Creating web-based GIS applications using automatic code generation techniques *

Nieves R. Brisaboa, Alejandro Cortiñas, Miguel R. Luaces, and Oscar Pedreira

Universidade da Coruña
Laboratorio de Bases de Datos
A Coruña, Spain
{brisaboa, alejandro.cortinas, luaces, oscar.pedreira}@udc.es

Abstract. Geographic Information Systems (GIS) have increased its popularity for some time now, specially in the context of mobile devices. There are many disciplines and companies improving their workflow by using GIS on devices with geolocation features. To satisfy the emergent demand, lots of web-based GIS applications are being developed. These applications diverge in their target and context, but they all share a common set of the features. For some time an effort has been carried out to define standards in GIS, and currently the level of interoperability between GIS software assets is the highest ever. Given that there is a need to create web-based GIS applications sharing a set of features and that, thanks to the standards, GIS technologies are interoperable, it is not only possible but desirable to apply strategies of reuse, mass-customization and software generation to develop web-based GIS applications.
This work summarizes the design of a tool, GISBuilder, for the semi-automatic generation of web-based GIS applications. GISBuilder is a Software Product Line (SPL) with enhanced capabilities through the usage of a *scaffolding*-based transformation engine, which is able not only to assemble static software assets but to generate product-specific code.

Keywords: web-based geographic information systems, software product lines, model-driven engineering, scaffolding

1 Introduction

The technologies used to implement Geographic Information Systems (GIS) [10,19,11] used to follow different and incompatible conceptual, logical and physical data models. For example, a simple concept like the data type *polygon* had inconsistent definitions between GIS technologies, making interoperability almost

* Funded by MINECO (PGE & FEDER) [TIN2016-78011-C4-1-R, TIN2016-77158-C4-3-R, TIN2015-69951-R, TIN2013-46238-C4-3-R, TIN2013-46801-C4-3-R]; CDTI and MINECO [Ref. IDI-20141259, Ref. ITC-20151305, Ref. ITC-20151247]; and FPI Program [Ref. BES-2014-068178].

impossible. The standardization process carried out by the Open Geospatial Consortium (OGC) and International Organization for Standardization (ISO) in the last years has helped to solve this problem defining a set of standards for GIS that are currently followed by most software libraries. Nowadays, web-based GIS applications are very similar to each other, and they share not only functional features but also most of the technologies. For example, there are two major open source alternatives to implement a web map viewer, OpenLayers or Leaflet, and almost all the web map viewers are implemented with one of them. The repetitive appearance of the same features in every web-based GIS, the existence of many software artefacts following international standards that implement these features, and the fact that most web-based applications also share the same technologies make suitable the application of techniques for the automatic generation of this kind of products, web-based GIS.

Software engineering has put much effort on facilitating and improving software reuse. The research field of Software Product Lines (SPL) focuses on reusing the same software artefacts on different products that share features: each feature is implemented once, and the resultant software component is shared between all the products with the feature [7,16]. However, the actual implantation of SPL in industry is very low [3] and focused mainly in embedded systems [18]. Similarly, the research field of Model Driven Engineering (MDE) applies the advantages of modelling to software engineering activities [3]. The main concepts in MDE are models, which are simplified representations of the reality focused on a concrete domain, and transformations, which are manipulative operations over these models. Even though direct application of MDE in the industry is low, there are frameworks like Ruby on Rails or Grails based on the *scaffolding* technique that is used frequently in the software industry to speed up software development by generating code from its specification. Therefore, *scaffolding* is somehow an informal application of some MDE principles.

We have used a hybrid approach, based on SPL and on MDE, to design a tool, named GISBuilder, for the semi-automatic generation of web-based geographic information systems. In order to apply both approaches simultaneously, we have created a SPL engine based on *scaffolding* that generates the source code of a web-based GIS dynamically. As far as we know, this is the first attempt to do such two things. The remainder of this paper is organized as follows. Section 2 explains background concepts of the techniques used in GISBuilder, some previous work and motivation. Section 3 describes the products we aim to build. Section 4 details the architecture of our tool, which is complemented by Section 5, focused on the transformation engine, and by Section 6, which presents a use case. Finally, conclusions and future work are explained in Section 7.

2 Related Work

The evolution of the software development industry has been directed primarily to increment the layers of abstraction between the pure machine-code and the way a developer must describe the desired behaviour for a software asset. Assem-

bly languages were followed by high-level procedural languages like Fortran, Lisp, Cobol or C. Then, new paradigms like Object Oriented Programming followed with languages such as C++ or Java. Nowadays, software frameworks like the Spring framework provide for higher levels of abstraction using techniques such as inversion of control, dependency injection, or aspect-oriented programming to save the programmer thousands of lines.

