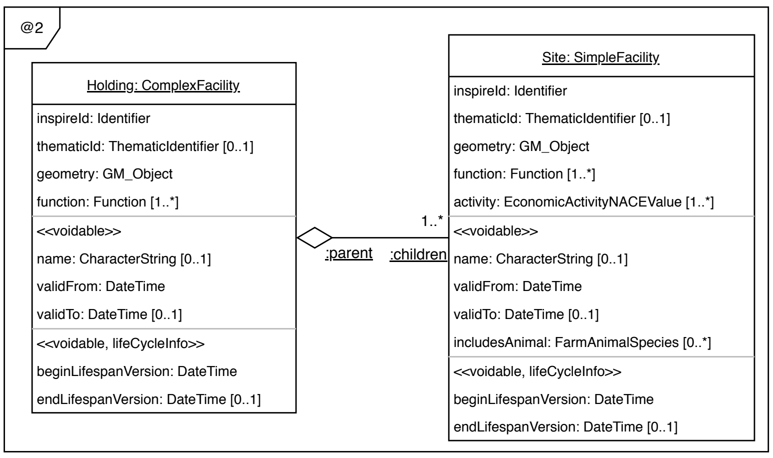
▲ **Figure B.16:** INSPIRE Environmental Management Facilities, from `https://inspire.ec.europa.eu/data-model/approved/r4618-ir/html/index.htm?goto=2:3:20:2:8857`


@2
:children
1..*
EnvironmentalManagementFacility: ComplexFacility
inspireId: Identifier
thematicId: ThematicIdentifier [0..1]
geometry: GM_Object
function: Function [1..*]
<<voidable>>
name: CharacterString [0..1]
validFrom: DateTime
validTo: DateTime [0..1]
<<voidable, lifeCycleInfo>>
beginLifespanVersion: DateTime
endLifespanVersion: DateTime [0..1]
<<voidable>>
type: EnvironmentalManagementFacilityTypeValue [0..*]
serviceHours: PT_FreeText [0..1]
facilityDescription: ActivityComplexDescription [0..1]
physicalCapacity: Capacity [0..*]
permission: Permission [0..*]
status: ConditionOfFacilityValue
:parent

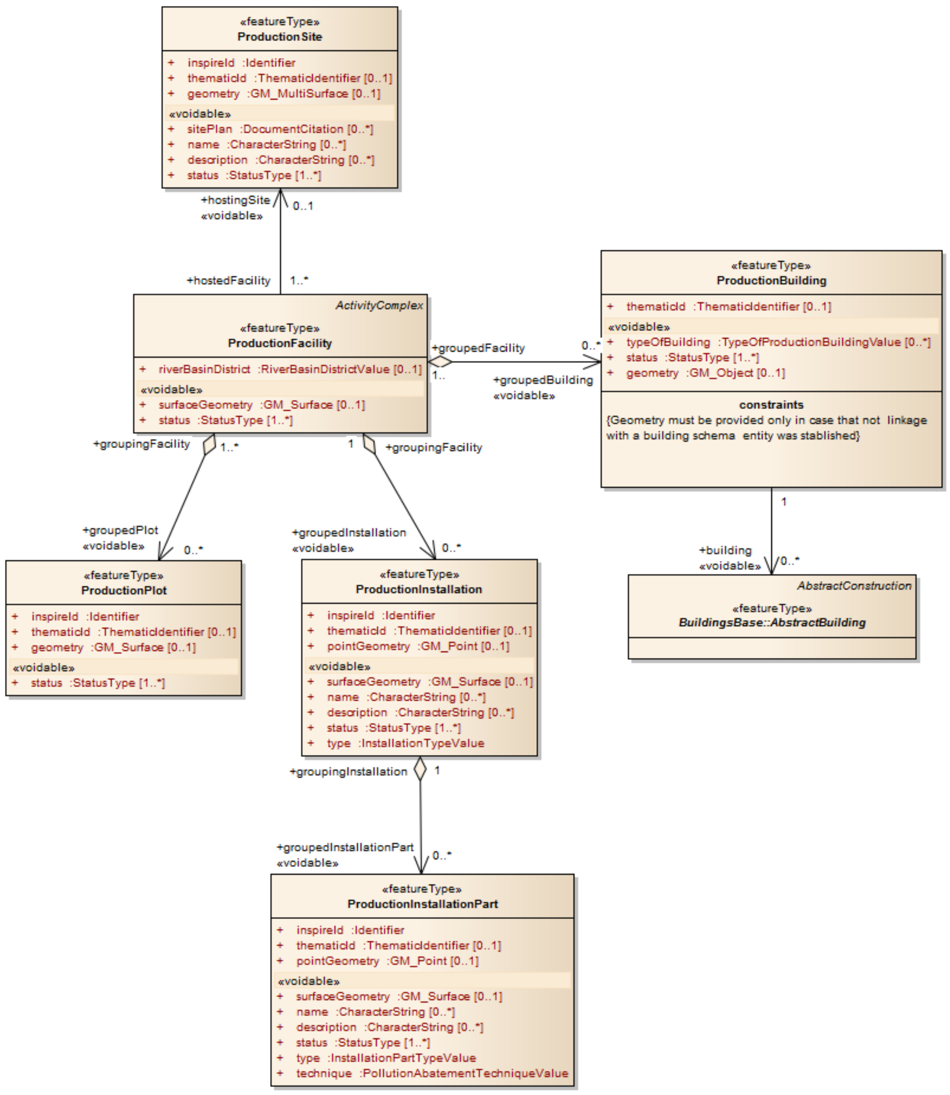
▲ **Figure B.17:** Modeling multilevel Facilities Management - Meta-level @2 for environmental management facilities.


