Journal of Communication and Computer 13 (2016) 125-134
doi:10.17265/1548-7709/2016.03.003

DSL Based on a Meta-Model for Optimization and Better Exploitation of Cloud Computing

Mehdi Mehdary, EL Habib Ben Lahmar and A. Tragha

MIT I (Information Treatment and Modeling Laboratory) Laboratory, Department of mathematics and computer sciences, Faculty of Science Ben M'sik, Hassan II University, Casablanca, Morocco

Abstract: We assume that the hybrid cloud computing model can perfectly meet the users' needs who want to fully exploit the benefits of competition from cloud providers to get the best quality service with the most optimized cost. The aim of this paper is to establish an implementation of meta-model suggested in our previous article for the purpose of creating a DSL specific for cloud computing. We will introduce the idea of decomposing the meta-model into modules independently distributed on different Clouds that form a hybrid cloud. These modules will be elementary to clarify the dependencies between units. In the first section of this article we are going to introduce a short definition of the concept of DSL (domain specific language) as well as some related concepts. In the second section we will introduce the benefits for the creation of a DSL to develop applications to host in a Cloud Computing environment. The third part introduces our approach, in the last section we will offer a multitude of tools and technology to facilitate the implementation of a DSL.

Key words: Cloud computing, hybrid cloud, meta-model, application architecture, engineering models.

1. Introduction

This article is to complement an earlier publication [1] about the deployment in a cloud environment, and establish a meta-model for the purpose of listing all the components, that we call functional units required for the deployment.

This article is composed of a first part introducing a short definition of the concept of DSM (domain specific language) and the other concepts related to it, the second part presents the advantages of the creation of our DSL for the development of applications which will be implemented in a cloud computing environment. In the third part we presents the approach addressed in this work, followed by a final section wherein it presents several tools and technologies to fluency the implementation of a DSL (domain specific language).

2. Definition of DSL

Programming languages can be classified into two categories: general languages or GPL (global purpose language) and specific languages domain or DSL (domain specific language). The first category is theoretically designated to solve problems regardless of their fields and complexity, e.g., Java, C++, C # are generalists languages and can troubleshoot different kinds of application problems, such as with C ++ we can create GUIs, manipulate data bases, create and control data flow and use sockets to transfer data and information.

DSLs are full-fledged programming languages, but in contrast to GPLs they are limited to solving problems for one area. DSL helps to provide keywords and ratings for the target domain, using its concepts and features. Thus it helps to provide an environment suitable for a field, to directly manipulate its concepts and their relationships. To define the concept of DSL Martin Fowler [2] wrote:

"The basic idea of a domain specific language (DSL) is a computer language that's targeted to a particular kind of problem, rather than a general purpose language that's aimed at any kind of software problem".

Corresponding author: M. Mehdary, Ph.D. student, research field: software engineering in cloud computing.

It is important to note that even in DSL we can identify two types [3] in terms of implementation, so we distinguish between internal DSLs (internal language) and external DSLs (external language).

Internal DSL: this type of DSL graft in GPL (general language) which aims to transform it into a DSL. It inherits the syntax and features of the host language. Thus it allows to exploit the features and access to the libraries written in host language. Also the pre-compilation spots or pre-interpretation like validation of syntax, semantics and lexical analysis are made in the host language environment.

External DSL: This type of DSL can be developed independently of other host languages. In this case you have to build a tool to analyze lexically source code and check syntax and semantics, in the following we can translate the source code into compressible code by the machine, or in a code of a GPL which will be then compiled or interpreted. External DSLs are more flexible in terms of representation of the concepts in the target domain, but they require a significant effort to implement them.

3. The Need for a DSL for Cloud Computing

The need of a DSL for cloud computing is to create cloud applications that do not depend exclusively on a single provider. The services delivered by cloud computing are distinguished by their ease of use, thanks to the online provisioning process. The applications run as SAAS (software as service), it means that users do not need to install and maintain these applications, users just have to subscribe to a provider and choose the application in question, then we can use a browser and the Internet (e.g., in the case of public cloud or hybrid cloud) to run these applications.

However, developers of cloud computing solutions are aware that the area is surrounded by significant challenges for both the development of SAAS —based solutions and for the deployment of applications to one or more suppliers. We will address some challenges related to the field of cloud computing by providing a solution based on domain specific languages to rise these challenges.

