Developing Web-based Geographic Information Systems with a DSL: Proposal and Case Study

Suilen H. Alvarado, Alejandro Cortiñas, Miguel R. Luaces, Oscar Pedreira and Angeles S. Places

Universidade da Coruña, Centro de Investigación CITIC, Laboratorio de Bases de Datos, Facultade de Informática, Elviña s/n, 15071 A Coruña, Spain
{s.hernandez,alejandro.cortinas,luaces,
opedreira,asplaces}@udc.es

Abstract

In this paper, we present a declarative domain-specific language (DSL) for the development of Geographic Information Systems (GIS). GIS applications manage information with a spatial component, usually in the form of points, lines, polygons, or variants of these basic data types, in domains where the spatial information plays a central role. They provide the user with different functionalities on different application domains, but they are usually developed according to a common architecture and using a common set of technologies. Hence, they share a significant number of elements that make some aspects of their development quite repetitive. Our DSL allows developers to specify the entities, geographic layers, and maps of the applications using a declarative language. Then, the specification is transformed into a working GIS application. We present the language, its implementation, and a case study on two sample projects that allowed us to evaluate the resulting software, paying special attention to the savings in the development effort.

Keywords: Domain specific language; geographic information systems.

1 Introduction

Geographic information systems (GIS) manage entities with a spatial component that plays a central role in the system's functionalities. The spatial component of the entities is usually represented as a point, a line, a polygon, or a variant of these data types. An important functionality of any GIS is visualizing information using maps that structure the data they show into layers. Typical applications of GIS include logistics management, civil infrastructure management, or guided navigation. Also, the rise of small mobile devices with geolocation capabilities has broadened the application range of GIS.

The application domain determines the functionalities and data model of each GIS. For example, guided navigation may be mandatory in a logistics software, but irrelevant in other domains. Despite the functional differences between two GIS, they all share common concepts, architecture, tools, and technologies. Both commercial and open-source solutions for GIS are implemented according to standards published by the Open Geospatial Consortium (OGC)[1]. Therefore, current GIS are composed of components that can be easily substituted. Hence, their development can be quite repetitive, at least in the components that implement aspects that appear in many application domains. Based on this motivation, we believe that the development of GIS can be improved by applying model-driven engineering techniques.

Model-driven engineering (MDE) is an approach to software development in which models play an active role beyond just describing the system [4,16]. Models are created according to a *metamodel* that defines the elements that can be used to describe the system. The models can be transformed into models at lower levels of abstraction, or into source code, according to a set of transformation rules. 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.

In this article, 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. 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, and a use example. A preliminary version of this

[1] OGC: http://www.opengeospatial.org/

article was published as a conference paper in [1]. This work extends that preliminary publication with the implementation of the GIS-DSL tool, and a case study in which we develop two sample applications to analyze the resulting software products.

The rest of the article is structured as follows: in Section 2 we review background and related work. In Section 3 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 4 details the implementation of the tool that allows us to transform a system specification in GIS-DSL into source code. Section 5 presents a case study on two sample applications. Finally, Section 6 presents the conclusions of the paper and lines for future work.

2 Background and Related Work

Model-driven engineering is a software development approach that promotes the use of models as active artifacts in all stages of software development. 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). 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) [4], 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 standard defined by the Object Management Group (OMG)[2] on its particular vision of MDD is the *model-driven architecture* (MDA)[3] [16], 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. 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 [7, 8, 15]. Although implementing a DSL can require a

[2] OMG: http://www.omg.org

[3] MDA: http://www.omg.org/mda

significant effort, their main benefit is that they allow us to specify/implement a system with significantly less effort.

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 [14], or to model performance tests for web applications [3]. The systematic mapping presented in [11] 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 [5,6], we 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. Our 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 article, 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, [13] and [18] presented an UML profile to support GIS-related concepts in UML conceptual models. Later, [9] 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[4] initiative) include metamodels that cover different concepts and application areas. Kutzner [12] addressed the model-driven transformation of geospatial data according to different metamodels that can present differences between them.