«featureType»
Activity Complex::ActivityComplex
+ inspireId :Identifier
+ thematicId :ThematicIdentifier [0..*]
+ geometry :GM_Object
+ function :Function [1..*]
«voidable»
+ name :CharacterString [0..1]
+ validFrom :DateTime
+ validTo :DateTime [0..1]
«voidable, lifeCycleInfo»
+ beginLifespanVersion :DateTime
+ endLifespanVersion :DateTime [0..1]
«featureType»
Holding
constraints
{ActivityValueNACE}
+contains
1
HoldingSites
1..*
«featureType»
Site
+ geometry :GM_Object
+ activity :EconomicActivityNACEValue [1..*]
«voidable»
+ includesAnimal :FarmAnimalSpecies [0..*]
«dataType»
FarmAnimalSpecies
«voidable»
+ livestock :LivestockSpeciesValue [0..*]
+ aquaculture :AquacultureSpeciesValue [0..*]

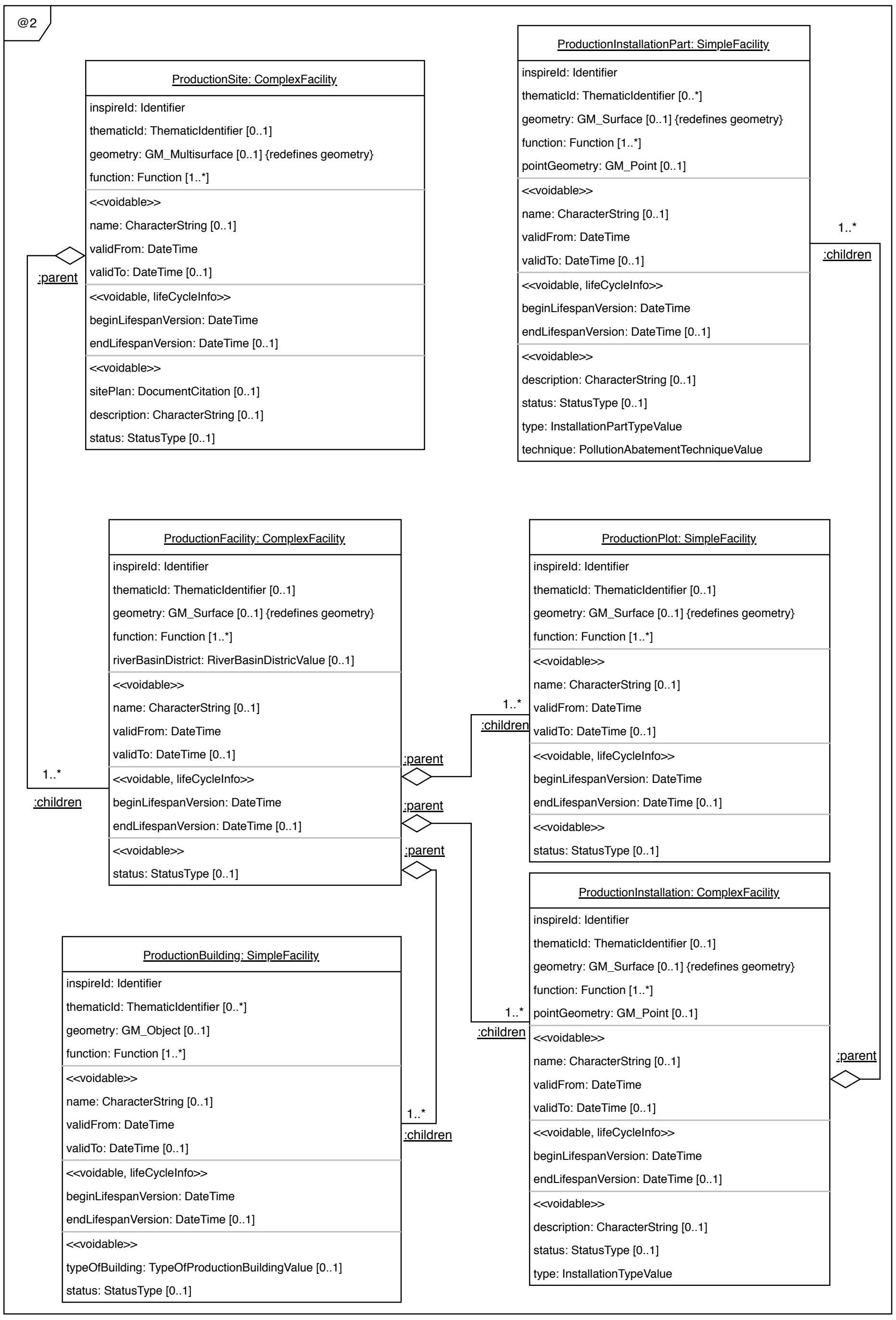
▲ **Figure B.18:** INSPIRE Agricultural and Aquaculture Facilities, from `https://inspire.ec.europa.eu/data-model/approved/r4618-ir/html/index.htm?goto=2:3:3:1:7925`


