GISBUILDER: A FRAMEWORK FOR THE SEMI-AUTOMATIC GENERATION OF WEB-BASED GEOGRAPHIC INFORMATION SYSTEMS[*]

Nieves R. Brisaboa, Laboratorio de Bases de Datos, Facultad de Informática, Universidade da Coruña, 15071 A Coruña, Spain, brisaboa@udc.es

Alejandro Cortiñas, Laboratorio de Bases de Datos, Facultad de Informática, Universidade da Coruña, 15071 A Coruña, Spain, alejandro.cortinas@udc.es

Miguel R. Luaces, Laboratorio de Bases de Datos, Facultad de Informática, Universidade da Coruña, 15071 A Coruña, Spain, luaces@udc.es

Oscar Pedreira, Laboratorio de Bases de Datos, Facultad de Informática, Universidade da Coruña, 15071 A Coruña, Spain, oscar.pedreira@udc.es

Abstract

Geographic information systems (GIS) have played a central role in many web-applications in the last years. Geolocating users and other things (e.g., cars, drones) is a functionality that has been included in all kind of information systems from social networks to mobile workforce management systems. Even though each GIS is used in a particular area with its own objectives, they all share multiple features and requirements. Moreover, there has been a strong effort on standardization that has led to the creation of many different software artefacts with similar functionality implemented with diverse approaches, but still interoperable between them. Therefore, it is possible to apply techniques based on intensive software reuse, such as software product line engineering (SPLE) and model-driven engineering (MDE) to reduce the development effort involved in the creation of a web-based GIS application.

In this work, we present a software framework based on SPLE and MDE techniques for the semi-automatic generation of web-based geographic information systems. The core of the framework is a SPL derivation engine that creates products with source code that is partly based on a repository of components, and partly dynamically generated using scaffolding (a specific MDE technique). Furthermore, we focus on building products using current web technologies with high-quality source code in terms of flexibility and sustainability.

Keywords: geographic information systems, software product line engineering, general-purpose software architecture, scaffolding.

[*] Funded by MINECO (PGE & FEDER) [TIN2013-46238-C4-3-R, TIN2013-46801-C4-3-R, TIN2013-47090-C3-3-P]; CDTI and MINECO [Ref. IDI-20141259, Ref. ITC-20151305, Ref. ITC-20151247]; and Xunta de Galicia (co funded by FEDER) [GRC2013/053].

1 INTRODUCTION

Geographic Information Systems have represented a major sector of the software development industry for more than 30 years, and they have been used in diverse applications in a wide variety of fields. Even though each application has its own particularities, there are a lot of common features that appear every time (e.g., filtering elements and searching on a map, importing layers of geographic information, generating lists from a map, selecting elements from the map, etc.). At first, GIS technologies that were used to implement applications followed different and incompatible conceptual, logical and physical data models. For example, even a simple concept like the data type *polygon* had inconsistent definitions between GIS technologies, making interoperability almost impossible. In the last years, Open Geospatial Consortium (OGC) and International Organization for Standardization (through ISO/TC 211 and the 19100 set) have defined a set of standards for GIS that are currently followed by most software libraries and that have helped to solve this problem. Therefore, web-based GIS applications developed nowadays are very similar to each other, and usually they share not only 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. Independently of the way the software is programmed or specified, each software product is built from scratch, going through all the common phases on software development from elicitation of requirements to maintenance of the product. Even when the same company works on products with similar functionalities, the whole process is repeated for each one of them. This approach leads to high development costs in order to produce high quality products.

Software Product Lines Engineering (SPLE) is a recent software engineering discipline focused 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 (Clements and Northrop 2001, Pohl et al. 2005). This is achieved by separating the development of reusable software assets and the development of actual applications. The former is the platform of the Software Product Line (SPL), which is modelled as a set of features. The latter, the applications, are called products and each one is built automatically by adapting and assembling the software assets from the platform. The main motivation of SPL is to reduce the high costs of the development of high quality products, to improve this quality and to reduce the *time to market* for each product. It is natural that some products require specific functionalities not covered by the platform, but this is not a problem since the products are provided as source code that can be extended.

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. Particularly, we have used a hybrid approach, based on SPLE and on Model Driven Engineering, to design a framework (called GISBuilder) for the semi-automatic generation of web-based geographic information systems. In order to apply both approaches simultaneously, we have created a SPL derivation engine based on scaffolding (a specific MDE technique) 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 briefly describes the products GISBuilder produces. Section 4 details the architecture of our framework, which is complemented by Section 5, focused on the derivation engine, and by Section 6, on the component repository. Section 7 presents a use case which shows the usefulness of GISBuilder. Finally, conclusions and future work are explained in Section 8.