Problems related to the process of deploying on SOA (service oriented architecture) or third in architecture have been defined in Ref. [4] and are listed in the following:

- If the application is made at least two-thirds or two independent functional units, [1] for example one for the presentation layer and the other one for the persistence layer. It will not be possible to have a statement on the location of the two layers in the cloud after deployment the fact that Cloud Computing offers an open environment. This can then lead conflicts at the connection between these layers.
- The Cloud offers flexibility based on the creation of similar bodies of a layer or of a functional unit of the application. Then a simple description of these bodies would allow to the user to easily modify it.
- Billing in the cloud is based on the principle “I pay what I eat”, a user can host all parts of its application in a single virtual machine, or each in a virtual machine. The billing cost differs from one case to another. A way to choose one of the above approaches will add more flexibility to developers.
- Providers allow various deployment mechanisms based on the services they provide. For example, a provider offering infrastructure as service use low-level tools (e.g., SSH, FTP). Other providers that deliver the platform as a service supports protocols related to the technologies it implements, such as WARs files to Google App Engine.

The abovementioned challenges can be solved by various methods, for example you can create a library to simplify the controls, or create a tool to support most deployment of providers. But the best way to proceed is to create a specific language of deployment area, this solution will abstract all details of application deployment for the various providers. Which leads only one way will be used to deploy the application on different platforms and infrastructures.

Sledziewski et al. [5] made sure to fix the difficulties cited by building a DSL easy to learn and use, however it does not have the flexibility to modify the generated model because of its graphic form.

The challenges of developing applications for the cloud are due to diversity of programming platforms and technologies used by providers. We see today that each provider of the execution platform for cloud application uses its own platform and its own programming technology, such as Windows Azure uses. Net technology, PHP and causes the notion of roles (Web Role, Worker Role) while Google App Engine uses the Python, java, and GO language. This implies that when we proceed to the migration of an application or an application portion from a provider to another, we must resume again the implementations to adapt to the new service provider, this dependence can have a significant financial impact, so a provider can increase the cost of lodging an application without reason. Another disadvantage is that some applications are incomplete vis-a-vis the requirements. because developer can not provide all the needs of an application [6], this problem can be treated by a hybrid deployment where the application is distributed into independent parts that can be transported in different clouds as needed, then a connection between the various parts must be performed to ensure homogeneity of application, and to make services accessible, and where execution requires collaboration between several plot of the application deployed in different cloud environment. In this context, we consider the first disadvantage, the application development task can be difficult in view of its inflexibility, the multitude of technologies and application architectures supported by each provider.

Specify a common language for the cloud computing application development can be advantageous. The most important of these is to abstract the design details and application development for developers, especially those who do not have knowledge in cloud computing and application architecture, we can so have significant savings on development time, instead of studying and treated each provider environments we will have only coded with a DSL which is responsible for the translation, and even the selection of provider since the DSL itself is based on a meta-model that unifies multiple model of providers, and developers will not have to seek the intention behind the code. Also with a DSL we can just let high-level instructions or implement features that normally require details that are repeated from one application to another. This will automatically generate artifacts for the application source files or configuration files. This will reduce maintenance costs of the code compared with a GPL through the use of ratings that directly represent the concepts of cloud application architecture.

4. The Proposed Approach

It is obvious that before embarking on the implementation of a specific language area, we have to pass through a most important step for the success of a DSL, namely the extraction of all the domain concepts studied and the relationships between those concepts. Of course, this step can be done well by using of ontology to pick up the essential field of cloud computing architecture, provider, model and meta-model [1], then identify and represent their relationships. On the other hand we can use a modeling language such as UML to design our field by these charts. On this approach we propose to do this step by using meta-model cited in Ref. [1] which may be extended to perform this step. This meta-model is described in UML representing the concepts of classes and their relationships in associations between these classes, it allows to introduce an architecture for SAAS application design while merging architectures are relating to current providers in the market cloud computing like Google App Engine and Windows Azure. It allows to separate the design of an application platform and the provider’s infrastructure, which may be useful to

achieve the goal of the proposed DSL.