@2
Holding: ComplexFacility
inspireId: Identifier
thematicId: ThematicIdentifier [0..1]
geometry: GM_Object
function: Function [1..*]
<<voidable>>
name: CharacterString [0..1]
validFrom: DateTime
validTo: DateTime [0..1]
<<voidable, lifeCycleInfo>>
beginLifespanVersion: DateTime
endLifespanVersion: DateTime [0..1]
1..*
:parent
:children
Site: SimpleFacility
inspireId: Identifier
thematicId: ThematicIdentifier [0..1]
geometry: GM_Object
function: Function [1..*]
activity: EconomicActivityNACEValue [1..*]
<<voidable>>
name: CharacterString [0..1]
validFrom: DateTime
validTo: DateTime [0..1]
includesAnimal: FarmAnimalSpecies [0..*]
<<voidable, lifeCycleInfo>>
beginLifespanVersion: DateTime
endLifespanVersion: DateTime [0..1]

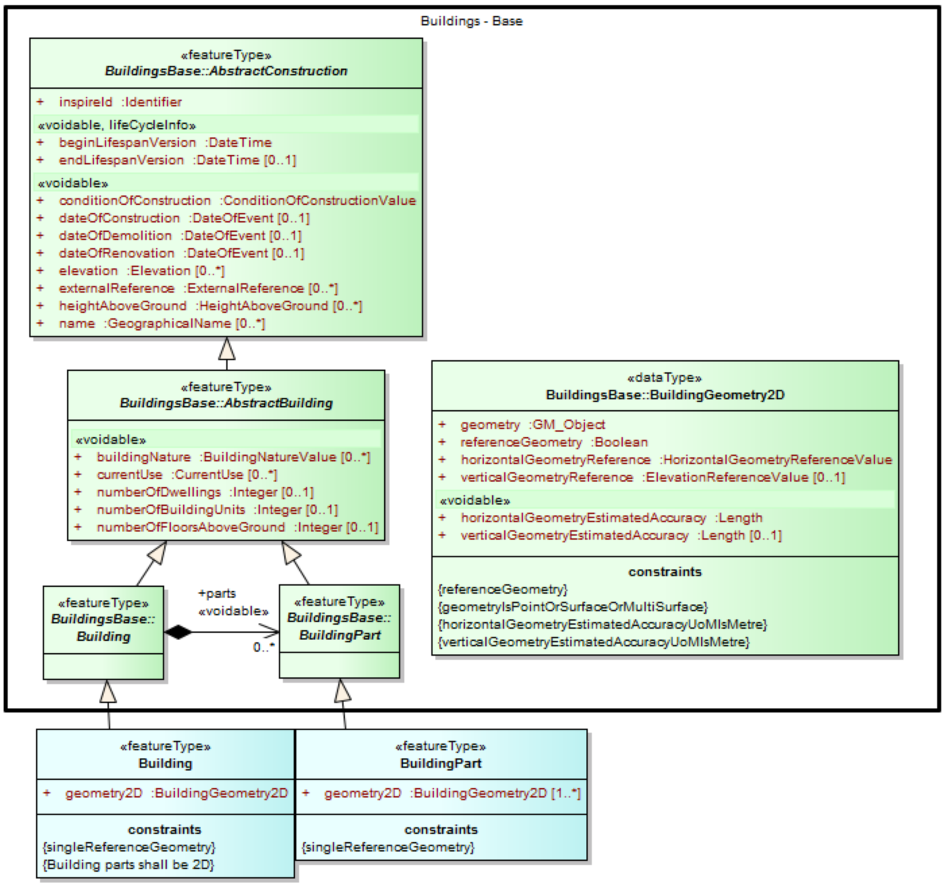
▲ **Figure B.19:** Modeling multilevel Facilities Management - Meta-level @2 for agricultural facilities.


«featureType»
ProductionSite
+ inspireId :Identifier
+ thematicId :ThematicIdentifier [0..1]
+ geometry :GM_MultiSurface [0..1]
«voidable»
+ sitePlan :DocumentCitation [0..*]
+ name :CharacterString [0..*]
+ description :CharacterString [0..*]
+ status :StatusType [1..*]
+hostingSite
«voidable»
0..1
+hostedFacility
1..*
ActivityComplex
«featureType»
ProductionFacility
+ riverBasinDistrict :RiverBasinDistrictValue [0..1]
«voidable»
+ surfaceGeometry :GM_Surface [0..1]
+ status :StatusType [1..*]
+groupedFacility
1..
0..*
+groupedBuilding
«voidable»
«featureType»
ProductionBuilding
+ thematicId :ThematicIdentifier [0..1]
«voidable»
+ typeOfBuilding :TypeOfProductionBuildingValue [0..*]
+ status :StatusType [1..*]
+ geometry :GM_Object [0..1]
constraints
{Geometry must be provided only in case that not linkage with a building schema entity was stablished}
+groupingFacility
1..*
1
+groupingFacility
+groupedPlot
«voidable»
0..*
+groupedInstallation
«voidable»
0..*
1
+building
«voidable»
0..*
«featureType»
ProductionPlot
+ inspireId :Identifier
+ thematicId :ThematicIdentifier [0..1]
+ geometry :GM_Surface [0..1]
«voidable»
+ status :StatusType [1..*]
«featureType»
ProductionInstallation
+ inspireId :Identifier
+ thematicId :ThematicIdentifier [0..1]
+ pointGeometry :GM_Point [0..1]
«voidable»
+ surfaceGeometry :GM_Surface [0..1]
+ name :CharacterString [0..*]
+ description :CharacterString [0..*]
+ status :StatusType [1..*]
+ type :InstallationTypeValue
AbstractConstruction
«featureType»
BuildingsBase::AbstractBuilding
+groupingInstallation
1
+groupedInstallationPart
«voidable»
0..*
«featureType»
ProductionInstallationPart
+ inspireId :Identifier
+ thematicId :ThematicIdentifier [0..1]
+ pointGeometry :GM_Point [0..1]
«voidable»
+ surfaceGeometry :GM_Surface [0..1]
+ name :CharacterString [0..*]
+ description :CharacterString [0..*]
+ status :StatusType [1..*]
+ type :InstallationPartTypeValue
+ technique :PollutionAbatementTechniqueValue