2 RELATED WORK

In the software development industry, automatic (or semi-automatic) generation of code is an important topic. With the objective of reducing development costs, it has been always desirable to increment the layers of abstraction between the pure machine-code and the way a developer must specify the functions desired for a software asset. At the end, each new layer just generates the code of the inferior one from a specific syntax the developer must know.

The first of this abstraction layers above machine-code were assembly languages. They allow the developers to use some readable syntax, specific to each particular processor architecture, which would be assembled into machine code. Modern languages like Fortran, Lisp, Cobol or C allow the developer to write code in an even more readable syntax that is independent of the architecture. Before running the code, it must be compiled into processor-specific assembly language. New layers of abstraction were also provided by languages following new paradigms, like Object Oriented Programming on languages such as C++ or Java, instead of the classic procedural paradigm. On top of programming languages, *ad-hoc* techniques focused on practical application and without academic formalism provide for higher level of abstraction. For example, Spring framework, used for developing Java web applications, uses techniques such as Inversion of Control, Dependency Injection, or Aspect Oriented Programming to save the programmer thousands of lines.

A recent concept that also avoids writing lot of generic code is *scaffolding*. Scaffolding is a popular technique in many trending software development frameworks, starting from Ruby on Rails, which first version was on 2005. A scaffolding engine generates code from a set of pre-define templates and a specification provided by the developer. While the scaffolding concept applied to software development is relatively new, in fourth generation programming languages (4GL) there was a similar feature in database code generators, such as Oracle's CASE Generator or dBase IV APPSGEN tool. However, 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. Furthermore, the scaffolding process can occur either in runtime or in design time. The counterpart of scaffolding in academia is Model Driven Engineering (MDE), which is a very mature research field that has been proposing, for a long time, conceptual formalisms to describe software and to generate it automatically.

Another field which focuses more on reusing existing code than in generating new code is Software Product Lines Engineering. When a SPL is designed following a feature oriented approach (Apel et al. 2013), each product is defined as a set of *features*. The set of features that can be selected for a product is what we call *variability* of the SPL. Each feature will need one or more *components* or software *assets* to implement its functionality. The complete set of components is the platform, as explained in Sect. 2. To build a new product, the analyst selects some features 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. Therefore, since Software Product Lines usually do not generate code but they only select code to be added or removed from the products, scaffolding techniques have not been used to implement them so far.

In a previous work (Brisaboa et al. 2015) we have identified the features and components that should belong to a generic web-based geographic information system, and we have designed a SPL to generate these products. One of the conclusions of that work is that in order to be able to generate really generic products, the data model of each product has to be specified by the analyst. There is no point in automatically generating web-based GIS if they all manage the same data. Therefore, the analyst must be able to specify the data model and all the code related to the data schema must be automatically generated during the assembling or derivation process, even the creation of the database tables. As future work of Brisaboa et al. (2015), we indicated that the implementation of the SPL was pending, and that we were evaluating several techniques to perform it. After that evaluation, we can see that current SPL implementation techniques, most of them shown on (Apel et al. 2013) and

(Meinicke et al. 2014), are not suitable for dynamic code generation from the specifications of the products.

An important drawback of the advances made in MDE and SPLE is that they are usually far away from the current technological proposals, which makes the success of these techniques reduced since development teams consider the field as technologically outdated. Furthermore, there are certain techniques for implementing SPL that make the product code really complex and not legible. We propose to apply the conceptual formalisms behind SPLE and MDE to create a framework capable of generating software products, but both the framework and the products generated are implemented using the most up-to-date technology.

Furthermore, legibility of the final code is a common problem in SPL (Kästner et al. 2008). Given that we do not aim at making static final products with our platform, but rather accelerating the development of products that may require to be extended to include some specific functionality not present in the platform, we want it to produce code that can be easily manipulated. Therefore, the code of the generated products will be *well-written* and easily extendable, in contrast to the result achieved by using most of existing techniques for implementing SPL that make the product code really complex and not legible.

3 PRODUCT ARCHITECTURE

The products built by GISBuilder are web-based GIS applications. These products have three main characteristics:

- They are based on a complex data model that is defined by the SPL analyst. This data model is composed as a set of entities that will be used in the web-based GIS as the basis for listings, reports, creation and modification forms and map layers.
- They have the common elements in a 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 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 or 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.)

