Journal of Universal Computer Science 23.10 (2017): 953-968

DOI: https://doi.org/10.3217/jucs-023-10-0953

PsiLight: a Lightweight Programming Language to Explore Multiple Program Execution and Data-binding in a Web-Client DSL Evaluation Engine

Enrique Chavarriaga
(Universidad Autónoma de Madrid, Madrid, Spain
enrique.chavarriaga@inv.uam.es)

Francisco Jurado
(Universidad Autónoma de Madrid, Madrid, Spain
francisco.jurado@uam.es)

Fernando Díez
(Universidad Autónoma de Madrid, Madrid, Spain
fernando.diez@uam.es)

Abstract: Domain-Specific Languages (DSLs) allow building software applications by simplifying the labour of both software engineers and domain experts thanks to the abstraction provided by a high-level code. Introducing a DSL in the software development process requires the use of technologies and frameworks in the design and implementation activities. If we are restricted to web-client applications, then XML-based languages and JavaScript frameworks and widgets are commonly used and combined in order to provide fast, robust and flexible solutions. Under this scenario, we have developed the *PsiEngine*, an interpreter able to evaluate programs coded in high-level XML-based DSLs (XML-DSLs) to provide solutions to domain specific problems within a web-client application. Thus, the goal of this article is to detail how we have built PsiLight, a lightweight programming language that runs on web-client. PsiLight supposes the exploratory case study we have conducted to check some features of *PsiEngine*, namely: multiple programs execution and data-binding capabilities in our interpreter.

Keywords: Domain-Specific Language, XML, XML Interpreter, JavaScript, Web Application, XML Programing Language, Data Access Object
Categories: D.1.5, D.2.3, D.2.6, D.3.3

1 Introduction

Under a standard perspective, XML allows storing auto-documented information as well as structured data interchange [Fawcett, 12]. As metalanguage, XML provides elements to enrich the web page presentation model (e.g., SVG, SMIL or MathML). From a server-side perspective, we can find XML-based solutions that integrate data models, graphic user interface models, access-control models, etc. (e.g. ASP.NET,

JSP or JSF).

On the other hand, Domain-Specific Languages (DSLs) provide a high-level abstraction approach in order to model specifications, structures and functionalities to solve domain-specific problems. The goal of a DSL is to make easier the design, the definition and the implementation of systems by allowing domain experts to better perform their tasks and building high quality and reliable systems in order to provide domain-specific solutions [Voelter, 13]. In this sense, Fowler [Fowler, 10] describes a DSL as a Computer Programming Language of limited expressiveness and useful only if it is focused on a small domain.

If we reduce the scope to build DSL approaches for web-client, we can find solutions like Jison, which generates JavaScript parsers for text-based DSLs [Carter, 09]. However, the Jison's main drawback while managing text-based DSLs is that developers must modify the generated code in order to obtain the final parser. Another alternative is to use the browser features to create plugins that perform this task, but it is just a browser-dependent solution.

Opposite to general text-based DSLs, the development and implementation of DSLs that follows XML-based grammars are easy to handle on a web-client. They are extensible and combinable, and also they can be processed as a DOM and manipulated with the JavaScript language. Thus, high-level XML-based languages that encapsulate a lot of functionality can be built, and we can faster create more robust and flexible solutions by joining them with other languages, widgets and frameworks. Furthermore, we can apply security policies and good programming practices in order to have safer and reliable DSL [Kern, 14][Yue, 13].

Under this context, we developed *PsiEngine* [Chavarriaga, 17], an evaluation engine that provides the necessary tools to implement and deploy XML-DSLs on the web-client. This engine includes the PsiXML Interpreter to evaluate programs written in DSLs, and also can bind information from either XML or JSON formats, and to execute *inline* JavaScript code.

The aim of this work is to detail the exploratory case study we have conducted to check how the *PsiEngine* can execute multiple programs and to perform data-binding with external resources. Thus, we have followed the qualitative case study methodology suggested in [Yin, 14] and adapted for software engineering [Baxter, 08] to test specific features in our engine. Hence, we will show how we have used the *PsiEngine* working methodology for the implementation of an XML-DSL on web-client. The developed DSL is *PsiLight*, a lightweight programming language that allows defining variables, functions, classes, instances and JavaScript blocks code. This case study combines XML, JavaScript, external resource binding and the inherent characteristics of the Psi Languages supported by *PsiEngine*. Also, the *PsiLightWeb* application will be presented as a lightweight development environment for the *PsiLight* language.

The rest of the paper is organized as follows. In section 2 an overview of the already existent tools for the development of DSL is introduced. Section 3 provides the general settings of the Psi Engine Evaluation and the Psi languages and components family. In section 4, we will formalize and implement the *PsiLight* language and also present the *PsiLightWeb* application. Finally, in the last section, some concluding remarks will be detailed.

2 Related works

In the literature, the term Domain-Specific Language (DSL) is not rigorously defined. In [Fowler, 10] describes it as *«a computer programming language of limited expressiveness focused on a particular domain»*. On her part, 0 centers on the concept of abstraction, defining it as *«a cognitive process of the human brain that enables us to focus on the core aspects of a subject, ignoring the unnecessary details»*.

Briefly, both authors agree that a DSL is a programming language that targets specific problem domains. In such way, their syntax and semantics contain the same level of abstraction determined by the problem domain and aims to implement information systems that provide solutions to that problem.