One of the latest concepts in this direction is *scaffolding*, which is a popular technique in many trending software development frameworks, starting with Ruby on Rails in 2005. A *scaffolding* engine generates code from an specification or model created by a developer and a set of pre-defined templates. This way, most of the repetitive and generic code of the applications is automatically written, and the developer can start its work from a mid-stage or half-built architecture. *Scaffolding* is not limited to any context, and it can be used to generate any kind of code, from the user interface of an application to the documentation.

We can see *scaffolding* as an informal application of Model-Driven Engineering. MDE combines domain-specific modelling languages, which formalize the application structure, behaviour, and requirements within particular domains, with transformation engines and generators that analyse certain aspects of models and then synthesize various types of artefacts [17]. This is, from a set of models defined for an application, and through a series of transformations, some code is generated specifically for the product. However, this code is usually not replicated in any other product unless the models are the same. Therefore, MDE is useful to reduce the cost of developing a single product, but it is not as useful in the case of families of similar products.

The field of Software Product Lines [1] is specialized in sets of products sharing some common assets and features. In SPL, the features of a product family are modelled using a *feature model*, which represents the variability of the platform [8], and implemented in components or software assets. To build a new product, an analyst selects the features desired and the components related to them are assembled together. So, instead of strictly generating code, a SPL only handles inclusion or rejection of code blocks.

In [5], we identified the features and components of a generic web-based GIS product, and we designed a SPL to generate these products. We also concluded that the data model of each product has to be specified by the analyst because each application requires different data. Furthermore, all the source code related to the data schema must be automatically generated during the assembling process, even the creation of the database tables. However, current SPL implementation techniques, most of them shown on [1,14], are not suitable for dynamic code generation from the specifications of the products. Even though MDE and scaffolding are appropriate for this task, as far as we know, these techniques have not been used to implement SPLs so far.

3 Product Architecture

In this section we describe the products our tool is able to produce. We have identified the functional and non-functional features for a generic web-based GIS after analysing existing GIS applications with different scopes and features, such as [6,15,4,13,12]. We have also determined which software assets are required to provide all the features identified, and which product architecture is the adequate. The products of our platform have three main characteristics:

- Each product is based in a **complex data model** that is specifically designed for the product. The data model is composed by a set of entities, with their own properties and relationships among them. The different entities are used in the web application as the basis for listings, reports, creation and modification forms, and map layers.
- The products share a set of **common elements to any web-based application**: hierarchical menus to structure the contents, static HTML pages to display information that is not included in the data model, dynamic pages displaying listings and forms of entities from the data model, and private sections and functionalities available only to authenticated users with specific roles.
- Some pages of the product **display geographic information using a map viewer**. Each product may define its own collections of map layers, visualization styles, and maps. Furthermore, each product may define one or more map viewers, each one with its own configuration that includes the visualization type (e.g., the map viewer may be embedded in some other content, it may be shown in full-page mode, or the map viewer panning bounds may be limited) or the selection of the different tools that can be enabled (e.g., zooming and panning the map, showing a form for the selected map element, etc.)

Figure 1 shows the functional architecture of our products. It uses the classical three-layer architecture pattern composed of a *user interface*, responsible for the interaction with the user; a *processing layer* which contains all the functionality defined for the GIS; and a *model management* layer, responsible for physical data storage and data management. We use three shades of grey in the figure to classify the different modules according to how much they change for each product. The darker ones are *reusable software assets* that can be used in any product without any change in their code because they are not designed for a specific product but instead they can be used by any of them. Modules filled with white must be generated specifically for each product (e.g., menu structure or data model). Finally, those with a medium shade of grey are external software assets connected to the products through some API, so they are totally independent of each product (e.g., the database management system). The different modules are described next.

Reusable software assets. The reusable software assets of the products are the following:

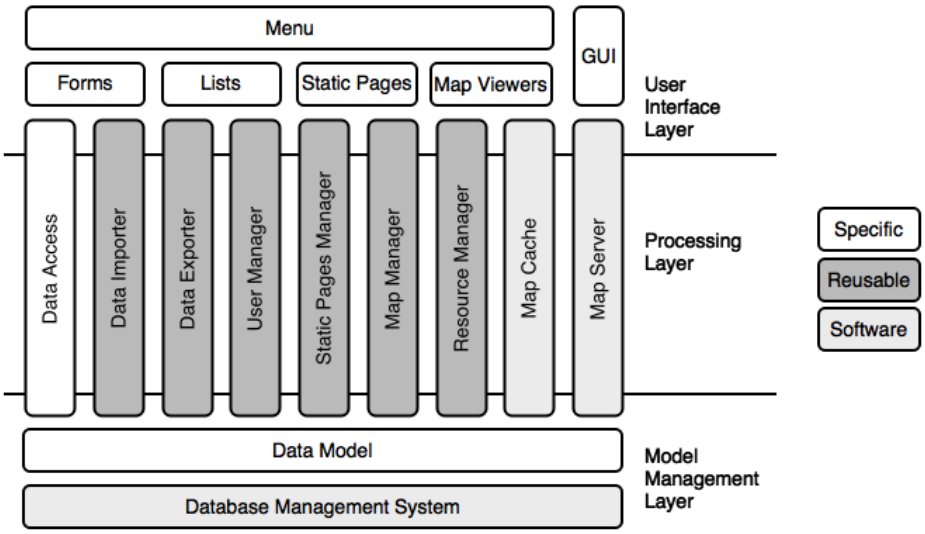

Fig. 1. Functional architecture of the products

- **Data Importer**. Even though the products provide a way for the final user to input data directly into the different entities using web forms, in many cases this is not enough because there are applications with lots of data in which using the manual way is not acceptable. Focusing on GIS, manually loading the data is simply not doable because this would mean that the final user has to draw the geographic objects manually. Therefore, our products can include a component for loading files directly in the applications. The files supported are the most common: *comma-separated values* (CSV), *Excel and LibreOffice spreadsheets*, *shapefiles*, and *raster* file formats. The component also allows the user to map the data from the files into the different properties and entities of the data model (i.e., the attribute names do not have to be the same).
- **Data Exporter**. Exporting listings into CSV files or spreadsheets and the current map visualization into a downloadable image or *shapefile* is a common feature in information systems. This component provides this feature.
- **User Manager**. Most web-applications handle different types of users, providing different functionalities depending on the role of the connected user. They also allow the user to sign up, and there is an administrator who can list and edit the existing set of users. This component provides for all these features.
- **Static Pages Manager**. Even though many content pages in web applications are static in the sense that they do not change during the lifetime of the application, some other content in the pages must be editable. This component provides this functionally, and it can also used by an administrator user to create or delete content pages.

- **Map Manager**. Geographic data, maps, layers and styles, can be specified by the analyst when the product is generated. When this is not enough for the requirements of a product, this component lets the administrator user change the different maps, layouts and styles used by the application on runtime. The changes are applied to the different map viewers existing in the product.
- **Resource Manager**. Most web applications have to deal with multimedia resources, whether they are *pdf* files, *images*, *music* files or *videos*. This component offers the administrator user an interface to define new categories of resources, and to define the attributes for the resources of each category. The rest of the users of the product have features to list, edit and create resources in any of the existing categories.

Specifically generated. The following modules must be specifically generated for each product:

- **Data Model**. The specification of the entities made by the analyst is translated into Java entity classes and tables in the database. The mapping between these two elements is also done from the relationships between the different entities established by the analyst.
- **Data Access**. As in most web applications, the database is accessed and modified through a series of services that connect the user interface with the storage layer, separating the business logic. The code required by theses services is generated for each product depending on the specification of the entities, and it consists of REST services to communicate with the user interface and the data access objects and services to communicate with the database management system.
- **Forms**. Forms are the standard way to edit elements stored in the database and to create new ones. The forms are created from the entities choosing which properties can be edited in each form.
- **Lists**. Lists are created in a similar way to forms, that is, by selecting which properties of each entity should be shown in the listing. There are certain additional options for the lists, such as sorting, filtering or searching over the elements listed.
- **Static Pages**. The analyst can create a set of *static pages* using a WYSIWYG (*What You See Is What You Get*) editor. These pages can be accessed by the final user through the menus, and one of them can be chosen as the welcome page.
- **Menu**. Most of the features of any software are accessed trough a menu. The different components and modules provide a set of views that allow the user to access their features. Some of the views are the *authentication page*, the *csv import page*, any of the *lists* dynamically generated, or any of the resources created in the resource manager. The analyst defines the menu structure combining three types of menu elements: a link to a view provided by any of the modules, an external link to any web page or a sub-menu element that groups another set of elements.

- **Map Viewers**. All the entities which have a geographic property can be visualized using a map viewer in the final application. To do that, the analyst must define those viewers and choose the options that the user will have over each entity. For example, a link to edit an element in the map can be set to go to one of the *forms* defined before.
- **Graphical User Interface** (GUI). Our tool generates not only the visual theme of the application but also lets the analyst decide about the position and number of menus of the application, and the position of the widgets provided by other components.