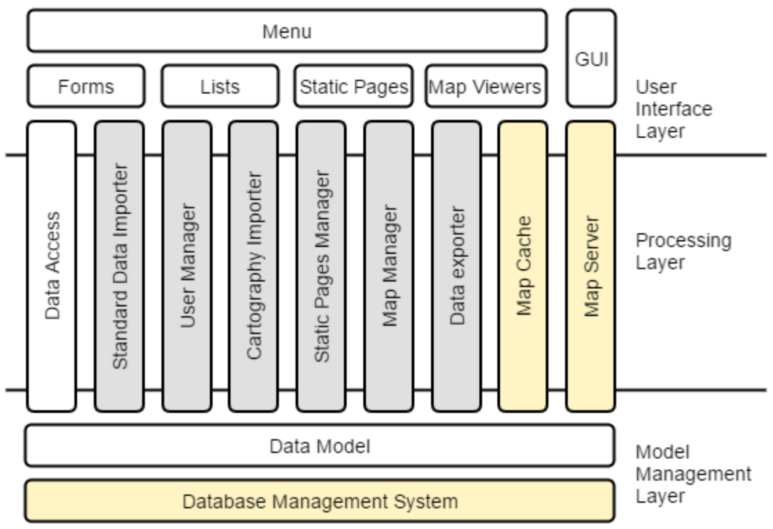

Figure 1: Functional architecture of the products

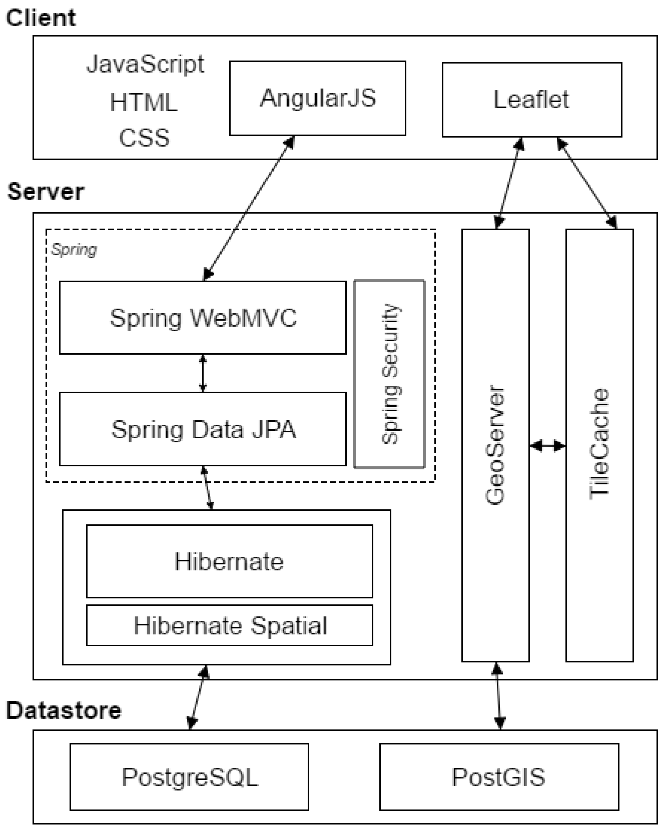

Figure 2: Technological architecture of the products

Figure 1 shows a functional architecture of the products generated by GISBuilder. We have defined a three-layer architecture 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. The components are painted with three colours according to their type. The grey ones are those that do not benefit substantially from code generation. They are totally or partially included in the final products, depending on the feature selection. Those in yellow are similar, but they are not components specifically implemented for our products but technological artefacts that are configured on deployment, like a DBMS or a Map Server. Lastly, functional components seen in white in the figure must be dynamically generated from the specific needs of every product because different products do not have the same "menu", neither the same "data model" nor "data access" features.

Regarding the technological architecture of our products, which can be seen in Figure 2, we have selected the common modern technologies used nowadays for building web applications with GIS features. In the user interface layer we have decided to use an open-source web application framework called AngularJS because it presents a modular design which enables us to define a flexible configuration and to implement each variant with different software artefacts. We have also decided to use Leaflet as the map viewing technology because it is mobile-oriented and light in terms of download size, and at the same time it is flexible enough to support additional functionality by means of plugins. The processing layer is based on Spring MVC. The REST services used to communicate the user interface with the processing layer are implemented with Spring controllers, and the services that provide communication with the model management layer 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 is used as the data access technology and PostgreSQL and PostGIS are selected as the Database Management System.

4 GISBUILDER ARCHITECTURE

In Figure 3 shows the architecture of the platform, which is mainly divided in four modules: specification interface, project repository, derivation engine and component repository. The complete workflow for the generation of a new product, with the participation of each module, is described next.

The analyst is the person in charge of creating a new product. He or she is supposed to be aware of the features the product must have, as well as the data schema that must be managed, the desired graphical aspect and the structure and navigability of the web application. The analyst interacts with the specification interface, which provides a friendly way for configuring the product to be made.