In [Kosar, 16] we can find a Systematic Mapping Study (SMS) on DSLs to identify research trends in the period 2006-2012. Their authors looked for possible open issues and an analysis on what they called demographics of the literature. In their SMS study, the authors observed that the DSL community appears to be more interested in the development of new techniques and methods that support the different phases of the development process (analysis, design and implementation) of DSLs, rather than researching new tools, and only a small portion of studies focus on validation and maintenance. In addition, the authors observed that most of the works do not indicate the tools they utilized for the implementation. In the field of DSLs, we can mention some impressive works like [Sánchez, 09] ModelSec, a generative architecture for managing security requirements, from the requirement elicitation to the implementation stage; and ASD [Vara, 12], a DSL toolkit for modeling the structural part of Abstract Service Descriptions.

Building a DSL solution involves the use of tools for the implementation of interpreters and compilers, by using scanners and parsers generators like Lex and Yacc [Brown, 92] or Flex and Bison [Levine, 09] to create them. However, nowadays, widely used Integrated Development Environments (IDE) such as Eclipse and Visual Studio .NET, provide tools and languages specifically designed to implement DSL. Thus, *we* can highlight several plugins for the Eclipse environment, such as *Spoofax* [Kats, 09], *Antlr* [Parr, 13], *Xtext* [Betinni, 13] and *Eclipse Modeling Project* [Gronback, 09]. In .NET framework we can mention *DSL Tools* 0 and *Boo* [Rahien, 10]. Additionally, it is also possible to develop DSL languages by taking advantage of programming language features like in the cases of *Groovy* [Dearle, 10] and *Clojure* [Kelker, 13], or alternative approaches like *pyparsing* using Python [McGuire, 07], the C# based Virtual Machine VM framework [Kourie, 08], or Aspects-oriented programming [Kniesel, 09].

To create DSLs approaches able to run in a web-client, we have solutions like *Jison* as well the *PsiXML* interpreter. Jison generates JavaScript parsers for text-based DSLs [Carter, 09]. Some examples are CoffeScript [Lee, 12], and js-sequence-diagrams [Carter, 10]. On the other hand, the *PsiXML* interpreter [Chavarriaga, 17] produces parsers directly in the web client for XML-based languages, by using DOM and the JavaScript language. *PsiXML* can binding XML and JSON information, and also executes *inline* code, i.e., programming statements written in the languages natively inherited.

3 The *PsiEngine* for building XML-DSLs

As briefly introduced, our starting point relies on the Programmable Solutions Interpreter Engine, noted as *PsiEngine* [Chavarriaga, 17]. *PsiEngine* implements, evaluates, interprets and executes XML-DSL code within the web-client. The *PsiEngine* uses HTML5, CSS3, JavaScript and DOM together with technologies, services and tools from Web 2.0 [Anderson, 12] and the specification of XML-DSL grammars in order to build web components, widgets, and dynamic web sites to give the solution to specific web application problem or a part of it.

Figure 1 summarizes the main concepts from [Chavarriaga, 17]. As we can see, the *PsiEngine* takes as input the source code written in an XML-DSL. These programs are what we call *PsiCode*. Then the *PsiXML Interpreter* (denoted as PsiXML) evaluates the *PsiCode* in order to dynamically build a specific kind of JavaScript object we call *PsiObject,* which can be used in the web application.

The PsiXML is a generic lightweight JavaScript framework (it works fine in every web browser device) that process and evaluates programs written in PsiLanguage [Chavarriaga, 17]. A PsiLanguage is an XML-DSL that has a specific document structure, i.e. XML tags, and their corresponding associated functionality, with the ability to bind to XML and JSON information sources natively as well as to execute *inline* JavaScript code.

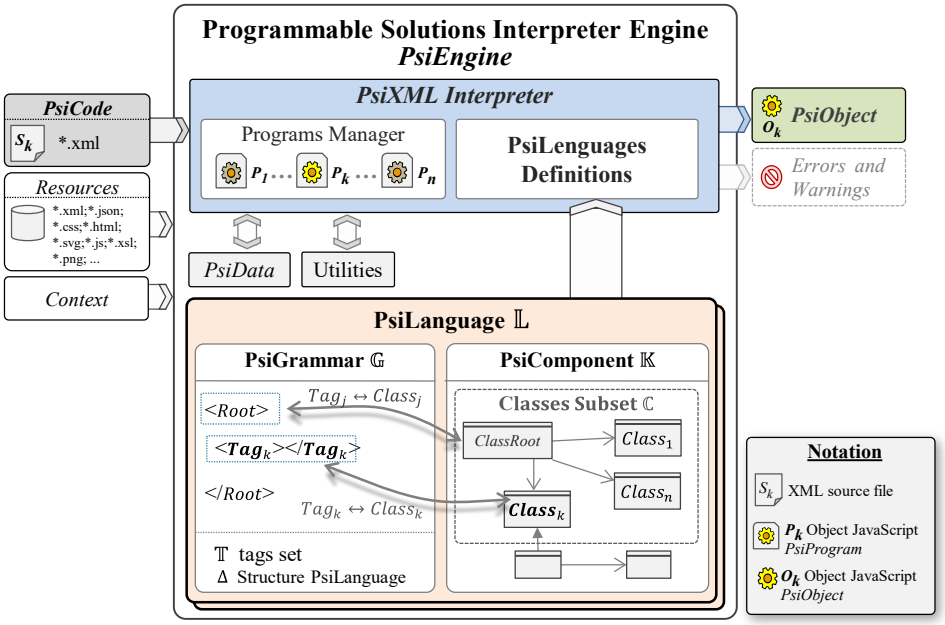


Figure 1: Programmable Solutions Interpreter Engine (PsiEngine) that allows executing PsiPrograms

PsiLanguages have similarities with other XML-based languages. To specify and

use the different XML-DSLs in the web-client, we follow the same approach of other XML-based languages, such as XSL, SVG, MathML, etc. In summary, all of them are intended to enrich the content of web pages by defining an XML grammar, where each XML element has its own semantic implemented by the corresponding associated functionality to achieve its objective once interpreted by the web-client. However, in spite of native code the interpreter into current Web-Browsers or utilizing plugins, *and PsiEngine* perform all the analysis directly using the language specifications that come from the server, and that can dynamically change. In this way, the *PsiEngine* efficiently manages new XML-DSLs, associates the functionality corresponding to their semantic, and provides a working environment that facilitates their evaluation.