External software assets. The following modules are considered external to the product and deployed without modification:

- **Database Management System**. The storage layer of our products must be provided by an external component. Nowadays, most web applications store their data in a relational database. In our case, since we are managing data of geographical nature, we need to use a database management system with geographic capabilities such as PostGIS, Oracle Spatial or MySQL.
- **Map Server**. If the volume of data is not really high, a web-based GIS can be built without a map server drawing all geographic information on the client-side (and possibly fetching base layers from public services like OpenStreetMap or similar ones). However, in cases where the amount of data is high, or if we need to customize the styles of the layers, our product needs a map server connecting the database data with the map viewers.
- **Map Cache**. Rendering each layer every time a user accesses a map viewer can be very costly, specially with large sets of geographic data. With a map server cache the layers remain rendered for some time, so the processing and bandwidth cost are reduced.

In Figure 2 we show the technological architecture of the products generated by GISBuilder. All the technologies involved are commonly used nowadays by web developers. The web client is implemented using an open-source web application framework called Angular. It presents a modular design and takes advantages of patterns like inversion of control, which allows us to easily create variants of the products using different software artifacts. The map viewing technology chosen is Leaflet because it is lightweight, mobile-oriented and flexible, and it supports additional functionality by plugins. The server-side is based on Spring MVC: we implement the REST services with Spring controllers and the services that provide communication with the database are built using the dependency injection pattern of Spring. We also use Spring Security to support user authentication and access-control. The geographic information is served to the client by an internal map server (GeoServer) and a map cache (TileCache), even though external map services may also be used. Finally, Spring JPA and Hibernate are used as the data access technology and PostgreSQL (and PostGIS) is our alternative as database management system.

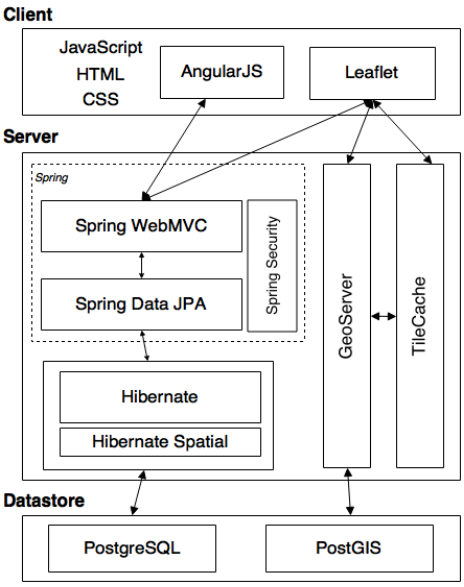

Fig. 2. Technological architecture of the products

4 GISBuilder

The GISBuilder architecture is shown in Figure 3, and it is consists of the *specification interface*, the *project repository*, the *transformation engine*, and the *component repository*.

The *specification interface* allows an *analyst* to define different products using two strategies: on the one hand, he or she can select the features included in the product as in any other SPL; on the other hand, the interface provides tools for the analyst to define the data model, the menu structure, and the lists, forms or map viewers that are included in the final product. Behind the scenes, the interface builds the product specification, an instance of the GISBuilder Domain Specific Language (DSL) represented as a JSON document, and validates it using JSON Schema.

The *project repository*, another GISBuilder module, is a database where all the product specifications (JSON documents) are stored. This way, an analyst can restore a previously defined product and generate it again, refine it, or create a new version of it.

When the analyst decides to generate the source code of a project, the JSON document representing the product specification is sent to the *transformation engine*. Then the *transformation engine* assembles the different components and generates the required code, using the reusable software assets and code templates from the *component repository* to achieve this. The output of the engine would be the source code of the product specified by the analyst. Being the *trans-*

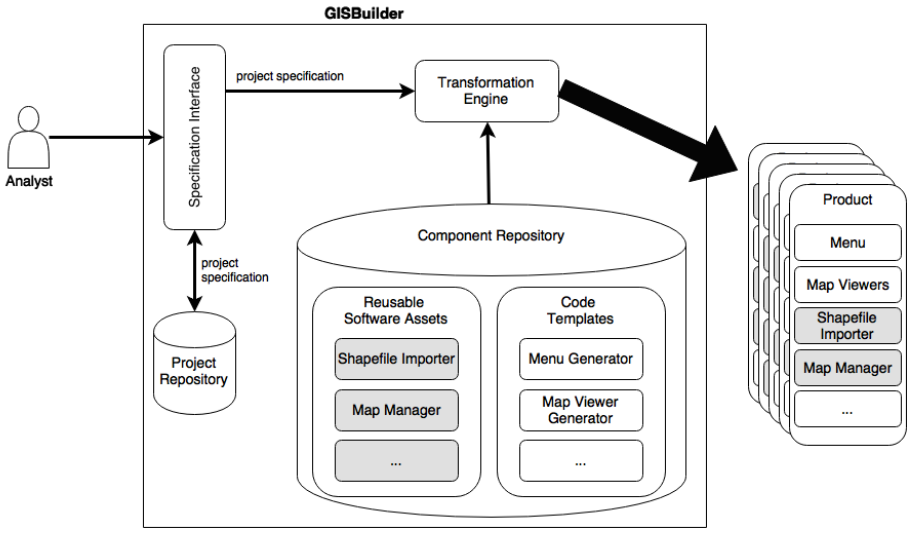

Fig. 3. Functional architecture of GISBuilder

formation engine the most important part of our system, we carefully explain it in Section 5.