In SPL it is common that the specification interface is simple or almost inexistent, since it is only used to select which features are included in the product. In our case the analyst must choose not only over the variability of the product but also he or she must parameterize the components, so we have designed a proper web application as the specification interface. In this application, there are two groups of functionalities. On one side, we can manage the product specifications: create a new product specification, save it persistently in the repository, load it again later, duplicate an existing product specification or remove a product. On other side, we have the proper specification interface, where we can select the set of features of a product, parameterize each component, and finally send the specification to the derivation engine so we can get its source code.

The output of the specification interface is the complete specification of the products, including variability selection and parameterization. This is represented as JSON documents following a predefined, but flexible, JSON Schema. This JSON documents are the ones stored in the product repository and they serve as input for the derivation engine. Also, the source code of the components of the platform must be annotated accordingly to this schema.

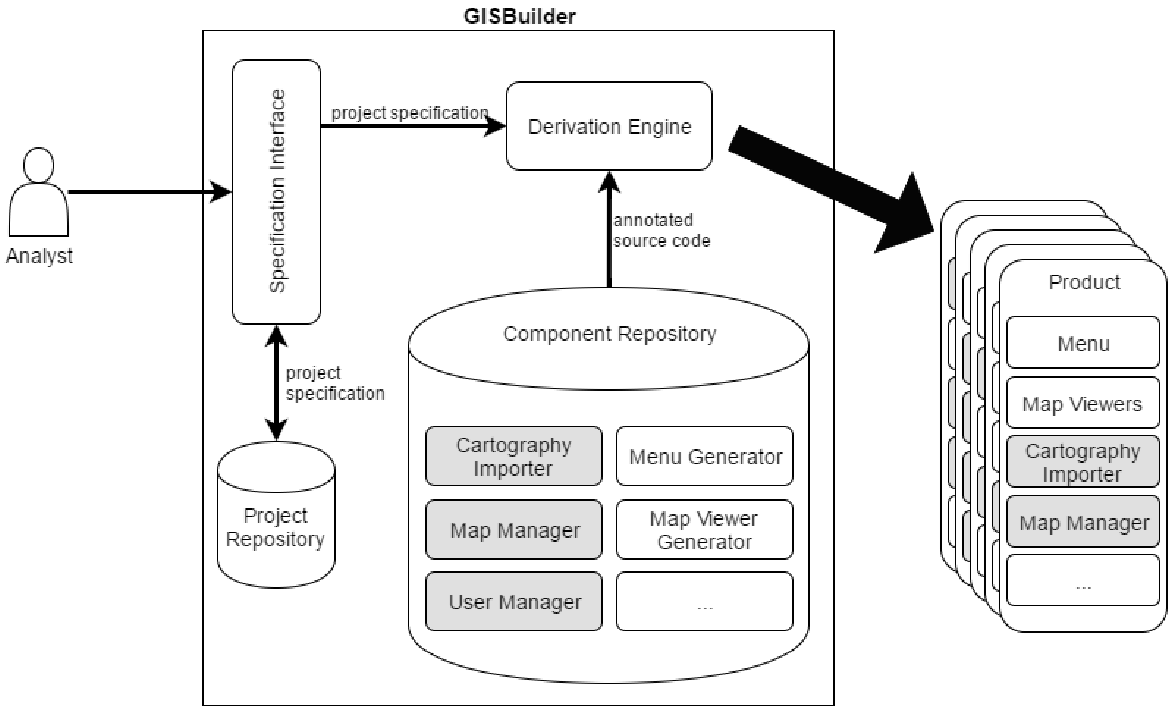

Figure 3: Architecture of the platform

As mentioned above, our platform has the capability to store the specification of the products. The analyst may start the process of specifying a product, save it after a while and close the application. The partial specification will be ready to be loaded again in the interface so the analyst can continue its work at any other moment. Furthermore, the platform also handles versioning of the products, allowing the analyst to upgrade the version of a product and to change its specification while keeping all the specification for the previous versions. This is achieved by storing all the project specifications in the project repository, sent from the specification interface.

When the analyst finishes the configuration of a product and chooses to generate it, the JSON document of its specification is sent to the derivation engine. The generation process begins with the validation of the received JSON. If this validation is passed, the derivation engine starts assembling the required components from the repository according to the specification needs. The output of this process is the source code of the product. These two modules are described with more detail in two separate sections.

In terms of the technology used to implement GISBuilder, using a platform that allows us to decouple the different modules is very beneficial. Also, there is more need for flexibility than stability, since the framework 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.

Node.js is a platform build on *Chrome's JavaScript runtime* for easily building fast applications. It is lightweight, independent of operating systems and IDEs and one of the currently most important platforms, with growing popularity. Node.js provides for a huge flexibility that facilitates the integration of its libraries and applications. It meets the necessary characteristics to be used as the technology platform for our framework.