The *PsiEngine* launches the PsiXML interpreter and registers the PsiLanguages to use. The execution of a P_k program in the *PsiEngine* consists of parsing the *PsiCode* (S_k source file), fetching the corresponding information sources (e.g. via AJAX) and evaluating the program in the PsiXML. To evaluate a program in the PsiXML consists of parsing the source code and evaluate the associated JavaScript code is for each element in the DOM. In the end, it is obtained a specific O_k JavaScript object (*PsiObject*) is obtained, which provides the solution to a Domain-Specific problem in a web application. The PsiXML can evaluate multiple programs written in different PsiLanguages allowing information, functionality and object exchange in a shared area called *PsiData*.

A **PsiGrammar** (see Figure 1), can be regarded as a tuple with the tag set for a PsiLanguage, the root tag, and the grammatical language structure. Formally, a PsiGrammar $\mathbb{G}$ for an XML-DSL can be defined by a tuple:

$$\mathbb{G} = \langle \mathbb{T} | Root | \Delta \rangle \tag{1}$$

where $\mathbb{T} = \{Root, Tag_1, Tag_2, \dots, Tag_m\}$ is the tag set for a PsiLanguage, been $Root$ the root tag, and Δ the grammatical language structure.

A **PsiLanguage** (see Figure 1), can be considered as a tuple with a PsiGrammar, a reusable component called PsiComponent and composed by a set of classes that implement the functionality associated with the tags, and the bindings between tags and classes. Formally, a Psi Language $\mathbb{L}$, can be noted as a tuple:

$$\Delta\mathbb{L} = \langle \mathbb{G} | \mathbb{K} | \mathbb{T} \leftrightarrow \mathbb{C} \rangle \tag{2}$$

where $\Delta_i \in \Delta$ is an object specified by:

$$\Delta_i = \{\text{TAG: } v_T, \text{ CHILDREN: } v_H, \text{ MULTIPLICITY: } v_M, \text{ STRICT: } v_S, \text{ VALIDATOR: } v_V\} \tag{3}$$

where $\mathbb{G}$ is the Psi Grammar defined in (1). The reusable PsiComponent $\mathbb{K}$ is the solution to the problem. The classes set $\mathbb{C} = \{Class_1, \dots, Class_n\}$ contained in $\mathbb{K}$ are the classes that implement the functionality associated with the language tags, i.e., $\mathbb{T} \leftrightarrow \mathbb{C}$ are bindings between Tag_k and class $Class_k$, for each $Tag_k \in \mathbb{T}$ and $Class_k \in \mathbb{C}$. For more details see [Chavarriaga, 17] (please consult http://www.github.com/echavarriaga/PsiEngine).

Additionally, the PsiLanguage Structure Diagram (PsiLSD), is the graphical representation of the grammatical structure. In that diagram, it is connected to the root tag and its corresponding class. For every tag from the PsiGrammar, we have the tag multiplicity, the associated class, the list of children tags with their corresponding associated classes; and the strict validation (i.e. other tags are not allowed or not is

required). On the other hand, the PsiGrammar Attributes Validator (PsiGVA) specifies the attributes validation for every tag. Both PsiLSD and PsiGVA simplify the specification of language for PsiXML. For more details see [Chavarriaga, 16] [Chavarriaga, 17].

To simplify the creation of PsiLanguages. We have the *PsiModel,* which allows defining PsiGrammars and implementing PsiComponents, as well as other JavaScript components (please consult http://www.github.com/echavarriaga/PsiModel).

To apply the *PsiModel* while building PsiLanguages, we have got a lightweight development environment called *PsiEnvironment*. This environment implements several features including code autocompleting for PsiModel, JavaScript, XML, HTML and CSS languages. It also has a visual component for online display of PsiLSD diagrams, UML Class diagrams as well as source code. Once the PsiLanguage is defined and implemented, the *PsiEnvironment* will automatically generate all the JavaScript code [Chavarriaga, 17] to evaluate and run the DSL.

The necessary steps to design and implement the PsiComponents can be summarized according to the following steps:

i. Create the PsiLSD and specify PsiGVA.
ii. Create the UML Class Diagram for the PsiComponent.
iii. Implement the PsiComponent within the *PsiEnvironment.*
iv. Perform functional tests for the PsiComponent.

Taking advantages of the dynamical nature and features of the *PsiEngine* and its PsiModel, in the next section, we will detail two PsiLanguages to build dynamic SVG diagrams, in order to provide an approach to building solutions for creating DSVL for web-client.