Then the DSL design can be in the form of two types of specific domain languages, either an internal DSL or an external DSL. In the case of an external DSL, we had to create new tools to edit and compile the instructions of DSL. The second choice is to use a framework for a host language, such as Java, to build the grammar of the DSL and implement it using the host language tools, then we can use these tools to edit and compile or interpret the source code of DSL.

Although the two choices are different in their implementation, the expected goal of DSL is still the same. Indeed, the DSL should automatically generate artifacts of an application for the different platforms desired by the developer. This leads on the one hand we will use one way to implement most of the functions of the application, and on the other hand artifacts generated will be compatible with several platforms. In fact it is for this goal that the DSL must be delivered by "Multi-Version" where each version will correspond to a given platform.

In the case of an internal DSL, developers can create instances models of the DSL. And they can change it later to fit the functionality of their applications. Then once the application is created it can be deployed in one or more providers platform according to the deployment model chosen [7], finally users can use browsers to run the application in the cloud.

The aim of this work is to show that using DSL for cloud computing would be a good way to control it and to exploit it. The following figure (Fig. 1) shows the approach referred by DSL.

5. Axes of the Proposed Approach

Using meta-model we find that each application contains functional units used for all case of hybrid deployment of cloud application:

Persistance Layer;

Business layer;

Presentation layer;

Management resources.

Each unit has properties:

virtual IP;

Communication port;

Protocol;

security element;

Definition of the unit.

Using the XML (markup language), it is possible to create a file for configuring the application (Fig. 2), in fact this file contains the functional units and their configurations. This configuration file contains configuration information used by DSL to adapt or customize its running.

The structure of the cloud application is always relative to the given configuration in the XML file, in each generation of the application, DSL reads and detects changes made to the configuration file (XML) to immediately apply new settings.

The supposed format for XML file (Fig. 3).

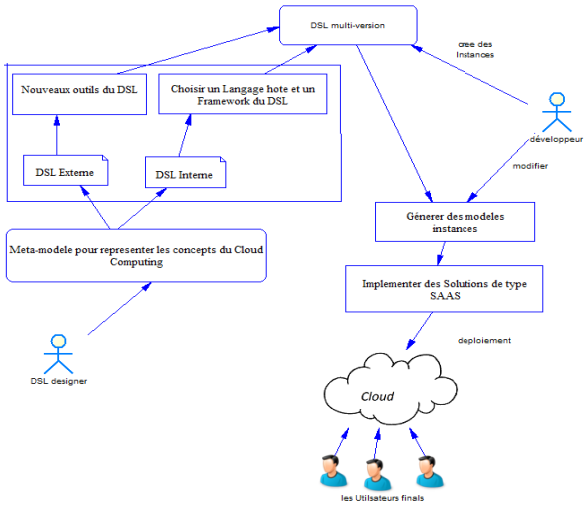


Fig. 1 The proposed approach to designing a DSL dedicated to cloud computing.

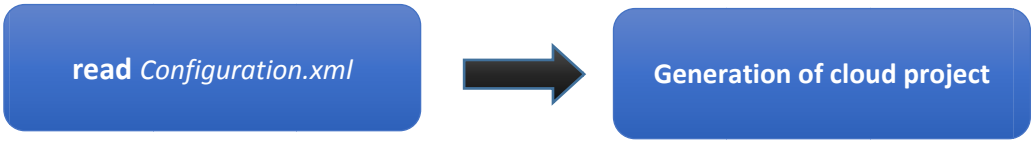


Fig. 2 Process of generating an eclipse project.

```
<?xmlversion="1.0"encoding="UTF-8"?>
<application>
<AppInfo>
<SolutionName>Gestion Stock</SolutionName>
<Developer></Developer>
<Version>1.0</Version>
</AppInfo>
<persistenceUnit>
<DBMS>Mysql</DBMS>
<hostIp>127.0.0.1</hostIp>
<protocole>tcp</protocole>
<port>3306</port>
<SecurityEl>SSL</SecurityEl>
</persistenceUnit>
<businessModelUnit>
<platform>J2EE</platform>
<framework></framework>
</businessModelUnit>
<presentationUnit>
<presentationLayertechnology>JSP</presentationLayertechnology>
</presentationUnit>
</application>
```

Fig. 3 Contenu dufichier config.xml.