GISBuilder's specification interface is a web application implemented with AngularJS that communicates with a Node.js server via REST. This server handles the interaction with the project repository and the derivation 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 integration with the specification interface server, achieved with a light Node.js API, is straightforward. The derivation engine is a Node.js tool with an API used, again, by the specification interface server. It also has a wrapper to be used directly by node shell (can be used in any OS), which comes handy for pure SPL with no parameterization, since there is only one function required in this case. Lastly, the components repository is nothing more than a directory with the files of the code of every component. There is no technology itself for this, since each code file is just a text template with the annotations.

5 DERIVATION ENGINE

In order to design a truly useful derivation engine we have been analysing the existing alternatives from the point of view of a software development team, pointing out the more problematic common issues. Our engine has been designed with the next requirements in mind:

- The output must be independent source code that can be taken and provided to a third party without any relationship. So, our derivation engine must provide static binding, allowing the derivation process to be done in the earliest step, before the compilation of the products (Apel et al. 2013).
- The source code of the generated products must be of good quality, as *well-written* as possible, legible and easily extended, and the effects of the process in the source code must be minimal. There are other implementation techniques like the Compositional ones or using Aspect Oriented Programming that affects seriously the final product: using composition, the legibility of the generated code is severely damaged (Kästner et al. 2008) and using AOP we force the components to be implemented with that technique.
- The code can be written in any programming language, so multi-language support must be provided.
- The usage of any IDE cannot be imposed by our tool. Using a specific IDE is a really big constraint that should not be imposed to the developers, especially if we want a multi-language tool. Besides that, during the analysis of the state of art we have found many tools hard to even try

since they are based on obsolete versions of Eclipse runtime platform, others that only work on a specific OS, and so on. Designing a tool with the minimum dependencies prevent this kind of problems in the future.

From these requirements we have designed and implemented a small library that generates final products with the smallest possible input: the feature model of the platform, a JSON document with the specification of a product and the annotated source code of the components.

Briefly explained, a feature model is a tree with all the features provided by a SPL. In the feature model (Czarnecki et al. 2005), besides the node-leaf relationship of any tree, certain restrictions may be specified and the selected features must complain them. For example, if we have the restriction “Subtract implies Add”, then if the feature “Subtract” is selected and the feature “Add” is not, the feature selection is invalid. Our derivation engine uses the feature model to validate the selection of features made by analyst. It may be provided in XML, compliant with the generated by FeatureIDE (Kästner et al. 2009), in JSON, using a schema we have defined, or even programmatically with an API provided by our engine.

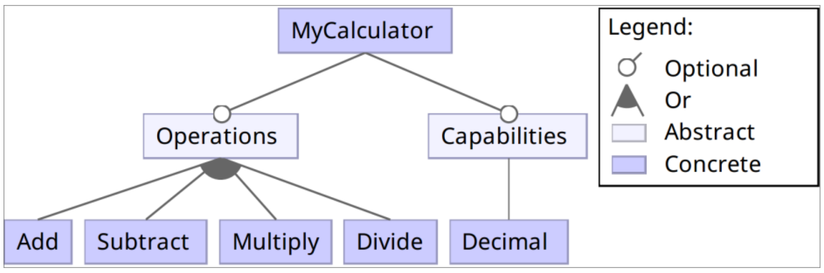

Figure 4: Feature Model example

About the specification of a product, the derivation engine itself does not have any convention regarding the schema in which it must be provided. As long as the source code of the components is annotated accordingly, the derivation process will work. However, we have defined a specific JSON Schema for our platform that includes both the selection of features and all the parameterization of the components made by the analyst. To save space, we do not explain it in this paper, since it has not any special value.

To understand the way the components are annotated, it is enough to know that each feature is a Boolean property on the path “feature” of the JSON, and that the parameterization specification is under in the path “data". For example, let’s assume that the analyst selects two features for a product, “add” and “divide”, and also its title can be parameterized. Then, *{ feature: { add: true, divide: true }, data: { title: “This is the title” } }* may be the specification JSON document.

Following the example, the components must be annotated with the schema of the specification in mind. The annotations are made using JavaScript syntax inside specific markers. Unlike in CPP, which is the most known pre-processor, these markers are flexible. That way, we can use code comments in any case. For example, in Java we annotate code like */*% if (feature.add) { %*/*, while in HTML the same annotation will be *<!--% if (feature.add) { %-->*, and in python it will be *#% if (feature.add) { %#*. As you can see, all of them are comments for the respective languages so they do not interfere with compilers or IDEs. All the JavaScript operators and functions can be used inside the annotations, like *forEach* iterator or logical conjunctions, which comes handy for the parameterization of components. To print one value on the final code, the annotation in HTML would be *<!--%= data.title %-->*, while in JavaScript would be */*%= data.title %*/*.