4 Case study: PsiLight Programing Language

The **PsiLight Language** is a DSL that follows an XML-based grammar for creating variables, functions, classes and JavaScript code blocks. FRAGMENT 1 shows an example of a piece of code written in PsiLight. In this fragment, we can see a program called *hello-program*. This program has a variable called *hello* with a string associated. Then a function called *Greet* is defined, with just one argument called *message*. The JavaScript code associated with this function is `alert(message)`. Finally, the program specifies a code block where the *Greet* function is called with a specific parameter.

FRAGMENT 1. Example of a piece of code implemented in PsiLight.

```
<?xml version="1.0" encoding="utf-8"?>
<Program name="hello-program">
  <Var name="hello">"Hello World!!"</Var>
  <Function                                                    name="Greet"
arguments="message">alert(message);</Function>
  <Block>Greet(hello);</Block>
</Program>
```

4.1 PsiLight Specification

Thus, starting from a tuple as previously defined in (1), we can specify the PsiLight Grammar, as follows:

$$\mathbb{G}_{PsiLight} = \langle \mathbb{T}_{PsiLight} | Program | \Delta_{PsiLight} \rangle$$

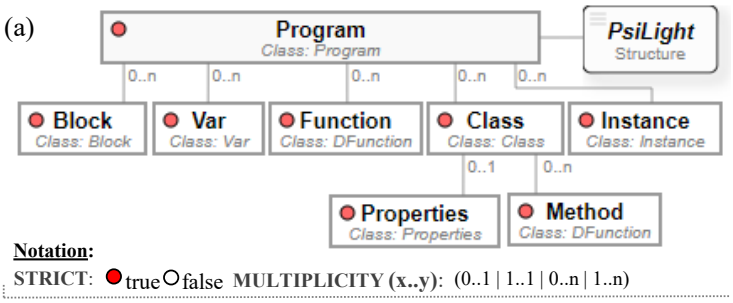
where tags set

$$\mathbb{T}_{PsiLight} = \begin{Bmatrix} \text{Program, Var, Function, Block,} \\ \text{Class, Properties, Method, Instance} \end{Bmatrix},$$

tag root is Program, and $\Delta_{PsiLight} = \{t_i : \Delta_i | t_i \in \mathbb{T}_{PsiLight}\}$.

Figure 2(a) shows the PsiLSD (graphical representation for $\Delta_{MiniPsi}$) of the PsiLight Language and Figure 2(b) shows the PsiGVA of the PsiLight Grammar. In that figure, we can summarize that the root tag **Program** is the beginning of the program. Then, we can define: variables (multiple **Var** tags), functions (multiple **Function** tags), classes (multiple **Class** tags), class instances (multiple **Instance** tags) and JavaScript execution blocks (multiple **Block** tags). The **Class** tag can contain a **Properties** tag to define the list of properties, and multiple **Method** tags to define their methods.

(a)



(b)

PsiLight Grammar

Tag	VALIDATOR
Program	*name:text
Var	*identifier:text
Function	*name:identifier|arguments:text
Class	*name:identifier|module:identifier |inherits:identifier
Method	*name:identifier|arguments:text
Instance	*psi-class:identifier |psi-name:identifier

Figure 2. (a) PsiLSD of the PsiLight Language. (b) PsiGVA of the PsiLight Language

4.2 PsiLight Component

In Figure 3, is shown the class diagram for PsiLight Component. The *Program* class defines the body of the program. We can define a module to encapsulate the definitions of variables, functions, and classes. Thus, *Var* class defines the behavior

associated with the *Var* tag. *DFunction* class defines the functionality related to the *Function* and *Method* tags. *Block* class has the responsibility to execute the JavaScript code contained within the *Block* tag. The *Class* class defines a JavaScript class and has properties (an instance of *Properties* class) and methods (multiple instances of *DFunction* class). The *Properties* class manages the properties, which are the attributes and their JavaScript values from the Properties tag. The *Instance* class represents an instance of the defined class. The reader is encouraged to look up the detailed implementation of the PsiLight Component in http://hilas.ii.uam.es/psilight or http://www.github.com/echavarriaga/PsiEngine.

In summary, if

$$\mathbb{C}_{PsiLight} = \begin{Bmatrix} Program, Var, DFunction, Block, \\ Class, Properties, Instance \end{Bmatrix},$$

such that

$$\mathbb{K}_{PsiLight} = \mathbb{C}_{MiniPsi} \cup \{\text{PsiLightParser}\}.$$

Moreover, the classes associated set is:

$$\mathbb{T}_{PsiLight} \leftrightarrow \mathbb{C}_{PsiLight} = \begin{Bmatrix} \text{Program} \leftrightarrow Program, \text{Var} \leftrightarrow Var, \\ \text{Function} \leftrightarrow DFunction, \text{Block} \leftrightarrow Block, \\ \text{Class} \leftrightarrow Class, \text{Properties} \leftrightarrow Properties, \\ \text{Method} \leftrightarrow DFunction, \text{Instance} \leftrightarrow Instance \end{Bmatrix}$$

By (2), the **PsiLight Language** it is defined as:

$$\mathbb{L}_{PsiLight} = \langle \mathbb{G}_{\text{PsiLight}} | \mathbb{K}_{PsiLight} | \mathbb{T}_{PsiLight} \leftrightarrow \mathbb{C}_{PsiLight} \rangle$$