6. Creation of DSL

Xtext setup from Update Site [8]

Open Eclipse IDE and choose **Help -> Install New Software...**

and then click on Add**...**and enter this Update Site according to

your version of Eclipse**:**

http://download.eclipse.org/releases/luna/ (In our case we have Luna version)

after installing Xtext and restarting Eclipse:

Then click on “Finish”, Eclipse will automatically create four projects (Fig. 5)

Now we will create the syntax of our DSL:

opens the syntax file(Fig. 6)

N.B: In this file we just created syntax, later we will set the working of each command, in view of this description is not contained in the file.

So far we only defines the syntax, we have to assign to each control it functioning Open file “Cloud DSL Generator.xtend”*(*Fig. 7*), this file is always* running if you want to generate LPG from the DSL.

The class Cloud DSL Generator implements the IGenerator interface that contains the do Generate procedure:

Question: What is the structure of the application ?

Answer:

Problem: How to create an Eclipse IDE project with this structure?

Answer: We will define a new type of project under Eclipse IDE (Cloud Project and cloud module), to do this we need to create a plug in for Eclipse benefiting from Eclipse RCP.

The RCP platform provides basic software components to build an application and the core executive to run it. Furthermore Eclipse RCP is used to customize Eclipse IDE.

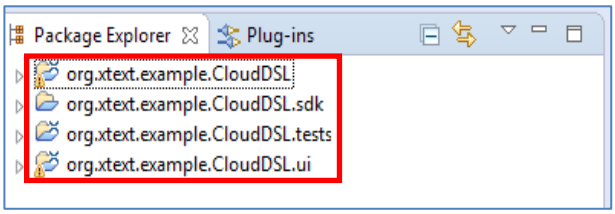
N.B: we have to install the SDK of Eclipse (help -> Install New Software ->http://download.eclipse.org/eclipse/updates/4.4)

After installing the Eclipse SDK' and restarting Eclipse, we will add a wizard, click twice on the plugin.xml, then click the Extensions tab above :

After creating of our wizard (Fig. 12) we test application:

A new Eclipse IDE instance will run (Fig. 10)

As you see our new project type “Cloud Application” is added to the list of projects that can be created, we click on“Next”.

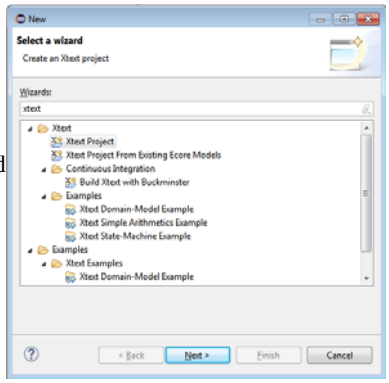


org.xtext.example.CloudDSL : *contains syntax validation, code generator*
org.xtext.example.CloudDSL.sdk : *contains the XtextSDK* org.xtext.example.CloudDSL,tests : *contains* the *Unit tests*
org.xtext.example.CloudDSL.ui : *Contains changes that will add on Eclipse (Eclipse RCP)*

Fig. 4 Tree of the xtext project.

First we will use the wizard to create the Xtext project:

File -> New -> Project... ->Xtext ->Xtext project



Fill with the name of Xtext project and the extension of your DSL files

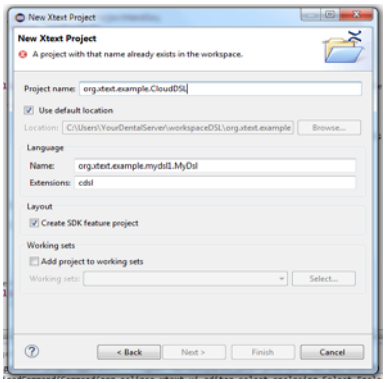


Fig. 5 Wizard for the creation of a xtext project.

```
grammar org.xtext.example.mydsl1.CloudDSL withorg.eclipse.xtext.common.Terminals
generatecloudDSL"http://www.xtext.org/example/mydsl1/CloudDSL"
Model:
(types+=Type)*(instances+=Instance)*(connections+=connection)*;
connection:
"Connection"name=ID "database="db=STRING "user="user=STRING "pass="pass=STRING;
Type:
Entity;
Entity:
'Entity'name=ID '{'      (properties+=Property)*'}';
Instance:
"new"ob_name=ID entity=[Entity]"("(inits+=Init)*")";
Property:
(type="Int«|»String") name=ID (PK="PK")? (value=STRING)?;
```