We are aware of the problems of annotation based approaches and the use of pre-processors to implement variability, as well as all the criticism from academia (Feigenspan et al. 2013, Le et al. 2011, Spencer and Collyer 1992). One of the big issues is that the annotated code can get really complex, depending on the granularity of the annotations. However, we prefer to handle this issue with a strict methodology when developing the components instead of worsening the legibility of the generated code, which is the real product. Furthermore, it is very significant that in industry CPP remains as one of the main alternatives (Hunsen et al. 2015, Medeiros et al. 2015), while the academic approaches are not very popular.

Our derivation engine, unlike pre-processors, is not a general purpose solution that happens to be use in this context, but focuses on SPL derivation, manages variability concepts, perform validation of specifications against the feature model and it is easy to use and to integrate with modern tools (being a Node.js library, not a Linux tool). It overcomes one of the main issues of CPP, obstructing syntax which interferes with IDEs, compilers, validators or any code related tool for a specific language[†], and this is achieved independently of the programming language code thanks to the flexibility of the annotation markers.

6 COMPONENT REPOSITORY

Over the components we made two classifications. On one side, we classify the components belonging the SPL according to the kind of features they provide to the system. In this way, we have *application support* components, *data input* components and *data exploitation* components.

On the other side, we classify them also in the way the “provide” to the final product. There are certain *static* components that, if selected, are included almost directly in the final code, like the “CSV Importer”; and there are other components that limit their area of effect to generate final code, without being themselves in the final products, like the “Menu Generator”. I.e., the code of the component that generates the menus of the products according to the parameterization is not included in the product, only the generated code for the menu is included. This *generator* components may be not be related with any *feature* of the platform when they are set only by parameterization. On the figures in this paper, the *static* components are shown in grey, while the *generator* components are shown in white. This division is not totally strict but more conceptual.

A list with the main the main components of the platform comes next. In it, the generator components are easily identified by their name. As you can see, we have more components than the ones mentioned in Brisaboa et al. (2015), and we have already included some powerful ones providing advanced features not as standard as to say they belong to any generic GIS. However, we have introduced them to represent that, while GISBuilder is based on building generic GIS products, the components do not need to be basic and it is pretended than new components with more specific but GIS related functionalities are added over time.

We can also see that the Map Server, Map Cache Server or the Database Management System are not included in this list of components. This is because these mentioned, and many more, are just technological components that may affect to the code of rest of the components (it is not the same to load the data from MySQL or from PostgreSQL) but, as components themselves, they are just included at the deployment stage.

6.1 Application support components

This block of components is not GIS related, but essential to any general purpose application. There are web applications that do not have menus, for example, but it is not usual.

Using the **menu generator**, the analyst may define one or various menus in the specification interface that will be included in the final product. The elements of the menu may link to a view of the provided

[†] It is really curious that even when there are better suited pre-processors to be used in the context of SPL, like GPP or GNU M4, *general purpose pre-processors* that handle different types of annotations, CPP remains the most known and used. Anyway, these tools share the remaining problems of CPP.

by the rest of the components (i.e., the authentication page, a form, a list, a map…), to an external URL or may be a submenu where more elements are disposed.

The graphical aspect of the product can also be chosen for each product with the **GUI generator.** There are various designs for the products, each one with its own configuration.

Most of web applications require managing the users, with different user roles with restricted access to some functionalities of the product. **User manager**, if included, allows that.

In case the product needs to have a database (there may be static applications that do not need one), the analyst indicates the set of entities and their properties, and the **data schema generator** will create all the required code. On each property there are many things to be selected, beyond its name:

- Type of the property. It can be a simple type like String or Integer, a geographic type like Point or Polygon, an enumerated (defined also for the product) or another entity, creating a relation between the two.
- Whether the property is a single item or a collection.
- If the item is single, it also may be mandatory (i.e., not null).
- Also certain constraints can be set, like whether a property is a primary key of the relation, or its value must be unique.

The **static pages** component has three subcomponents that will be used or not depending on the selected features for a product: a static pages generator, a static pages viewer and a static pages manager. The analyst may parameterize this component by defining the statics pages to be included in the product. In case the product do not use a database, the generator will create the pages specified by the analyst as pure HTML and CSS. In case the product use a database, the static pages viewer will be included and the defined pages will be stored in the database, so the viewer loads them dynamically. The last component, the manager, requires both database and user manager in the final product, and allows to manage the static pages on the product in runtime.

6.2 Data input components

With the **form generator**, the analyst may insert forms on the final product to create, view, edit and delete elements of the data schema. These forms can be linked by the menu, in some cases, or by other components. For example, in a list, each element listed may have a link to an edit form.