3 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

[4] `http://inspire.ec.europa.eu`

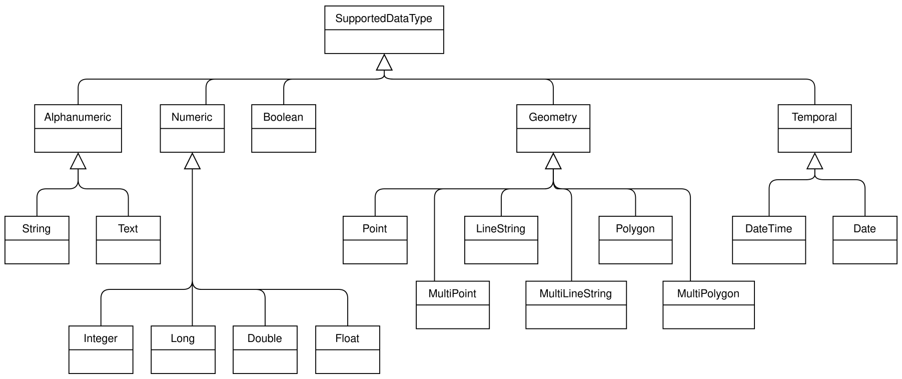

Figure 1 Supported data types for our metamodel of web-based GIS.

the web in these standards since many network-based services were defined, such as the *web map service*[5] (WMS) or the *web feature service*[6] (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.

3.1 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 1 shows 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.

[5] `http://www.opengeospatial.org/standards/wms`
[6] `http://www.opengeospatial.org/standards/wfs`

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 [13] and [18] includes data types to represent networks and other spatial phenomena. In [12], 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[7] or Oracle Spatial[8]. 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 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*

[7] `https://postgis.net/`
[8] `https://www.oracle.com/database/technologies/spatialandgraph.html`

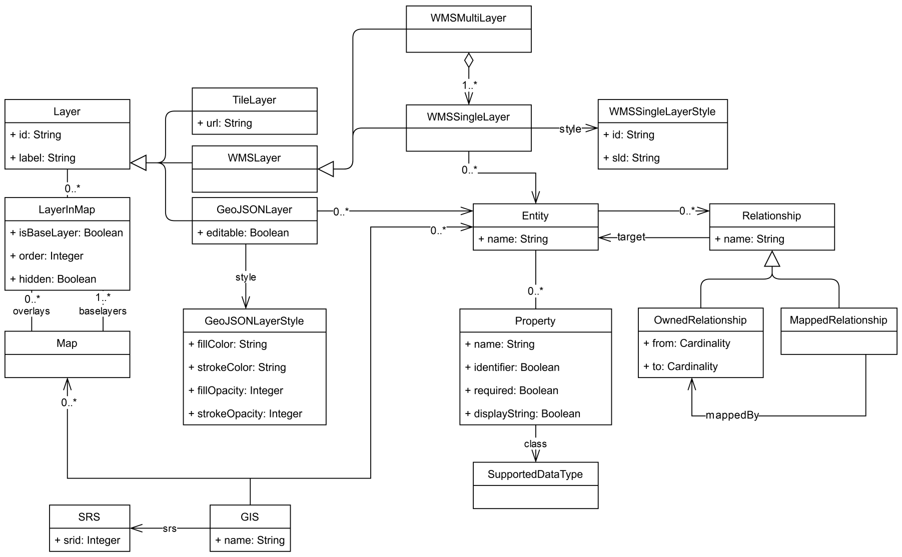

Figure 2 A metamodel of web-based GIS (reduced version, adapted from [1]).

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 Figure 2 formalizes these concepts and how they relate to each other.

Figure 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 followed the complete MDA architecture, since we transform the specifications of the DSL directly into code. However, the DSL

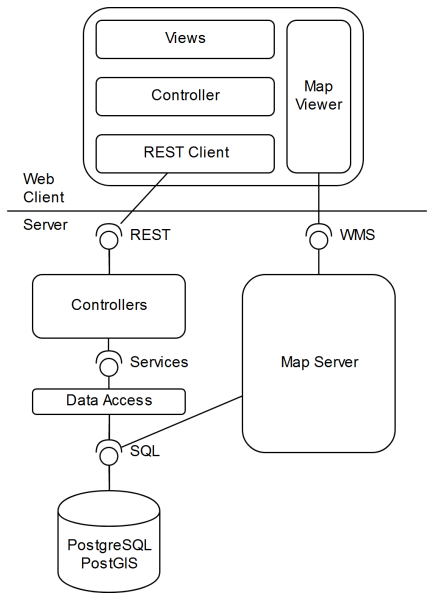

Figure 3 Typical architecture of a web-based GIS (reproduced from [1]).

would still be valid if we were interested into a MDA-based implementation, able to generate applications for different platforms.

3.2 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 [7, 19].

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 1 `CREATE GIS` sentence.