In terms of the technology used to implement GISBuilder, using a framework that allows us to decouple the different modules is very beneficial. Also, there is more need for flexibility than stability, since our platform is an evolving set of artefacts whose components are supposed to grow in size and quantity, and it is not expected to be used by many people at the same time. That is the reason to implement it on Node.js, a platform build on Chrome's JavaScript runtime for easily building fast applications. It is lightweight, independent of operating systems and IDEs and currently one of the most important platforms, with growing popularity. Node.js provides for a huge flexibility that facilitates the integration of its libraries and applications.

In SPL it is common that the specification interface is simple or almost non-existent, since it is only used to select which features are included in the product. However, due to the complexity of our products and their definition, GISBuilders *specification interface* is a web application implemented with Angular that communicates with a Node.js server via REST. This server handles the interaction with the *project repository* and with the *transformation engine*. Since the project specification is represented with a JSON document, the technology chosen to store these specifications is MongoDB, a document oriented database that handles the data precisely in this format. The *transformation engine* is also a Node.js tool, so the integration is straightforward using an API provided by the engine. Lastly, the *components repository* is nothing more than a directory with files of the code of every asset and template, which are accessed directly by the *transformation engine*.

5 Transformation Engine

Figure 4 shows the structure of our transformation engine, which consists of three different components: the *feature model manager*, the *file manager*, and the *template engine*. The *transformation engine* defines an API used by the *specification interface* to generate the different products. It also provides a small command line utility so the engine can be used independently of the tool, which specially useful when developing or debugging the platform.

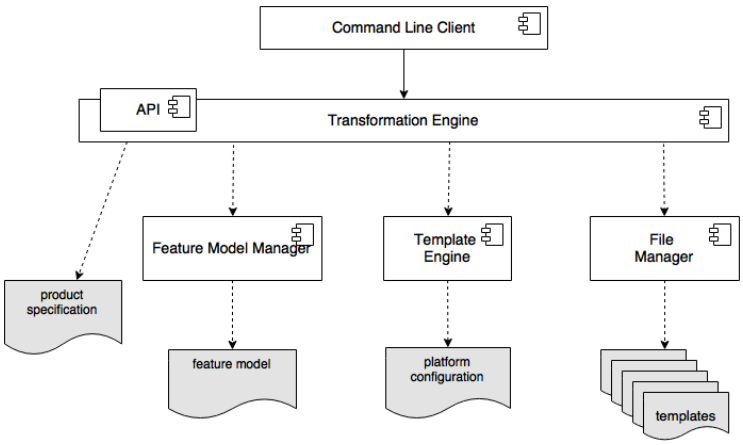

Fig. 4. Component diagram of the transformation engine

The ***feature model manager*** main purpose is to manage the feature model of the platform and to check whether the selection made by the analyst is correct. This is performed using the well-known operation *Valid Product* of SPL[2]. It also provides some of the other analysis features to improve the maintainability of the platform and the products, like finding features used in the code templates that are not defined in the feature model.

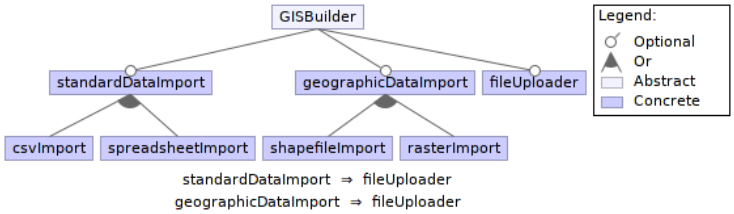

Fig. 5. Excerpt of the GISBuilder feature model

The feature model is represented following the specification of Feature IDE [9], and it can be loaded from a file in three different formats: XML, JSON or YAML. It can also be created programmatically, a feature expected to be used in the future to allow changing the feature model graphically from the *specification interface*. In addition to features, the feature model also allows to define constraints defined with logical operators. In Figure 5 a excerpt of our feature model is shown.

The ***file manager*** handles all task related to the access to the templates. For example, it allows the *transformation engine* to walk through every template of a directory recursively and apply changes to them. It also detects when a binary file is found in order to skip processing it and just copy it to the output.