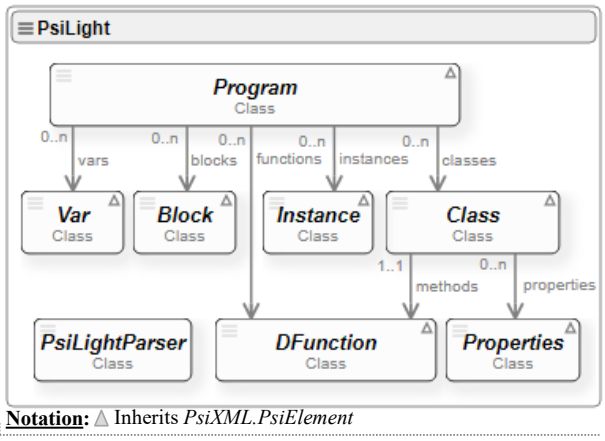


Figure 3. Class diagram for the PsiLight Component

4.3 Examples Programming

To better understand the concepts explained above, we contribute a couple of examples, namely PsiLI and PsiCA languages, featuring a Psi Language. Thus, suppose we have some information from people like in XML code shown in

FRAGMENT 2, and then we write the code shown in FRAGMENT 3, which defines a *Context* that includes information from these people.

FRAGMENT 2. File "people.xml"

```
<?xml version="1.0" encoding="utf-8"?>
<People>
  <Person id="p1" first="Luke" last="Skywalker" age="25"/>
  <Person id="p2" first="Obi-Wan" last="Kenobi" age="45"/>
</People>
```

FRAGMENT 3. Context for people from file "people.xml"

```
var context = {
  people: PsiXML.loadXMLSync("people.xml")
}
```

Then, FRAGMENT 4 shows a PsiLight Program based on PsiLI and PsiCA languages. The Var tag sets the *psi-context* attribute to call the PsiLI language. As a result, the *context* object takes the *people* field, seeking the person identification p1 (jQuery selector `Person[id=p1]`) and assigns it to the variable data *info*, adding it to the instance of the class associated with Var tag. It should be noticed that *psi-context* (*Context* information) and *psi-document* (Psi Data information) attributes are reserved, and both are available natively for all tags in any Psi Language.

The attribute *value* from the Var tag, in FRAGMENT 4, applies the PsiCA, and then *the info* is obtained. The value for the variable *person* would *"Luke Skywalker"*, and the value for the variable *age* would be *"20"*. From here, after evaluating the Block tag a message *"Luke Skywalker has 20 years"* will be shown.

FRAGMENT 4. Using PsiLI and PsiCA microlenguajes in PsiLight programs

```
<?xml version="1.0" encoding="utf-8"?>
<Program name="person-program">
  <Var name="person"
       psi-context="people:info=Person[id=p1]"
       value="$='{{info.first}} {{info.last}}'"/>
  <Var name="age"
       psi-context="persons:info=Person[id=p1]"
       value="$@info.age"/>
  <Block>alert(person+" has "+age+" years");</Block>
</Program>
```

A different application of PsiLI and PsiCA in PsiLight is to specify access to data from an XML file by using the Data Access Object (DAO) design pattern, so that the properties of the class can be obtained directly from XML data sources. Hence, FRAGMENT 5 is the instance class that implements access to XML data and creates an instance of the class properly.

Just like the class, the variables and functions parameters can be obtained directly from XML data sources. Examples are shown in "Associating XML data" on PsiLightWeb application.

FRAGMENT 5. Using PsiLI and PsiCA micro-lenguages for DAO design pattern

```
<?xml version="1.0" encoding="utf-8"?>
<Program name="person-class" module="MyModule">
  <Class name="Person" arguments="first, last"
         module="Users">
    <Properties first="first|s:@first" last="last|s:@last"
                age="null|i:@age" alias="null|s:@alias"/>
    <Method name="fullName" arguments="">
      return this.first+" "+this.last;
    </Method>
  </Class>
  <Instance psi-name="p2" psi-class="Users.Person"
            data-context="people:info=Person[id=p2]"
            psi-key="info"/>
  <Block>alert("Hello "+p2.fullName()+"!!");</Block>
</Program>
```

4.4 PsiLightWeb Development Environment

In Figure 4 we can see the web-based application for the PsiLight interpreter that we have developed. In http://hilas.ii.uam.es/psilight/examples it can be explored some examples. This page illustrated with examples (fragments including this paper) the basics elements of PsiLight language and association XML information. Available a set of classes and implemented in PsiLight language is implemented to create a simple graphic environment with graphic elements of HTML canvas.

In Figure 4, the “API Documentation” option shows the detailed implementation documentation PsiLight Language. The “Examples” option is the page in Figure 4 shown, and has the following tabs:

- **Files**. It contains a list of examples of PsiLight programs grouped in Basic Examples, Associating XML Data and Figures.
- **Program**. It is the PsiLight program's editor. The "New" option creates a new program. The "Execute" option evaluates the program on *PsiEngine*. And “Context” option, XML file edit in the context and use in different programs.
- **Result Text**. This text output for PsiLight programs. The `output.Print(message)` JavaScript function is used for this purpose.
- **Result Canvas**. This graphical output for PsiLight programs (HTML canvas).