The domain can be specified using the sentence `CREATE ENTITY` (see Listing 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 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 2 `CREATE ENTITY` 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). 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 `CREATE LAYER` sentence

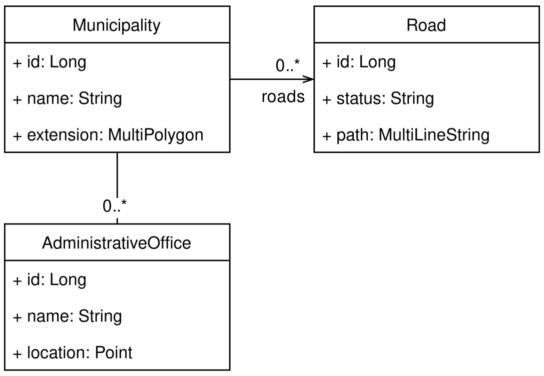

Figure 4 Data model of the example application (reproduced from [1]).

The `CREATE MAP` sentence allows us to define the map viewers (see Listing 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 4 `CREATE MAP` sentence.

Finally, 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 5 `GENERATE GIS` sentence.

3.3 Use Example

To illustrate the use of the DSL, we present an example with a basic GIS application. 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 centres, 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 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 6 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"
```

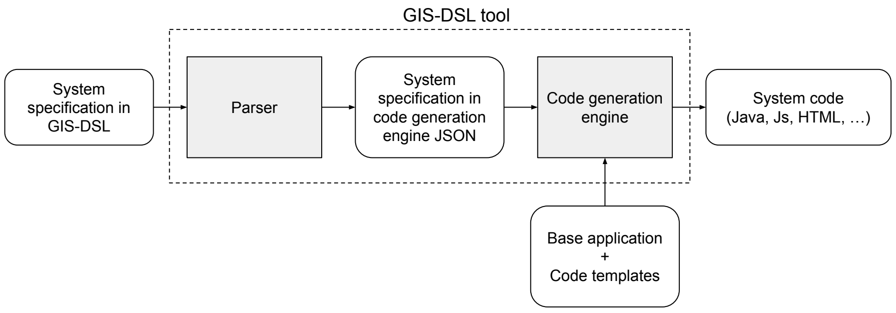
GIS-DSL tool
System specification in GIS-DSL
Parser
System specification in code generation engine JSON
Code generation engine
System code (Java, Js, HTML, ...)
Base application + Code templates

Figure 5 Architecture of the GIS-DSL code generation engine.

```
);

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 6 Example of application defined by the DSL.

4 Implementation of the DSL

The implementation of GIS-DSL is based on the use of the *generation engine* presented in [5]. 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 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.

4.1 GIS-DSL Parser

The parser was implemented with ANTLR[9]. 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 7 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)?
     OPAR_SYMBOL
```

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

```
    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 7 GIS-DSL grammar.

As an example, Listing 8 shows the specification of an entity *AdministrativeOffice*. The parser transforms this specification into an intermediate JSON file, shown in Listing 9.

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

Listing 8 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 9 Specification of *AdministrativeOffice* in JSON.

4.2 Code Generation Engine

The generation engine[10] used in our tool has been previously designed and developed to generate code from a system definition using a set of annotated code templates [5]. The generation 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 [10]. There are a set of operations related to *feature models* [2], 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.