There are some components that manage the reception of data by *harvesting* it from other sources, like the **text data harvester**. Apart from getting texts from different sources and classifying them, this component has functionalities of geolocation of toponyms. It recognizes named geopolitical entities in the text and applies an algorithm for toponym disambiguation, linking each text with the set of concrete toponyms and their geographic positions. This component is targeted mainly to work with news or articles from websites, harvesting them via RSS or using a web crawler.

Another harvester is the **data streaming** component, which allows the application to receive data from REST and other sources in any format or schema. Over the data received, some filtering and classification operations can be done. For example, the user of the product may configure a source that sends data in JSON format, discard the items sent with a property not equal to a value, and store everything else in a distributed database. It also can make triggers from the result of some filters, like sending an email if a value is reached in the data.

Another way for input data to the products is by using importation features, like the provided by the **standard data importer**. Our platform generates products with custom data schemas and it also provides a way for input the data directly in the web application via forms. But there are applications with lots of data, and it is not acceptable to load the data only that way. So there is also a component that imports data from CSV or spreadsheets, allowing the user to make the mapping from the file to the different entities managed by application. Regarding geographic data, many GIS allow users to load data from shapefiles or raster files, which is the function provided by our **cartography importer**.

6.3 Data exploitation components

This block of components is composed, so far, with viewers and exportation tools. With the **list generator**, the analyst can include some list views in the application linked through the menu from the data schema specified before. The **map viewer generator** is its equivalent for data with geographic properties and allows the analyst to include a series of maps, linked through the menu, where the geographic data is displayed. The map viewers have generic features like zooming, panning, measuring distances, clicking on an element to see its details on a popup, getting a permalink to the view… which may be activated by each map individually. There is also a component that allows the final users to edit the existing views and to create new ones, the **map manager**.

While the texts harvested could be shown using regular map viewers, there is a component, the **harvested texts viewer**, which provides a specific way to visualize them. Each text is displayed by its positions on a map, and the ones with the same or near positions appear clustered. The view allows the user to load a specific text by clicking over it, to make both textual and geographic searches and to export the results to a list.

From a list or a map, our products have the capability to export the visualized data in different formats. For example, a map may be exported as an image, and a list as a CSV file. This functionality is provided by various subcomponents grouped as **data exporters**.

7 USAGE SCENARIO

Let us assume we want to create an application that handles a high number of sensors located all over a city. Each sensor communicates to little switchboards with internet connectivity which take the raw data and send it to server after structuring it as JSON documents. The information gathered by the sensors is variable and there are several different types of sensors so, even when the data goes through a switchboard that structures it, the data schema received by the server is variable. We want to be able to visualize the position of the sensors and switchboards over a map, to filter the data depending on the sensor type and to select a particular sensor and view its properties. The people in charge of placing the hardware over the city provide a CSV file with all the properties that can be used to load the data. The application must also allow the users to run specific analysis processes over the sensors (for the example, it is not important to describe these analysis). Only registered users can enter the application, and the administrator is the only one that can register users. Finally, there is a set of pages, editable by the admin, explaining the use of the application.

Using GISBuilder we can speed up the development of this example application, whose features are not totally basic, by following the next steps. To save space, we do not explain the process in full detail but only the most significant parts.

The first step when creating a new product is to choose then name, version, identifiers (maven-like group and artefact ids) and the languages of the product. After that, the features included in the product are selected. In this case, a non-exhaustive list of the features required would be: *userManagement, userRegistration, userRegistrationByAdmin, staticPages, staticPagesManagement, dataStreaming, dsToMongoDB, standardDataImport, csvImport, mapServer, mapViewer* and all the tools for the map viewer we want to include, like *mvItemSelector*. Each feature is referenced by its identifier, which is somehow describing of the proper feature, but there is no point on properly describing the features in this paper.

As data model, the analyst may define a structure with the next entities:

- Sensor type. Just an identifier for each sensor type, a description where the functionality for this specific sensor type is described and a textual field where the format can be explained.
- Sensor. Each one can have an identifier, which can correspond to a serial number of the actual hardware or something like that. The analyst also wants to know the type, the geographic point where it is placed and, maybe, the state of the sensor to know if it is working or there is a problem.

- Switchboard. An identifier, its geographic position, and the list of sensors that are physically linked to it.

If the application is receiving data from sensors, it is expected to receive lots of it. Besides, the schema of the data is variable and not known upfront. Due to these reasons, the best option is to store the received data in a distributed NoSQL database like MongoDB (supported by our component of data streaming).

While the properties of the sensors and the switchboards in the final product will be loaded from CSV files, the types of sensors is something managed manually, so the analyst includes some forms in the application to create, edit, view and remove types of sensor and a list to see them all.