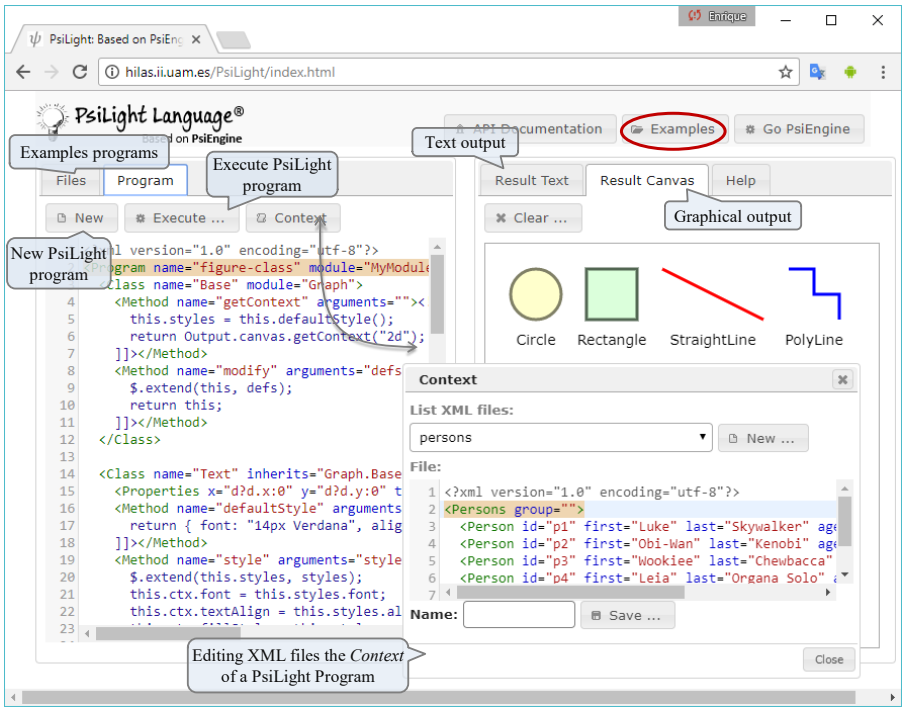


Figure 4. PsiLightWeb Application: examples PsiLight Language.

5 PsiLight Project: validation of the implementation

The *PsiLight Project* define and implement the PsiLight Component in *PsiModel* [Chavarriaga, 17]. Table 1 summarizes the *PsiEngine* project files. There are two files written in *PsiModel* Languages, with up to 303 lines of code in Psi Language that generate a total of 532 lines in JavaScript in the PsiLight.js file. This data means a conciseness ratio of 1.8. A *PsiLight Project* viewer is available in http://hilas.ii.uam.es/project?m=PsiLight.

	PsiModel			**JS generated code**		**Conciseness**
Components	MPsi	MIPsi	PSILOC	File	SLOC	PSILOC/SLOC
PsiLight	73 (1 file)	233 (1 files)	303 (2 files)	PsiLight.js	534	1.8

Note: **MPsi**: Psi languages specification; **MIPsi**: Psi component implementation. **PSILOC**: total Psi lines of code; **SLOC**: JavaScript generated lines.

Table 1. Grapher Project components summary.

Figure 5 shows a snapshot of the *PsiLight Project* metrics automatically generated by the *PsiEnvironment*. We would like to add that the average cyclomatic complexity CNN for the functions/methods of the project gives a value of "simple

functionalities" (CNN<10 according to [McCabe, 76]), the Maintainability Index MI is appropriated (MI>85 according to [Oman, 91]) and the commented lines of code CLOC are moderate (19.6%). Moreover, the time needed to implement or to understand a program rounds minor one day (14.3 hours according to Halstead Time [Halstead, 77]).

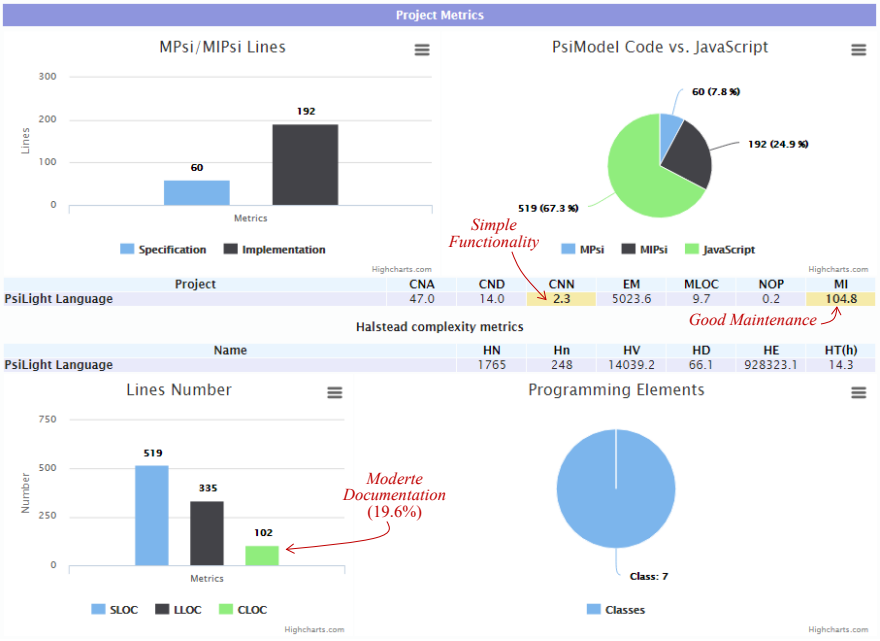


Figure 5. PsiLight metrics summary provided by the PsiEnvironment.

Finally, in Figure 6 shows the software metrics [McCabe, 76][Nguyen, 07] [Tahir, 12] at the *elements programming level* in the *PsiLight Project* (7 classes and 1 parser), the results of the metrics are summarized: a good MI, a simple CNN, and the elements are within the limits of the number of estimated error HB, ie. HB<2. These results can be seen directly in the *PsiLight Project*.