4.3 Generated Code

Figure 6 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

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

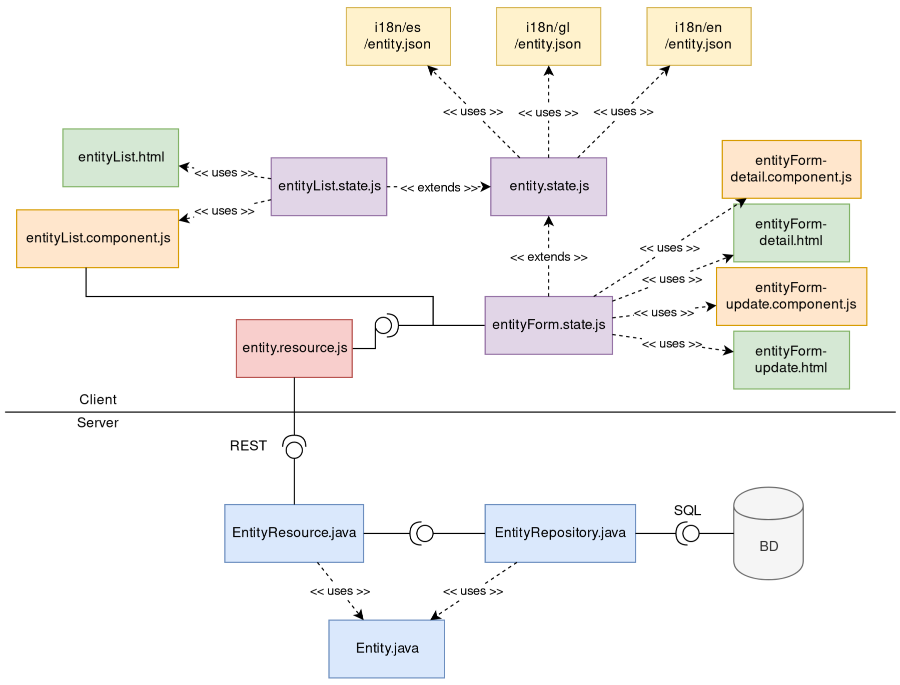

Figure 6 Low-level components of the generated source code.

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.

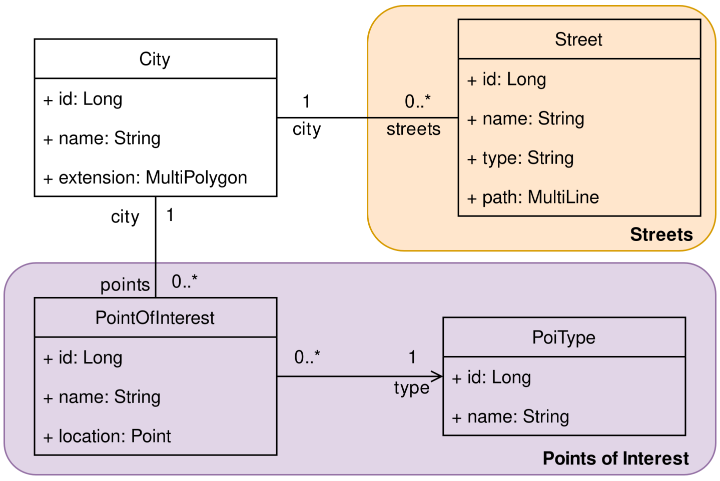

Figure 7 Class diagram of the first example project.

5 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.

5.1 Sample Project 1: Points of Interest

Figure 7 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.

5.2 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 8 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*