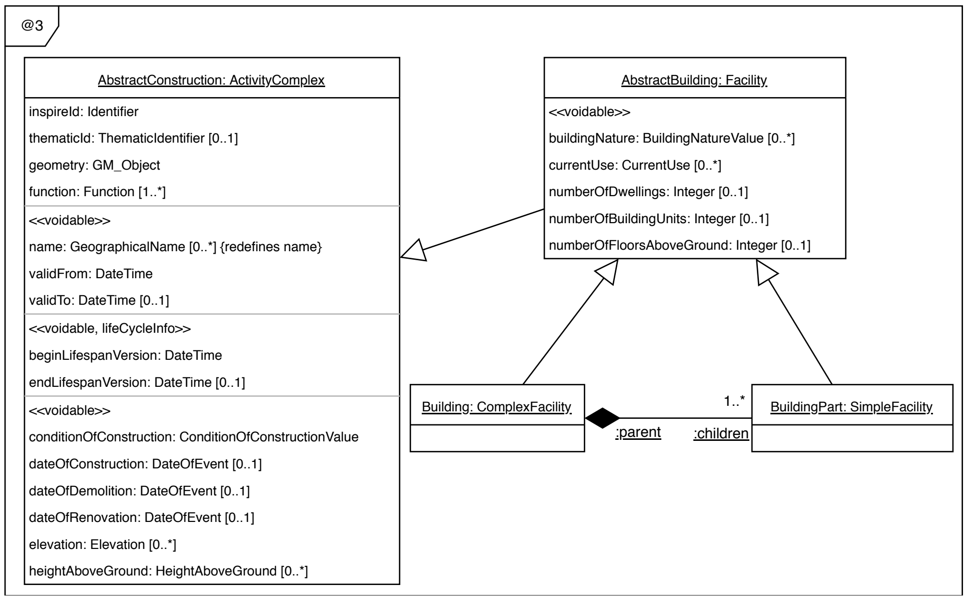
▲ **Figure B.20:** INSPIRE Production and Industrial Facilities, from `https://inspire.ec.europa.eu/data-model/approved/r4618-ir/html/index.htm?goto=2:3:15:1:8641`


@2
ProductionSite: ComplexFacility
inspireId: Identifier
thematicId: ThematicIdentifier [0..1]
geometry: GM_Multisurface [0..1] {redefines geometry}
function: Function [1..*]
<<voidable>>
name: CharacterString [0..1]
validFrom: DateTime
validTo: DateTime [0..1]
<<voidable, lifeCycleInfo>>
beginLifespanVersion: DateTime
endLifespanVersion: DateTime [0..1]
<<voidable>>
sitePlan: DocumentCitation [0..1]
description: CharacterString [0..1]
status: StatusType [0..1]
ProductionInstallationPart: SimpleFacility
inspireId: Identifier
thematicId: ThematicIdentifier [0..*]
geometry: GM_Surface [0..1] {redefines geometry}
function: Function [1..*]
pointGeometry: GM_Point [0..1]
<<voidable>>
name: CharacterString [0..1]
validFrom: DateTime
validTo: DateTime [0..1]
<<voidable, lifeCycleInfo>>
beginLifespanVersion: DateTime
endLifespanVersion: DateTime [0..1]
<<voidable>>
description: CharacterString [0..1]
status: StatusType [0..1]
type: InstallationPartTypeValue
technique: PollutionAbatementTechniqueValue
:parent
1..*
:children
ProductionFacility: ComplexFacility
inspireId: Identifier
thematicId: ThematicIdentifier [0..1]
geometry: GM_Surface [0..1] {redefines geometry}
function: Function [1..*]
riverBasinDistrict: RiverBasinDistricValue [0..1]
<<voidable>>
name: CharacterString [0..1]
validFrom: DateTime
validTo: DateTime [0..1]
<<voidable, lifeCycleInfo>>
beginLifespanVersion: DateTime
endLifespanVersion: DateTime [0..1]
<<voidable>>
status: StatusType [0..1]
ProductionPlot: SimpleFacility
inspireId: Identifier
thematicId: ThematicIdentifier [0..1]
geometry: GM_Surface [0..1] {redefines geometry}
function: Function [1..*]
<<voidable>>
name: CharacterString [0..1]
validFrom: DateTime
validTo: DateTime [0..1]
<<voidable, lifeCycleInfo>>
beginLifespanVersion: DateTime
endLifespanVersion: DateTime [0..1]
<<voidable>>
status: StatusType [0..1]
1..*
:children
:parent
:parent
:parent
1..*
:children
ProductionInstallation: ComplexFacility
inspireId: Identifier
thematicId: ThematicIdentifier [0..1]
geometry: GM_Surface [0..1] {redefines geometry}
function: Function [1..*]
pointGeometry: GM_Point [0..1]
<<voidable>>
name: CharacterString [0..1]
validFrom: DateTime
validTo: DateTime [0..1]
<<voidable, lifeCycleInfo>>
beginLifespanVersion: DateTime
endLifespanVersion: DateTime [0..1]
<<voidable>>
description: CharacterString [0..1]
status: StatusType [0..1]
type: InstallationTypeValue
1..*
:children
:parent
1..*
:children
ProductionBuilding: SimpleFacility
inspireId: Identifier
thematicId: ThematicIdentifier [0..*]
geometry: GM_Object [0..1]
function: Function [1..*]
<<voidable>>
name: CharacterString [0..1]
validFrom: DateTime
validTo: DateTime [0..1]
<<voidable, lifeCycleInfo>>
beginLifespanVersion: DateTime
endLifespanVersion: DateTime [0..1]
<<voidable>>
typeOfBuilding: TypeOfProductionBuildingValue [0..1]
status: StatusType [0..1]