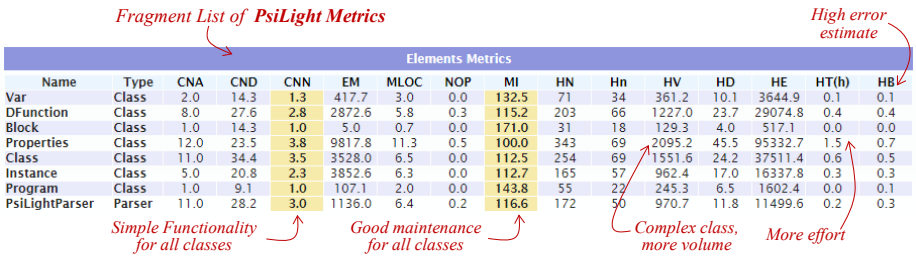


Figure 6. Software metrics snapshot for the components of the PsiLight Project *generated by* PsiEnvironment.

To conclude, Figure 7 displays the Maintainability Index (MI) for the PsiLight Component in addition to those components and frameworks used for these components and PsiLightWeb Application. As can be seen, all components and frameworks have good Maintainability (>85). The components developed from *PsiModel*, such as Graphs and Paint, are at the same level as renowned frameworks like CKEditor, Codemirror, and *PsiEngine*. The computing of the MI metric for different frameworks (CKEditor, Codemirror y jQuery) has been made with the JSComplexity tool (http://jscomplexity.org).

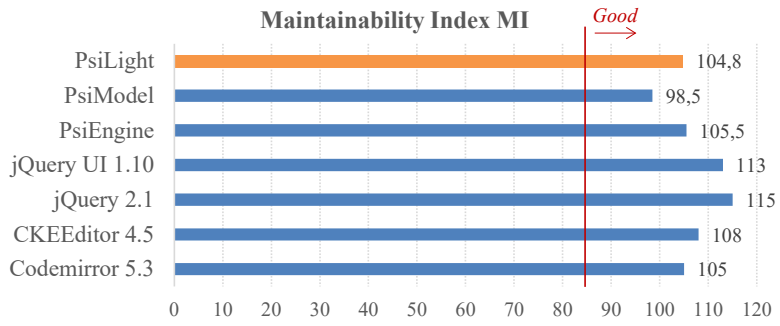


Figure 7. MI for PsiLight Component and frameworks used in PsiLightWeb Application.

6 Conclusions

Building DSLs to provide abstraction with high-level code involves the use of tools for the implementation of interpreters and compilers. However, as we have shown, there are few approaches to build web-client web DSL alternatives.

Along with this article, we have shown the Programmable Solutions Interpreter Engine (*PsiEngine*) that allows building web components, web widgets and/or dynamic web pages that provide solutions to specific problems in web applications. This engine includes the PsiXML Interpreter, which can evaluate programs coded in high-level XML-DSLs within a web-client application, and also can bind information from either XML or JSON formats, and to execute *inline* JavaScript code.

Thus, to check if the *PsiEngine* can execute multiple programs and to perform data-binding with external resources, we have detailed the exploratory case study we have conducted. To do so, we have shown how we have used the *PsiEngine* working methodology for the implementation of *PsiLight*, an XML-DSL for web-client. *PsiLight* is a lightweight programming language that allows defining variables, functions, classes, instances and JavaScript blocks code. This case study combines XML, JavaScript, external resource data-binding and the native characteristics of the Psi Languages supported by *PsiEngine*. Furthermore, along with the case study, the *PsiLightWeb* application has been presented as a lightweight development environment for the *PsiLight* language.

As a result, we have probed how our approach allows building XML-DSLs

solutions for web-client, since the developed engine can execute multiple programs written in the corresponding DSL, and furthermore it can perform data-binding with external resources. Also, the languages natively implemented together with its dynamical nature, makes possible to define and deploy new DSL solutions as required once the *PsiEngine* is running.

To facilitate the implementation of Psi Languages in the PsiEngine, we have used the *PsiModel* and the *PsiEnvironment*. The *PsiModel* is a programming model based on two Psi Languages: MPsi (specification language) and MIPsi (implementation language). For its part, the *PsiEnvironment* is a lightweight development environment that includes using full features such as code autocompleting, software metrics computation, diagrams displaying, etc.

Acknowledgments. This work has been partially supported by the DSVL-B2T research and development department from the B2T-Concept Company (http://www.b2tconcept.com/), and by the Ministry of Economy and Competitiveness (in Spanish Ministerio de Economía y Competitividad) through the project *Flexible Model-Driven Engineering for Mobile, Open, Dynamic Data Systems* REF: TIN2014-52129-R. The examples shown have been produced using B2T's technology and are reproduced with the permission of B2T.

References

[Anderson, 12] Anderson, P.: Web 2.0 and Beyond: Principles and Technologies, *Chapman and Hall/CRC*, London 2012.

[Baxter, 08] Baxter, P & Jack, S.; Qualitative Case Study Methodology: Study Design and Implementation for Novice Researchers, *The Qualitative Report*, vol. 13(4), 544-599, December 2014, http://www.nova.edu/ssss/QR/QR13-4/baxter.pdf

[Betinni, 13] Betinni, L.: Implementing Domain-Specific Languages with Xtext and Xtend, *Packt Publishing Ltd*, Birmingham, UK, Chapter 1, 2013.

[Brown, 92] Brown, D., Levine, J. & Mason, T., Lex & Yacc (2nd ed.), *O'Reilly Media, Inc*, 1992.

[Carter, 09] Carter, Z.: Jison, 2009, http://www.json.org/.