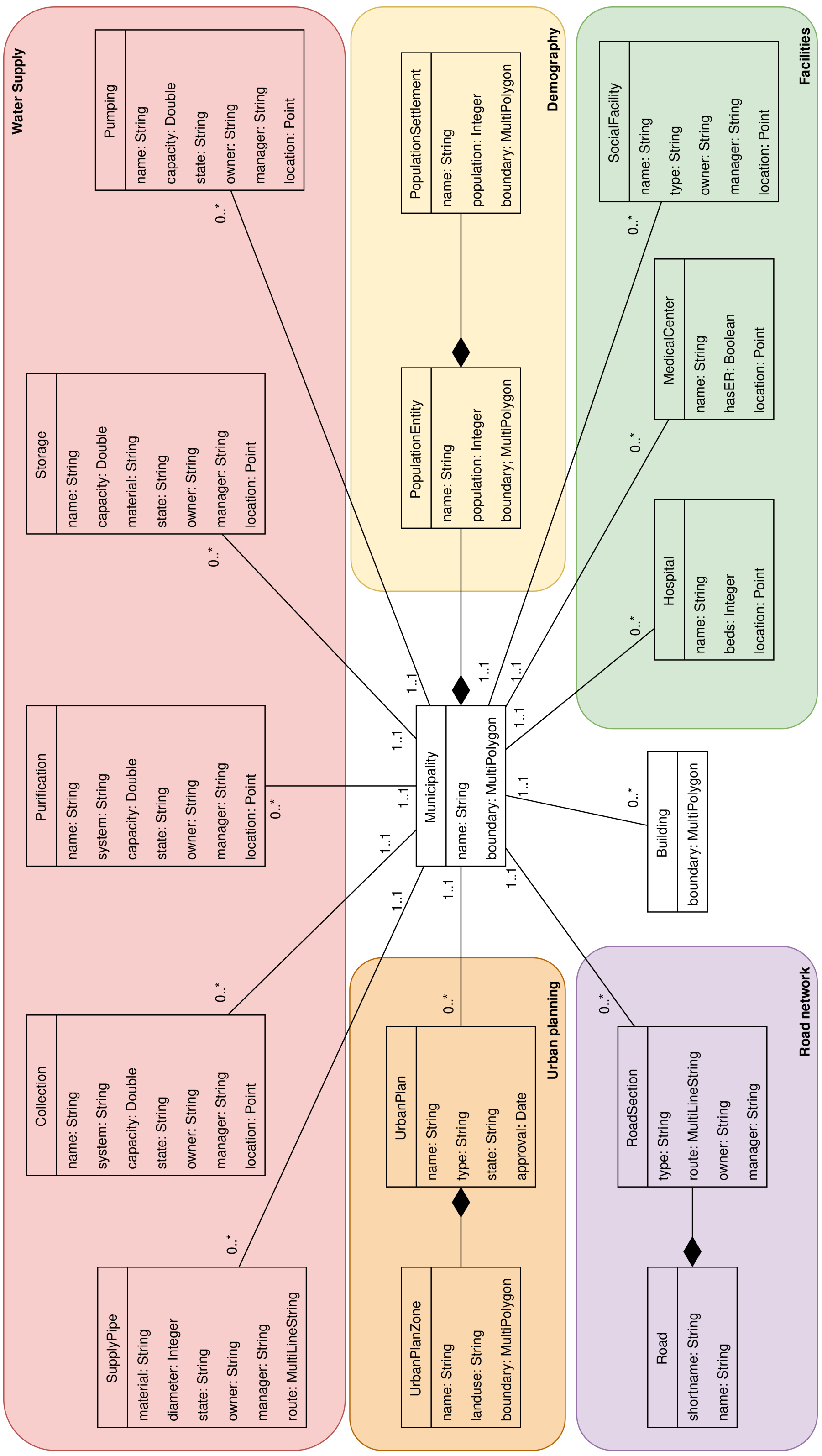

Figure 8 Class diagram of the second example project.

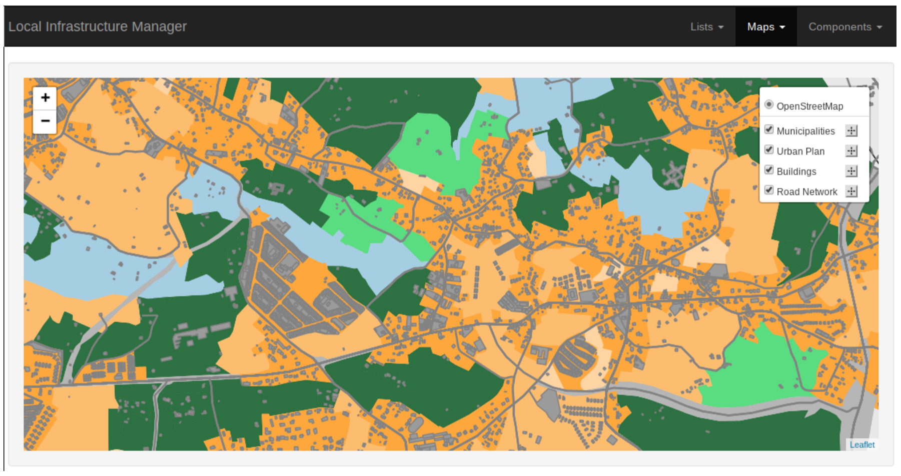

Figure 9 Example of a map defined for the product described in Section 5.2.

(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 Figure 9 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*).

5.3 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

Table 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 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

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 1 shows the characteristics of the two sample projects in terms of the number of entities, properties, relationships, maps, and layers. Table 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 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 of the fact that 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 [17], and even the satisfaction level of programmers using the DSL and users of the resulting software.

6 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 article, 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 taking into account 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.

ACKNOWLEDGEMENTS

Partially funded by: Xunta de Galicia FEDER-UE CSI grants: ED431C 2017/58 (Grupo de Referencia Competitiva), and IN852A 2018/14 (ConectaPeme, project GEMA); Ministerio de Economía-AEI FEDER-UE grants: Datos 4.0 (TIN2016-78011-c4-1-R), and Flatcity (TIN2016-77158-C4-3-R); and Ministerio de Ciencia e Innovación-AEI FEDER-UE grant BIZDEVOPS (RTI2018-098309-B-C32).

References

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

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

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

[18] Gustavo Breder Sampaio, Filipe Ribeiro Nalon, and Jugurta Lisboa Filho. Geoprofile - UML profile for conceptual modeling of geographic databases. In *Procs. of the 12^{th} International Conference on Enterprise Information Systems (ICEIS 2010)*, pages 409–412, 2010.

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