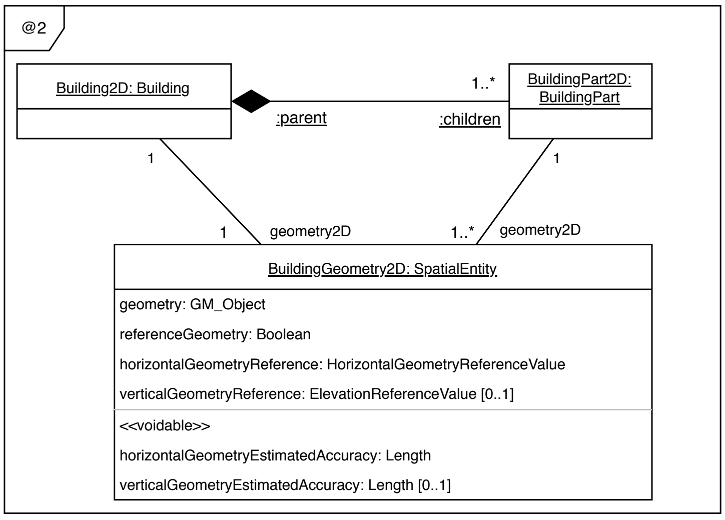
▲ **Figure B.21:** Modeling multilevel Facilities Management - Meta-level @2 for production and industrial facilities.


Buildings - Base
«featureType»
BuildingsBase::AbstractConstruction
+ inspireId :Identifier
«voidable, lifeCycleInfo»
+ beginLifespanVersion :DateTime
+ endLifespanVersion :DateTime [0..1]
«voidable»
+ conditionOfConstruction :ConditionOfConstructionValue
+ dateOfConstruction :DateOfEvent [0..1]
+ dateOfDemolition :DateOfEvent [0..1]
+ dateOfRenovation :DateOfEvent [0..1]
+ elevation :Elevation [0..*]
+ externalReference :ExternalReference [0..*]
+ heightAboveGround :HeightAboveGround [0..*]
+ name :GeographicalName [0..*]
«featureType»
BuildingsBase::AbstractBuilding
«voidable»
+ buildingNature :BuildingNatureValue [0..*]
+ currentUse :CurrentUse [0..*]
+ numberOfDwellings :Integer [0..1]
+ numberOfBuildingUnits :Integer [0..1]
+ numberOfFloorsAboveGround :Integer [0..1]
«dataType»
BuildingsBase::BuildingGeometry2D
+ geometry :GM_Object
+ referenceGeometry :Boolean
+ horizontalGeometryReference :HorizontalGeometryReferenceValue
+ verticalGeometryReference :ElevationReferenceValue [0..1]
«voidable»
+ horizontalGeometryEstimatedAccuracy :Length
+ verticalGeometryEstimatedAccuracy :Length [0..1]
constraints
{referenceGeometry}
{geometryIsPointOrSurfaceOrMultiSurface}
{horizontalGeometryEstimatedAccuracyUoMIsMetre}
{verticalGeometryEstimatedAccuracyUoMIsMetre}
«featureType»
BuildingsBase::
Building
+parts
«voidable»
0..*
«featureType»
BuildingsBase::
BuildingPart
«featureType»
Building
+ geometry2D :BuildingGeometry2D
constraints
{singleReferenceGeometry}
{Building parts shall be 2D}
«featureType»
BuildingPart
+ geometry2D :BuildingGeometry2D [1..*]
constraints
{singleReferenceGeometry}

▲ **Figure B.22:** INSPIRE Buildings Base and Core 2D, from `https://inspire.ec.europa.eu/data-model/approved/r4618-ir/html/index.htm?goto=2:3:2:2:7911`


@3
AbstractConstruction: ActivityComplex
inspireId: Identifier
thematicId: ThematicIdentifier [0..1]
geometry: GM_Object
function: Function [1..*]
<<voidable>>
name: GeographicalName [0..*] {redefines name}
validFrom: DateTime
validTo: DateTime [0..1]
<<voidable, lifeCycleInfo>>
beginLifespanVersion: DateTime
endLifespanVersion: DateTime [0..1]
<<voidable>>
conditionOfConstruction: ConditionOfConstructionValue
dateOfConstruction: DateOfEvent [0..1]
dateOfDemolition: DateOfEvent [0..1]
dateOfRenovation: DateOfEvent [0..1]
elevation: Elevation [0..*]
heightAboveGround: HeightAboveGround [0..*]
AbstractBuilding: Facility
<<voidable>>
buildingNature: BuildingNatureValue [0..*]
currentUse: CurrentUse [0..*]
numberOfDwellings: Integer [0..1]
numberOfBuildingUnits: Integer [0..1]
numberOfFloorsAboveGround: Integer [0..1]
Building: ComplexFacility
:parent
1..*
:children
BuildingPart: SimpleFacility

▲ **Figure B.23:** Modeling multilevel Facilities Management - Meta-level @3 for building facilities.