The latest and more important part of the *transformation engine* is the ***template engine***. In a SPL, there is usually a derivation engine handling the process of assembling the product from a set of features using the reusable components. In MDE, transformations are applied to models to generate new models in different levels of the architecture, until one of these transformations generates source code. Our ***template engine*** is able to handle these two kinds of operations, being able to generate products from:

- **SPL-like assemblable components**: for each product, it can be specified a set of features to be included in it, like in a classic SPL. Our engine takes the components implementing these features and assembles them into the final product. The set of components corresponds the *reusable software assets* from Section 3.
- **MDE-like model transformations**: a set of models are used to generate each product. These models specify the data model, layers and maps of each product, as well as many other generic parameters like graphical interface options. Our engine transforms these models into specific modules included in the final product. These are the specifically generated modules from Section 3.

In order to handle this duality, we have designed the *template engine* to be based on the *scaffolding* technique. It treats every module as a set of templates, no matter whether the module is a reusable component that needs just to be assembled or whether its code must be specifically generated from the application models. Thus, the *component repository* described in Section 4 is nothing but the templates of every module, including the code for the *reusable software assets* and the *code templates* that are used to transform the modules into product specific code.

In Figure 6 a excerpt of the Java code for the data importer is shown. In the *static* block we can see some variation of the code to produce depending on the selected features. Template annotations are defined as comments of the programming language in which the template is written and its content can be any JavaScript code. Thus, they do not interfere with the compiler, IDE or validation tool used by the developer of the platform code.

Besides annotations related to which code blocks are included in the product, our template engine allows using variables and complex control sequences in

```
private static final Set<String> validExtensions = new HashSet<String>();

static {
    /*% if (feature.shapefileImport) { %*/
    validExtensions.add("shp")
    /*% }  %*/
    /*% if (feature.csvImport) { %*/
    validExtensions.add("csv")
    /*% }  %*/
    /*% if (feature.spreadsheetImport) { %*/
    validExtensions.add("xls")
    /*% }  %*/
}

@Component
public class DataImporter() {
    // ...
}
```

Fig. 6. Annotated Java class

```
/*%@ return entities.map(function(en) {
    return {
        fileName: en.name + '.java',
        context: en
    };
}) %*/
package es.udc.lbd.gisbuilder.model.domain;

@Entity
@Table(name = "t_/*%= context.name.toLowerCase() %*/")
public class /*%= context.name %*/ {
    /*% context.properties.forEach(function(prop) {
            var propertyClass = prop.class;
            var validGeomTypes = ['Point', 'MultiLineString', 'MultiPolygon'];

            if (validGeomTypes.find(propertyClass) != null) { %*/
    @JsonSerialize(using = CustomGeometrySerializer.class)
    @JsonDeserialize(using = Custom/*%= propertyClass %*/Deserializer.class)
    @Column(columnDefinition="geometry(/*%= propertyClass %*/, 4326)")
        /*% } %*/
    private /*%= propertyClass %*/ /*%= normalize(prop.name) %*/;
    /*% }); %*/
}
```

Fig. 7. Simplified excerpt of model transformation template

the code. Moreover, the developer of the platform can even create JavaScript functions to use within the annotations, or use temporal variables to store data used more than once. In the Figure 7 a simplified template to generate entities of a product is shown. The template specifies how to create Java classes for each entity defined by the analyst. Inside the class, the code for each property of the entity is created depending on its specification. The latter is an example of a *code template* generating product specific code, whereas the former example is just a *reusable software asset* with an small variation depending on the sub-features selected. Even with the latter template simplified, we can see the difference of complexity between the two types of templates.