[Carter, 10] Carter, Z.: JS sequence diagrams, 2010, http://jison.org/.

[Chavarriaga, 17] Chavarriaga, E., Jurado, F., Díez, F.: An Approach to Build XML-based Domain Specific Languages Solutions for Client-Side Web Applications, *Computer Languages, Systems & Structures*, vol. 49, p. 133-151, 2017, DOI: https://doi.org/10.1016/j.cl.2017.04.002;.

[Chavarriaga, 16] Chavarriaga, E., Jurado, F., Díez, F.: PsiEngine, March, 2016, http://hilas.ii.uam.es/api.

[Cook, 10] Cook, S., Jones, G., Kent, S. & James, D.: Domain-Specific Development with Visual Studio DSL Tools, *Addison-Wesley Professional*, 1-23, 2010.

[Dearle, 10] Dearle, F.:. Groovy for Domain-Specific Languages, *Packt Publishing Ltd*, Birmingham, UK, Chapter 1, 2010.

[Fawcett, 12] Fawcett, J., Quin, L. & Ayers, D.: Beginning XML (5th. ed.). *Wrox Press*, Part III, V and VII, 2012.

[Fowler, 10] Fowler, M.: Domain Specific Languages. *Addison-Wesley Professional*, 21-27, 2010.

[Ghosh, 10] Ghosh, D.: DSLs in Action, *Manning Publications*, Greenwich, CT, USA, 9-15, 2010.

[Gronback, 09] Gronback, R.C.: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit, *Addison-Wesley Professional*, 2009.

[Halstead, 77] Halstead, M. Elements of Software Science, *The Computer Science Library,* New York, 1977.

[Kelker, 13] Kelker, R.: Clojure for Domain-specific Languages, *Packt Publishing Ltd*, Birmingham, UK, Chapter 1, 2013.

[Kern, 14] Kern, C.: Securing the tangled web, *Commun*, *ACM* vol. 57 (9), 38-47, September 2014.

[Kniesel, 09] Kniesel, G., Winter, V., Siy, H. & Zand, M.: Making aspect-orientation accessible through syntax-based language composition, *IET Software*. IEEE, vol. 3(1), 1-13, 2009.

[Kosar, 16] Kosar, T., Bohra, S., Mernik, M.: Domain-Specific Languages: A Systematic Mapping Study, *Information and Software Technology*, vol. 71, 77-91, 2016.

[Kourie, 08] Kourie, D.G., Fick, D. & Watson, B.W.: Virtual machine framework for constructing domain-specific languages, *IET Software*. IEEE, vol. 3(3), 219-237, 2008.

[Lee, 14] Lee, P.: CoffeeScript in Action, *Manning Publications Co.*, Greenwich, CT, USA, Chapter 1, 2014.

[Levine, 09] Levine, J.: Flex & Bison, *O'Reilly Media*, Inc, 2009.

[McCabe, 76] McCabe, T.: A Complexity Measure, *IEEE Transactions on Software Engineering*. SE-2:4, 308-320, 1976.

[McGuire, 07] McGuire, P.: Getting Started with Pyparsing, *O'Reilly Media Inc*, 2007.

[Nguyen, 07] Nguyen, V., Deeds-Rubin, S., Tan, T., Boehm, B.: A SLOC Counting Standard. *University of Southern California, Center for Systems and Software Engineering*, 2007, http://sunset.usc.edu/csse/TECHRPTS/2007/usc-csse-2007-737/usc-csse-2007-737.pdf.

[Oman, 91] Oman, P.W., Hagemeister, J., Ash, D.: A Definition and Taxonomy for Software Maintainability, *Technical Report, University of Idaho, Software Engineering Test Laboratory*, Moscow, 1991.

[Parr, 13] Parr, T.: The Definitive ANTLR 4 Reference (2nd ed.), *Pragmatic Bookshelf*, Raleigh, NC, USA, Part 1, 2013.

[Rahien, 10] Rahien, A.: DSLs in Boo: Domain-Specific Languages in .NET. *Manning Publications Co*, Greenwich, CT, USA, Chapter 3-4, 2010.

[Sánchez, 09] Sánchez, O., Molina, F., García-Molina, J., Toval, A.: ModelSec: A Generative Architecture for Model-Driven Security, *Journal of Universal Computer Science*, vol. 15 (15), 2957-2980, 2009.

[Tahir, 12] Tahir, A., MacDonell, S.G.: A systematic mapping study on dynamic metrics and software quality*,* Trento, s.n., 326-335, 2012.

[Vara, 12] Vara, J., Andrikopoulos, V., Papazoglou, M., Marcos, E.: Towards Model-Driven Engineering Support for Service Evolution, *Journal of Universal Computer Science*, vol. 18 (17), 2364-2382, 2012.

[Visser, 08] Visser, E.: WebDSL: A Case Study in Domain-Specific Language Engineering. *Generative and Transformational Techniques in Software Engineering II: Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, 291-373, 2008.

[Voelter, 13] Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L., Visser, E. & Wachsmusth, G.: DSL Engineering: Designing, Implementing and Using Domain-Specific Languages, *Dslbook.org*, 23-38, 2013.

[Yin, 14] Yin, R.K.: Case Study Research: Design and Methods (5st ed.), *Sage Publications, Inc*, London, United Kingdom, 2014.

[Yue, 13] Yue, C. & Wang, H. (2013). A measurement study of insecure javascript practices on the web. *ACM Transaction Web*, vol 7(2), Article 7, 1-39 pages, May 2013.