@2
Building2D: Building
:parent
1..*
:children
BuildingPart2D: BuildingPart
1
1
1
geometry2D
1..*
geometry2D
BuildingGeometry2D: SpatialEntity
geometry: GM_Object
referenceGeometry: Boolean
horizontalGeometryReference: HorizontalGeometryReferenceValue
verticalGeometryReference: ElevationReferenceValue [0..1]
<<voidable>>
horizontalGeometryEstimatedAccuracy: Length
verticalGeometryEstimatedAccuracy: Length [0..1]

▲ **Figure B.24:** Modeling multilevel Facilities Management - Meta-level @2 for building facilities.

Bibliography

[ACL$^+$19] Suilen Hernández Alvarado, Alejandro Cortiñas, Miguel R. Luaces, Oscar Pedreira, and Ángeles Saavedra Places. A domain specific language for web-based GIS. In *Procs. of the 15th International Conference on Web Information Systems and Technologies (WEBIST 2019)*, pages 462–469. ScitePress, 2019.

[ACL$^+$20] S. H. Alvarado, A. Cortiñas, M. R. Luaces, O. Pedreira, and A. S. Places. Developing web-based geographic information systems with a dsl: Proposal and case study. *Journal of Web Engineering*, 19:167–194, 2020.

[ADH$^+$89] Hiralal Agrawal, Richard DeMillo, R_ Hathaway, William Hsu, Wynne Hsu, Edward Krauser, Rhonda J. Martin, Aditya Mathur, and Eugene Spafford. Design of mutant operators for the C programming language. Technical report, Technical Report SERC-TR-41-P, Software Engineering Research Center, Department of Computer Science, Purdue University, Indiana, 1989.

[AHJK$^+$14] Samir Al-Hilank, Martin Jung, Detlef Kips, Dirk Husemann, and Michael Philippsen. Using multi level-modeling techniques for managing mapping information. In *Procs. of International Workshop on Muli-Level modelling (MULTI'14) - MODELS Workshops*, 2014.

[AK01] Colin Atkinson and Thomas Kühne. The Essence of Multilevel Metamodeling. In *International Conference on the Unified Modeling Language*, pages 19–33. Springer, 2001.

[AK02] Colin Atkinson and Thomas Kühne. Rearchitecting the uml infrastructure. *ACM Trans. Model. Comput. Simul.*, 12(4):290–321, October 2002.

[AK03] C. Atkinson and T. Kuhne. Model-driven development: a metamodeling foundation. *IEEE Software*, 20(5), September 2003.

[AK08] Colin Atkinson and Thomas Kühne. Reducing accidental complexity in domain models. *Software & Systems Modeling*, 7(3), July 2008.

[Atk97] C. Atkinson. Meta-modelling for distributed object environments. In *Proceedings First International Enterprise Distributed Object Computing Workshop*, pages 90–101, 1997.

[BCW17a] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis lectures on software engineering*, 3(1):1–207, 2017.

[BCW17b] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182, 2017.

[BCW17c] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice: Second Edition*. Morgan & Claypool Publishers, March 2017.

[Ben17] Björn Benner. A multi-level approach for model-based user interface development. In *Procs. of 4^{th} International Workshop on Muli-Level modelling (MULTI'17) - MODELS Workshops*, 2017.

[BSRC10a] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[BSRC10b] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information systems*, 35(6):615–636, 2010.

[Bud80] Timothy Alan Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven, CT, USA, 1980.

[BW06a] Bartosz Bogacki and Bartosz Walter. Aspect-oriented Response Injection: an Alternative to Classical Mutation Testing. In *Software Engineering Techniques: Design for Quality*, pages 273–282. Springer, Boston, MA, 2006.

[BW06b] Bartosz Bogacki and Bartosz Walter. Evaluation of Test Code Quality with Aspect-Oriented Mutations. In *Extreme Programming and Agile Processes in Software Engineering*, pages 202–204. Springer, Berlin, Heidelberg, June 2006.

[BZR16] Maicon Bernardino, Avelino F Zorzo, and Elder M Rodrigues. Canopus: A domain-specific language for modeling performance testing. In *Procs. of the IEEE International Conference on Software Testing, Verification and Validation (ICST 2016)*, pages 157–167. IEEE, 2016.

[CLP$^+$17] Alejandro Cortiñas, Miguel R Luaces, Oscar Pedreira, Ángeles S Places, and Jennifer Pérez. Web-based geographic information systems sple: Domain analysis and experience report. In *Procs. of the 21st International Systems and Software Product Line Conference-Volume A*, pages 190–194. ACM, 2017.

[CLPP17] Alejandro Cortiñas, Miguel R Luaces, Oscar Pedreira, and Ángeles S Places. Scaffolding and in-browser generation of web-based gis applications in a spl tool. In *Procs. of the 21st International Systems and Software Product Line Conference-Volume B*, pages 46–49. ACM, 2017.

[CN15] P Clements and L Northrop. Software product lines: Practices and patterns: Practices and patterns, 2015.

[CNM18] Pablo C. Cañizares, Alberto Núñez, and Mercedes G. Merayo. Mutomvo: Mutation testing framework for simulated cloud and hpc environments. *Journal of Systems and Software*, 143:187 – 207, 2018.

[Der06] Anna Derezińska. Advanced mutation operators applicable in C# programs. In *Software Engineering Techniques: Design for Quality*, pages 283–288. Springer, 2006.

[DH14] Anna Derezińska and Konrad Ha\las. Analysis of mutation operators for the python language. In *Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX. June 30–July 4, 2014, Brunów, Poland*, pages 155–164. Springer, 2014.

[dLG10] Juan de Lara and Esther Guerra. Deep meta-modelling with metadepth. In Jan Vitek, editor, *Objects, Models, Components, Patterns*, pages 1–20, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[dLGC14] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. When and How to Use Multilevel Modelling. *ACM Trans. Softw. Eng. Methodol.*, 24(2):12:1–12:46, December 2014.