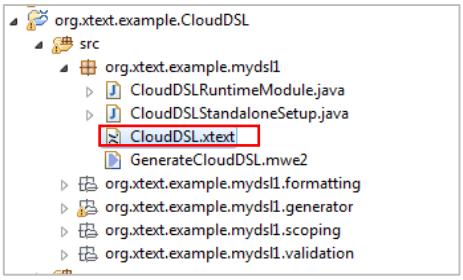


Fig. 6 Syntax of our DSL.

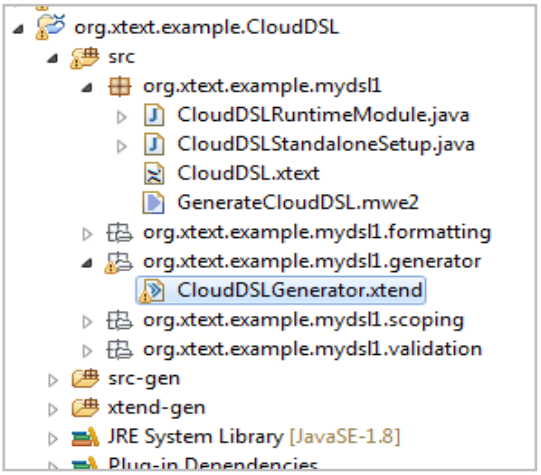


Fig. 7 Path of the xtend file relative to xtext file.

Projet Src Resources Src-gen	**Src :** contains the DSL extension files **Ressources:** Contains the resources used in the project (images, text file, xml file ...) **Src-gen:** contains the application generated by the DSL **Configuration.xml :** DSL configuration file

Fig. 8 Tree of the generated project.

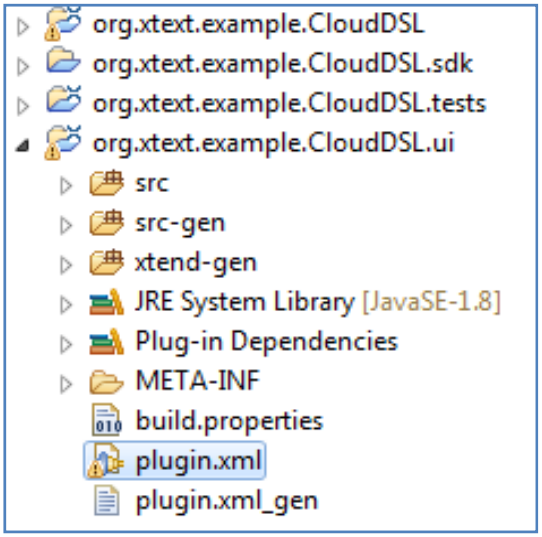


Fig. 9 Path of the plug in file relative to the generated project.