6 Use Case

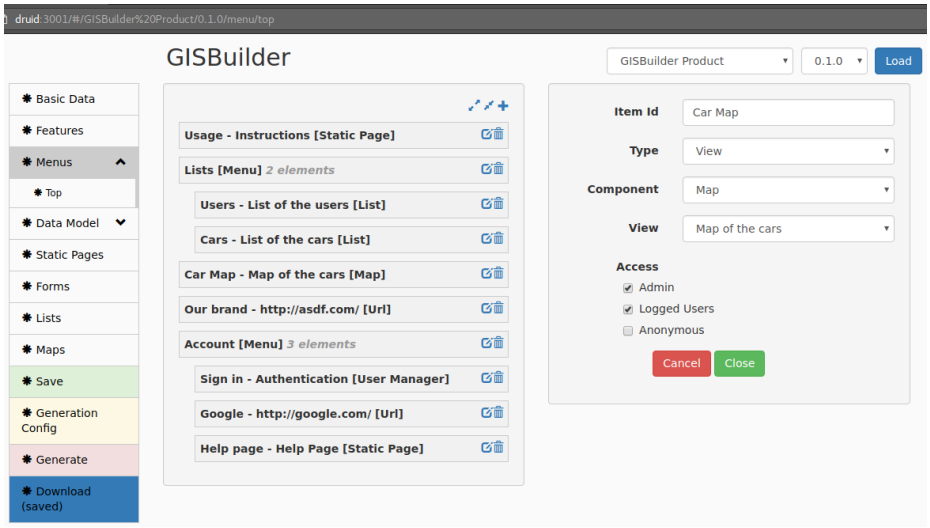

Fig. 8. Menu structure design page

In this section we describe the process an analyst follows to specify and create a product. The current implemented version is still a prototype, but it is already able to generate production-ready products with menus, static pages, forms, lists, simple maps and a small set of components, including the *data importer* and the *resource manager*.

When the analyst starts defining a new product, the first step is to define some general information about the product, such as the name and the version (because our tool supports different versions of the same product). The analyst must also choice which component and view will be the welcome page, and the set of languages in which the application will be available. The next step is to

decide which *reusable components*, from the ones shown in Section 3, must be included in the product. Some of the components have their functionality split between a set of subcomponents which can be selected independently. That is, if the product only needs the feature to import CSV, there is no need to include the rest of the features of the *data importer* component . However, there are some components depending on others, and the application checks that the required ones are selected.

The menu structure is designed using the interface shown on Figure 8 where we can see a list of elements and some of them grouped in sub-menus. The analyst can restrict the visibility of any element by choosing which kind of role must the logged user have. The different static pages required by the product can be defined during the specification stage, or once the application is running (if the *static pages manager* is included).

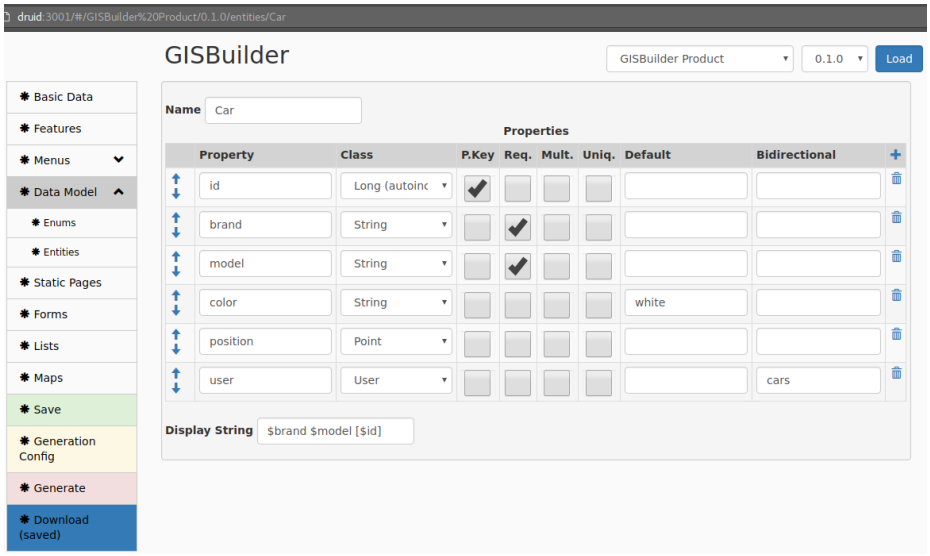

Fig. 9. Entity definition page

One of the most important elements the analyst must specify is the data model. Each entity existing in the application is defined using the interface shown in Figure 9. The properties of every element are detailed, indicating their names, types and some extra typical options such as if a concrete property is required or optional. If an entity has a relationship with another entity, the property determining this relation is also defined.

From the set of entities defined in the specification of the data model, the analyst can create forms, lists and maps. The analyst decides which properties of the entities can be edited in forms and which are shown in lists and maps. Both list and maps can be customized with static filters, and the functionality

of dynamic filters can be enabled over some maps, so the user can apply its own filters. In the case of the maps, there are also options regarding the layers to be shown on the map.

7 Conclusions and Future Work

In this paper we have presented a functional tool called GISBuilder able to generate semi-automatically production-ready web-based geographic information systems. To design the tool, we have reviewed the current state of art of GIS, we have identified the functional and non-functional requirements of a web-based GIS product, we have designed an up-to-date architecture for web-based GIS and we have implemented a transformation engine following the *scaffolding* technique that allows us to mix SPL and MDE concepts in the generation of our products.

As future work, we are designing and implementing the rest of the components required by our products and designing a methodology to confront the evolution of the products and the platform code supported by *git*, a Version Control System.