[DLS78] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. volume 11, pages 34–41, April 1978.

[DOAM17] Lin Deng, Jeff Offutt, Paul Ammann, and Nariman Mirzaei. Mutation operators for testing android apps. *Information and Software Technology*, 81:154–168, 2017.

[DPMBDJ$^+$15] Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Juan José Domínguez-Jiménez, Antonio García-Domínguez, and Francisco Palomo-Lozano. Class mutation operators for C++ object-oriented systems. volume 70, pages 137–148, 2015.

[DSDS16] Luís Moreira De Sousa and Alberto Rodrigues Da Silva. A domain specific language for spatial simulation scenarios. *GeoInformatica*, 20(1):117–149, 2016.

[Fow10] Martin Fowler. *Domain-specific languages.* Pearson Education, 2010.

[Fra13] Ulrich Frank. Domain-specific modeling languages: Requirements analysis and design guidelines. In Reinhartz-Berger I., Sturm A., Clark T., Cohen S., and Bettin J., editors, *Domain Engineering.* Springer, 2013.

[Fra16] Ulrich Frank. Designing models and systems to support it management: A case for multilevel modeling. In *Procs. of $2^n d$ International Workshop on Muli-Level modelling (MULTI'16) - MODELS Workshops*, 2016.

[Fra18] Ulrich Frank. Toward a unified conception of multi-level modelling: advanced requirements. In *Procs. of the 5th International Workshop on Multi-level Modelling (MULTI'2018)*, pages 718–727, 2018.

[FRM13] Fabiano Cutigi Ferrari, Awais Rashid, and José Carlos Maldonado. Towards the practical mutation testing of aspectj programs. *Science of Computer Programming*, 78(9):1639 – 1662, 2013.

[Gÿ4] Ralf Hartmut Güting. Graphdb: Modeling and querying graphs in databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, page 297–308, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[Inta] International Organization for Standardization. Iso 19107:2003 - geographic information: Spatial schema. https://www.iso.org/standard/26012.html. visited on 2019-07-02.

[Intb] International Organization for Standardization. Iso 19119:2016 - geographic information: Services. https://www.iso.org/standard/59221.html. visited on 2019-07-02.

[Intc] International Organization for Standardization. Iso 19125:2004 - geographic information — simple feature access — part 1: Common architecture. https://www.iso.org/standard/40114.html. visited on 2020-06-21.

[IPT$^+$07] Sean A. Irvine, Tin Pavlinic, Leonard Trigg, John G. Cleary, Stuart Inglis, and Mark Utting. Jumble java byte code to measure the effectiveness of unit tests. In *Testing: Academic and industrial conference practice and research techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 169–175. IEEE, 2007.

[JFD$^+$13] Lisboa-Filho J., Nalon F.R., Peixoto D.A., Sampaio G.B., and de Vasconcelos Borges K.A. Domain and model driven geographic database design. In Reinhartz-Berger I., Sturm A., Clark T., Cohen S., and Bettin J., editors, *Domain Engineering*. Springer, 2013.

[JH11] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. volume 37, pages 649–678, 2011.

[JSK11] Rene Just, Franz Schweiggert, and Gregory M. Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 612–615. IEEE Computer Society, 2011.

[Jus14] René Just. The Major Mutation Framework: Efficient and Scalable Mutation Analysis for Java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 433–436, New York, NY, USA, 2014. ACM.

[KBM16] Tomaž Kosar, Sudev Bohra, and Marjan Mernik. Domain-specific languages: A systematic mapping study. *Information and Software Technology*, 71:77–91, 2016.

[KCH$^+$90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University - Software Engineering Institute, 1990.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming*, pages 220–242. Springer, Berlin, Heidelberg, June 1997.

[Kut16] Tatjana Kutzner. *Geospatial Data Modelling and Model-driven Transformation of Geospatial Data based on UML Profiles.* PhD thesis, Technical University of Munich, 2016.

[LFSNdVB10] Jugurta Lisboa-Filho, Gustavo Breder Sampaio, Filipe Ribeiro Nalon, and Karla A. de V. Borges. A uml profile for conceptual modeling in gis domain. In *Procs. of DE Workshop at International Conference on Advanced Information Systems Engineering (CAISE 2010)*, pages 18–31, 2010.

[LK11] Steven Lolong and Achmad I Kistijantoro. Domain specific language (dsl) development for desktop-based database application generator. In *Procs. of the International Conference on Electrical Engineering and Informatics*, pages 1–6. IEEE, 2011.

[LVBT$^+$17] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Enabling mutation testing for android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 233–244, 2017.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344, 2005.

[MO05a] Yu-Seung Ma and Jeff Offutt. Description of class mutation operators for java. *Electronics and Telecommunications Research Institute, Korea, Tech. Rep.*, 2005.

[MO05b] Yu-Seung Ma and Jeff Offutt. Description of method-level mutation operators for java. *Electronics and Telecommunications Research Institute, Korea, Tech. Rep.*, 2005.

[MOK05] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. MuJava: An automated class mutation system. volume 15, pages 97–133, 2005.

[MOK06] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. MuJava: A Mutation System for Java. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 827–830, New York, NY, USA, 2006. ACM.

[MR10] L. Madeyski and N. Radyk. Judy - a mutation testing tool for java. volume 4, pages 32–42, February 2010.

[MU12] Pedro Reales Mateo and Macario Polo Usaola. Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases. In *28th IEEE International Conference on Software Maintenance, (ICSM 2012)*, pages 646–649, 2012.