The sensors must be visualized over a map, so the analyst has to include a map viewer from that entity. The map will cluster automatically the sensors when there are many in the same or near position and map zoom is far. When the zoom is close, each sensor will appear individually and the user can click over any of them to see its properties. The analyst should configure the map to allow filtering by types of sensors, so the user can view only the sensors of a particular type.

The rest of the application, including menus and static pages, should also be specified by the analyst.

In the final product, the data streaming component, in charge of receiving the data from the sensors using a REST service, must be configured by the admin (or any user, depending on the access rights of each role that also configures the admin). The component can be set to store the data from each type of sensor in a different data store.

The specific analysis processes mentioned in the description of this example must be implemented manually in the product. As we explained at the beginning of this paper, our framework may not be able to generate the full code of a specific application, which is not the objective as much as to speed up the *time to market* of new products. By generating everything but the analysis processes, our framework saves the time spent on implementing the most general functionalities and gives developers a quality product to extend with the specific functionalities.

8 CONCLUSIONS AND FUTURE WORK

Current SPL implementation techniques are not suitable for dynamic code generation from the specifications of the products. We have implemented a SPL derivation engine that handles classic variability and also what we have called parameterization of the components.

As future work, our main target is to handle efficiently the evolution of the platform and the already generated products using VSC, following an approach similar to Montalvillo and Díaz (2015). The first step to that would be the integration of GISBuilder with a VSC like git to generate the products directly in a code repository instead of getting an archive.

In parallel, the work on the components is not finished: some of them must evolve to provide new features, or more options for parameterization, specially the generators of lists, forms and map viewers, that are very basic yet, and some of them are not implemented and/or annotated yet.

References

Apel, S., Batory, D. Kästner, C. and Saake, G. (2013). Feature-Oriented Software Product Lines. Springer.

Brisaboa, N. R., Cortiñas, A., Luaces, M. R. and Pol'la, M. (2015). 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 '15), p. 320-331. Springer.

Clements, P. and Northrop, L. (2001). Software Product Lines: Practices and Patterns. Addison-Wesley.

Czarnecki, K. and Eisenecker, U. W. (2000). Generative Programming: methods, tools, and applications. Addison Wesley.

Czarnecki, K., Helsen, S. and Eisenecker, U. (2005). Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice (10), p. 7-29.

Feigenspan, J., Kästner, C., Apel, S., Liebig, J., Schulze, M., Dachselt, R., Papendieck, M., Leich, T. and Gunter, S. (2013). Do background colors improve program comprehension in the #ifdef hell? Empirical Software Engineering, p. 699-745. Springer.

Hunsen, C., Zhang, B., Siegmund, J., Kästner, C., Leßenich, O., Becker, M. and Apel, S. (2015). Preprocessor-based variability in open-source and industrial software systems: An empirical study. Empirical Software Engineering, p. 1-34. Springer.

Kästner, C., Apel, S. and Kuhlemann, M. (2008). Granularity in Software Product Lines. In Proceedings of the 30th International Conference on Software Engineering (ICSE ’08), p. 311-320. ACM, New York, NY, USA.

Kästner, C., Thum, T., Saake, G., Feigenspan, J., Leich, T., Wielgorz, F. and Apel, S. (2009). FeatureIDE: A tool framework for feature-oriented software development. In Proceedings of the 31th International Conference on Software Engineering (ICSE ’09), p. 611-614. ACM, New York, NY, USA.

Le, D., Walkingshaw, E. and Erwig, M. (2011). #ifdef confirmed harmful: Promoting understandable software variation. In Proceedings of the 2011 IEEE Symposium on Visual Languages and Human Centric Computing, VL/HCC 2011.

Medeiros, F., Kästner, C., Ribeiro, M., Nadi, S. and Gheyi, R. (2015). The Love / Hate Relationship with the C Preprocessor : An Interview Study. In Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP ’15), p. 999-1022. Leibniz International Proceedings in Informatics, Germany.

Meinicke, J., Thüm, T., Schröter, R., Benduhn F. and Saake, G. (2014). 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, volume 2 (SPLC ’14), p. 94-101. ACM, New York, NY, USA.

Montalvillo, L. and Díaz, O. (2015). Tuning GitHub for SPL development: branching models & repository operations for product engineers. In Proceedings of the 19th International Software Product Line Conference (SPLC ’15), p. 111-120. ACM, New York, NY, USA.

Pohl, K., Böckle, G. and Linden, F. (2005). Software Product Line Engineering. Foundations, Principles, and Techniques. Springer.

Spencer, H. and Collyer, G. (1992). # ifdef considered harmful, or portability experience with C News.