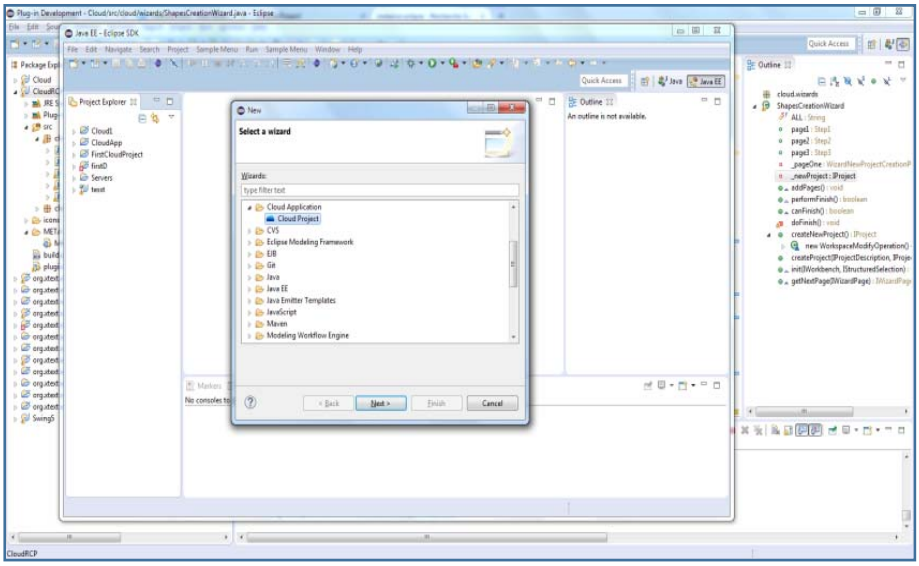


Fig. 10 New Eclipse IDE instance.

As shown in Fig. 12 our wizard has created the project and has implemented the configuration.xml file with the parameters entered in the wizard:

Once the project is created, the editor of Eclipse automatically opens the file “test.cdsl”

add this content to the file“test.cdsl” *:*

The command: Connection *—accepts 4 arguments, it serves to make a connection with the persistence unit within the parameters mentioned in configuration.xml*

The command: Entity *—used to create ORM, generate class (constructors, getters, setters ...)*

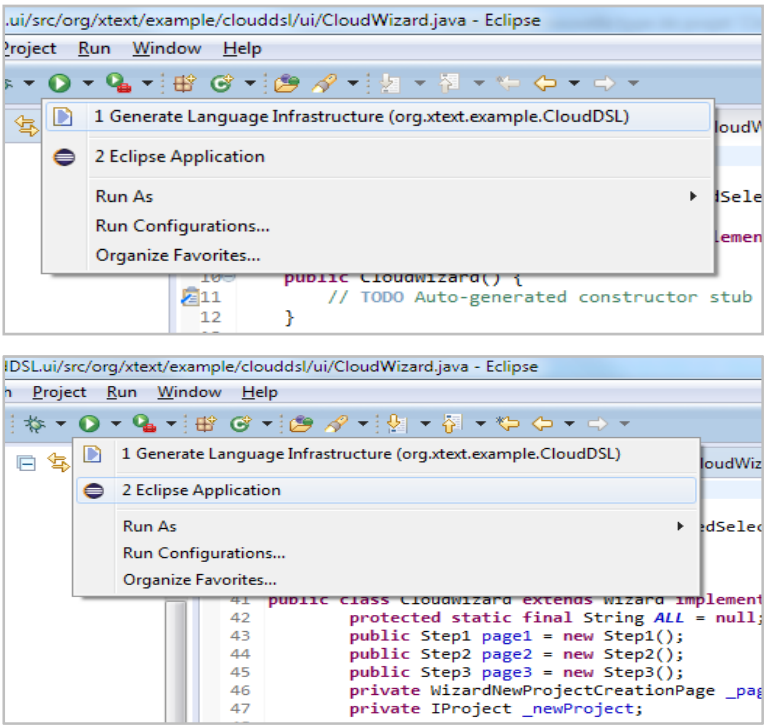


Fig. 11 **New content used for the generation of the target language and the generation of a new type of project.**