References

1. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines. Springer (2013)
2. Benavides, D., Segura, S., Ruiz-Corts, A.: Automated analysis of feature models 20 years later: A literature review. Information Systems 35(6), 615–636 (sep 2010)
3. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice, vol. 1. Morgan & Claypool Publishers (sep 2012)
4. Brisaboa, N.R., Cotelo-Lema, J.A., Fariña, A., Luaces, M.R., Parama, J.R., Viqueira, J.R.R.: Collecting and publishing large multiscale geographic datasets. Software: Practice and Experience 37(12), 1319–1348 (Oct 2007), `http://onlinelibrary.wiley.com/doi/10.1002/spe.807/abstract`
5. Brisaboa, N.R., Cortiñas, A., Luaces, M.R., Pol'la, M.: A Reusable Software Architecture for Geographic Information Systems based on Software Product Line Engineering. In: Proceedings of the 5th International Conference on Model & Data Engineering (MEDI 2015). vol. 9344, pp. 320–331. Springer (2015)
6. Brisaboa, N.R., Luaces, M.R., Places, Á.S., Seco, D.: Exploiting geographic references of documents in a geographical information retrieval system using an ontology-based index. GeoInformatica 14(3), 307–331 (Jul 2010), `http://link.springer.com/article/10.1007/s10707-010-0106-3`
7. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2002)
8. Kang, K.C., Cohen, S.G., Hess, J.a., Novak, W.E., Peterson, a.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Distribution 17(November), 161 (1990), `http://www.sei.cmu.edu/reports/90tr021.pdf`
9. Kästner, C., Thum, T., Saake, G., Feigenspan, J., Leich, T., Wielgorz, F., Apel, S.: FeatureIDE: A tool framework for feature-oriented software development.

In: 2009 IEEE 31st International Conference on Software Engineering. pp. 611–614. IEEE (2009), http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5070568
10. Laurini, R., Thompson, D.: Fundamentals of Spatial Information Systems. Academic Press, London, 1 edition edn. (Apr 1992)
11. Longley, P.A., Goodchild, M.F., Maguire, D.J.: Geographic Information Science and Systems. Blackwell Publ, Hoboken, NJ, edicin: revised. edn. (Mar 2015)
12. Luaces, M.R., Brisaboa, N.R., Paramá, J.R., Viqueira, J.R.: A Generic Framework for GIS Applications. In: Kwon, Y.J., Bouju, A., Claramunt, C. (eds.) Web and Wireless Geographical Information Systems. pp. 94–109. Lecture Notes in Computer Science, Springer Berlin Heidelberg (Nov 2004), http://link.springer.com/chapter/10.1007/11427865_8, dOI: 10.1007/11427865_8
13. Luaces, M.R., Pérez, D.T., Fonte, J.I.L., Cerdeira-Pena, A.: An Urban Planning Web Viewer Based on AJAX. In: Vossen, G., Long, D.D.E., Yu, J.X. (eds.) Web Information Systems Engineering - WISE 2009. pp. 443–453. Lecture Notes in Computer Science, Springer Berlin Heidelberg (Oct 2009), http://link.springer.com/chapter/10.1007/978-3-642-04409-0_43, dOI: 10.1007/978-3-642-04409-0_43
14. Meinicke, J., Thüm, T., Schröter, R., Benduhn, F., Saake, G.: An overview on analysis tools for software product lines. In: Proceedings of the 18th International Software Product Line Conference on Companion Volume for Workshops, Demonstrations and Tools - SPLC '14. pp. 94–101. ACM Press, New York, New York, USA (sep 2014), http://dl.acm.org/citation.cfm?id=2647908.2655972
15. Places, Á.S., Brisaboa, N.R., Fariña, A., Luaces, M.R., Paramá, J.R., Penabad, M.R.: The Galician virtual library. Online Information Review 31(3), 333–352 (Jun 2007), http://www.emeraldinsight.com/doi/full/10.1108/14684520710764104
16. Pohl, K., Böckle, G., Linden, F.V.D.: Software Product Line Engineering, vol. 49. Springer-Verlag New York, Inc. (2005), http://www.springerlink.com/index/10.1007/3-540-28901-1
17. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. Computer 39(2), 25–31 (Feb 2006), http://dx.doi.org/10.1109/MC.2006.58
18. Weiss, D.M., Clements, P.C., Krueger, C.W.: Software Product Line Hall of Fame. SPLC 2006: Proceedings of the 10th International Software Product Line Conference pp. 237–237 (2006), http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1691614
19. Worboys, M.F., Duckham, M.: GIS: A Computing Perspective, Second Edition. CRC Press, Boca Raton, Fla, 2 edition edn. (May 2004)