[NN17] Damir Nesic and Mattias Nyberg. Applying multi-level modeling to data integration in product line engineering. In *Procs. of 4^{th} International Workshop on Muli-Level modelling (MULTI'17) - MODELS Workshops*, 2017.

[NWH+15] Jay Nanavati, Fan Wu, Mark Harman, Yue Jia, and Jens Krinke. Mutation testing of memory-related operators. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–10. IEEE, 2015.

[OLR+96] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. volume 5, pages 99–118, 1996.

[PM07a] Oscar Pastor and Juan Carlos Molina. *Model-driven architecture in practice: a software production environment based on conceptual modeling.* Springer, 2007.

[PM07b] Oscar Pastor and Juan Carlos Molina. *Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling.* Springer, 2007.

[Pol14] Macario Polo. Using aspect-oriented programming for mutation testing of third-party components. In *CIBSE 2014: Proceedings of the 17th Ibero-American Conference Software Engineering*, pages 247–260, 01 2014.

[PURT21] Macario Polo-Usaola and Isyed Rodríguez-Trujillo. Analysing the combination of cost reduction techniques in android mutation testing. *Software Testing, Verification and Reliability*, page e1769, 2021.

[RDIR19] M. T. Rossi, M. De Sanctis, L. Iovino, and A. Rutle. A multilevel modelling approach for tourism flows detection. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 103–112. IEEE Press, 2019.

[RP14] Moisés Rodríguez and Mario Piattini. Software product quality evaluation using ISO/IEC 25000. *ERCIM News*, 2014(99), 2014.

[RRD+18] Alejandro Rodríguez, Adrian Rutle, Francisco Durán, Lars Michael Kristensen, and Fernando Macías. Multilevel modelling of coloured petri nets. In *Procs. of 5th International Workshop on Muli-Level modelling (MULTI'18) - MODELS Workshops*, 2018.

[SA20] Ahmad A. Saifan and Adnan Alzyoud. Mutation testing to evaluate android applications. *International Journal of Open Source Software and Processes (IJOSSP)*, 11:23–40, 2020.

[Seb16] Robert W. Sebesta. *Concepts of Programming Languages.* Pearson, 2016.

[SHTB07] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007.

[SN14] Iman Saleh and Khaled Nagi. Hadoopmutator: A cloud-based mutation testing framework. In *Procs. of the ICSR 2015: International Conference on Software Reuse for Dynamic Systems in the Cloud and Beyond*, pages 172–187, Cham, 2014. Springer International Publishing.

[SNF10] Gustavo Breder Sampaio, Filipe Ribeiro Nalon, and Jugurta Lisboa Filho. Geoprofile - UML profile for conceptual modeling of geographic databases.

In *Procs. of the 12th International Conference on Enterprise Information Systems (ICEIS 2010)*, pages 409–412, 2010.

[SZ09a] Hossain Shahriar and Mohammad Zulkernine. Mutec: Mutation-based testing of cross site scripting. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, pages 47–53. IEEE Computer Society, 2009.

[SZ09b] Mark Strembeck and Uwe Zdun. An approach for the systematic development of domain-specific languages. *Software: Practice and Experience*, 39(15):1253–1292, 2009.

[TA19] Bedir Tekinerdogan and Ethem Arkin. Pardsl: a domain-specific language framework for supporting deployment of parallel algorithms. *Software & Systems Modeling*, 18(5):2907–2935, 2019.

[Thea] The Open Geospatial Consortium. OpenGIS Simple Feature Access - Part 1: Common Architecture. http://www.opengeospatial.org/standards/sfa. visited on 2020-06-21.

[Theb] The Open Geospatial Consortium. OpenGIS Web Feature Service 2.0 Interface Standard. http://www.opengeospatial.org/standards/wfs. visited on 2019-07-02.

[Thec] The Open Geospatial Consortium. OpenGIS Web Map Server Implementation Specification. http://www.opengeospatial.org/standards/wms. visited on 2019-07-02.

[Tom69] R.F. Tomlinson. A geographic information system for regional planning. *Journal of Geography*, 78(1):45–48, 1969.

[TSCDLR07] Javier Tuya, Ma José Suárez-Cabal, and Claudio De La Riva. Mutating database queries. *Information and Software Technology*, 49(4):398–417, 2007.

[URRH17] M. P. Usaola, G. Rojas, I. Rodríguez, and S. Hernández. An Architecture for the Development of Mutation Operators. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 143–148, March 2017.

[VBD+19] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart CL Kats, Eelco Visser, and Guido Wachsmuth. Dsl engineering-designing, implementing and using domain-specific languages. 2013. *URL: http://voelter. de/dslbook/markusvoelter-dslengineering-1.0. pdf, http://dslbook. org*, 2019.

[VDE+] Markus Völter, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart Kats, and Eelco Visser. Wachsmuth. 2013. dsl engineering: Designing, implementing and using domain-specific languages.

[WD04] Michael F Worboys and Matt Duckham. *GIS: a computing perspective*. CRC press, 2004.

[WNH+17] Fan Wu, Jay Nanavati, Mark Harman, Yue Jia, and Jens Krinke. Memory mutation testing. *Information and Software Technology*, 81:97 – 111, 2017.

[Woo93] Martin R Woodward. Mutation testing—its origin and evolution. *Information and Software Technology*, 35(3):163–169, 1993.

[ZF09] C. Zhou and P. Frankl. Mutation testing for java database applications. In *2009 International Conference on Software Testing Verification and Validation*, pages 396–405, 2009.