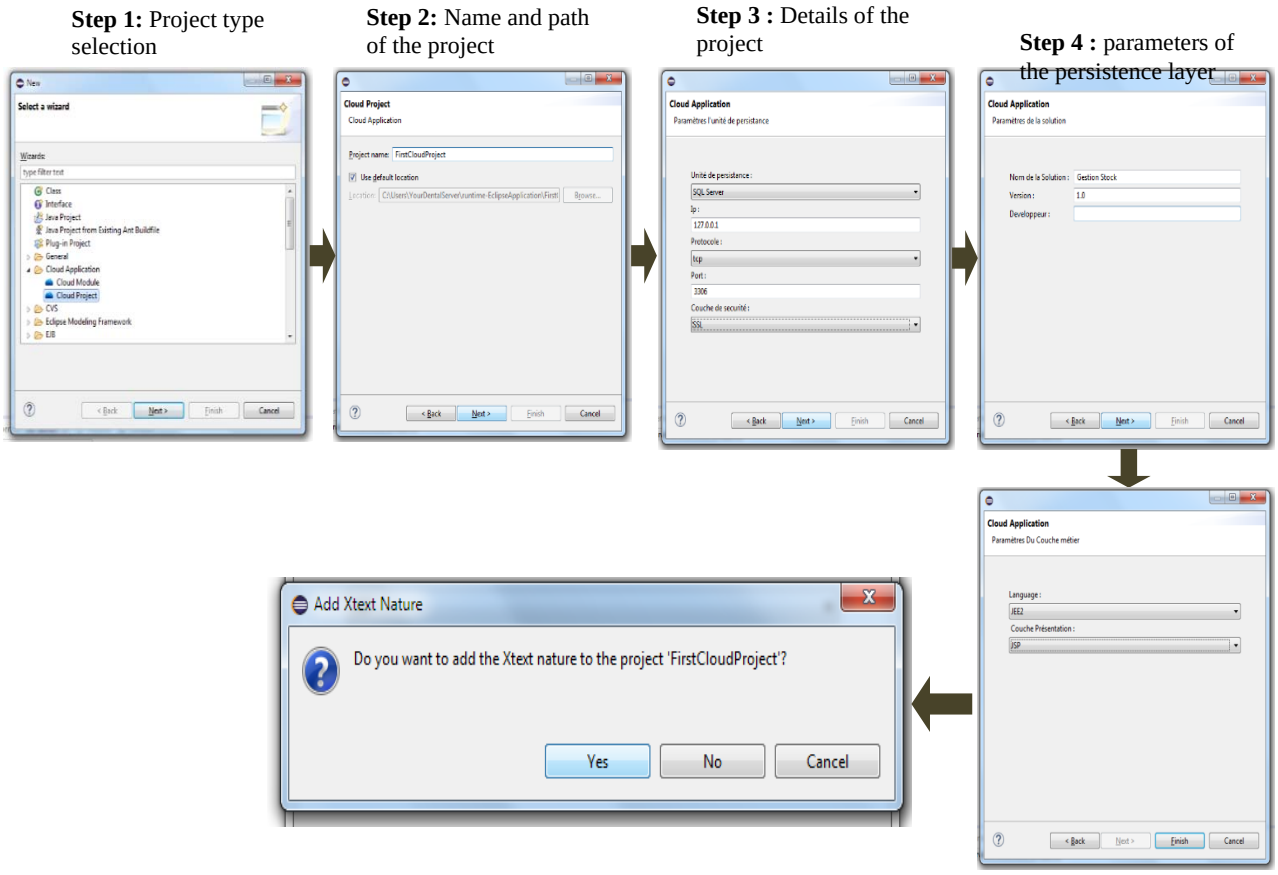


Fig. 12 **Proposed wizard.**

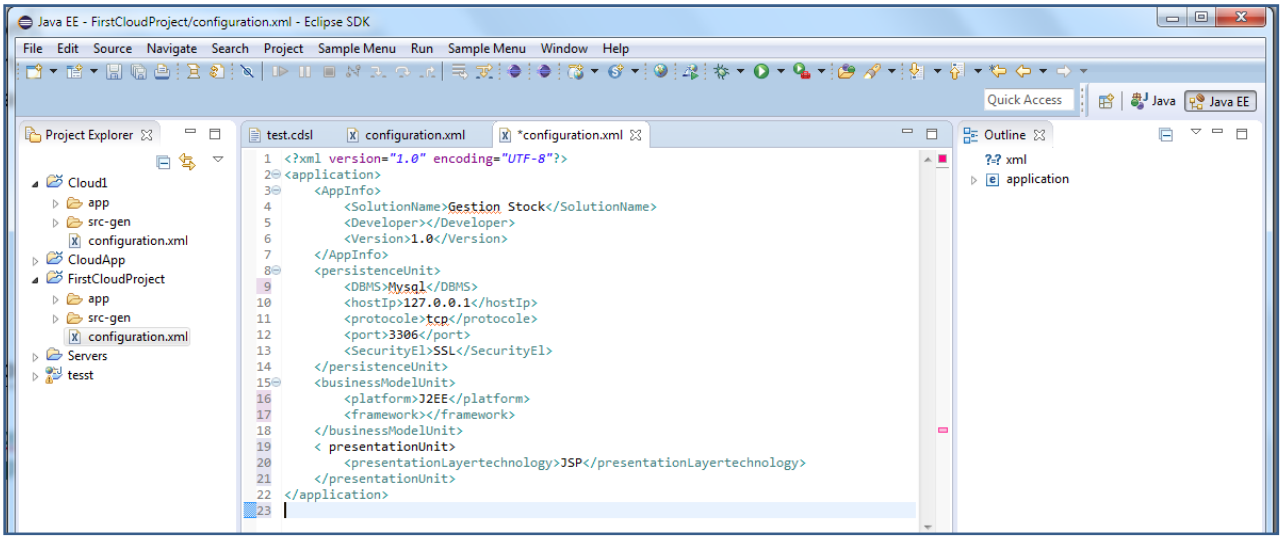


Fig. 13 Location of the .xml configuration file.

```
Connection Con1 database="test"user="root"pass="1"

EntityPersonne {
            Int Id PK
            StringFirstName
            StringLastName
      }
      new P1 Personne (
            Id=1
            FirstName="MEHDI"
          LastName="MEHDARY"
                        )
```

Fig. 14 Content to the file"test.cdsl".

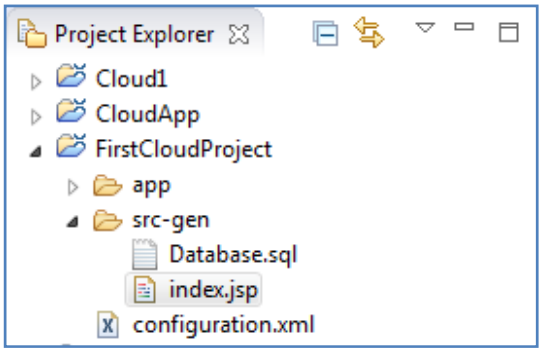


Fig. 15 Generated project by DSL.

The automatically generated files:

- *Database.sql*
- *Index.jsp*

Each entity has properties.

The command: new—is used to create instances (objects, then an Entity already created, it contains the boot arguments).

ORM: is an object-relational mapping.

If you save the *file* "test.cdsl" (Fig. 14) infrastructural language will generate the functional units within the parameters listed in the file configuration.xml (Fig. 13).

target language (in our case we choose between J2EE or PHP) (Fig. 11)

target database (in our case we choose between MySQL or SQL server)

protocol used

Technology of the presentation layer (**J2EE** => JSP (fig 15); **PHP** =>twig ou SMARTY)

After save of test.cdsl file, DSL generates the application.

7. Conclusions

In this paper we addressed the main problem encountered in cloud computing namely compatibility issues between suppliers and migration challenges in a cloud vendor implementation to another. We proposed a solution that involves establishing a language to unify the programming an application cloud without cared provider, we also proposed an approach that can be used to create the language, namely the creation of a schedule for the implementation of a functional unit and for changing these attributes.

DSL has been created to allowing developers the

ability to compile the functional units of the project to the platform he wants with the way he wants, and also to modify or change units whenever he wants. DSL acquires this flexibility through the Meta model proposed in our previous article [1].

References

[1] Design a Meta-Model For the Implementation of Hybrid Cloud, ijareeie, Volume 2, Issue 11, November 2013

[2] http://martinfowler.com/bliki/DomainSpecificLanguage.html.

[3] Paul, L. 2010. “Un Langage Specifique au Domaine Pour L'outil DE Correction DE Travaux DE Programmation Oto.” *Fevrier*.

[4] Eirik, B., Parastoo, M., and Sebastien, M. 2012. “Towards a Domain-Specific Language to Deploy Application in the Clouds.” .

[5] Krzysztof, S., Behzad, B., and Rachid, A. 2012. “A DSL-based Approach to Software Development and Deployment on Cloud.”

[6] Clement, Q., Nicolas, H., Romain, R., and Laurence, D. 2013. “ Towards Multi-Cloud Configurations Using Feature Models and Ontologies.”

[7] Francois, T. 2009. “Cloud computing: Strategie et Revolution de l'infrastructure Informatique, de la maniere de concevoir les applications et Leur consommation dans le nuage sous forme de services.”

[8] Eclipse Documentation. “Xtext 2.1 Documentation.”Octobre 31, 2011.