Defining DSL design principles for enhancing the requirements elicitation process

J. Guadalupe Ramos-Díaz[*] , Isela Navarro[*], Josep Silva[**] , Gustavo Arroyo[***]

ABSTRACT

Requirements elicitation is concerned with learning and understanding the needs of users w.r.t. a new software development. Frequently the methods employed for requirements elicitation are adapted from areas like social sciences that do not include executable (prototype based on) feedback. As a consequence, it is relatively common to discover that the first release does not fit the requirements defined at the beginning of the project. Using domain-specific languages (DSLs) as an auxiliary tool for requirements elicitation is a commonly well accepted idea. Unfortunately, there are few works in the literature devoted to the definition of design principles for DSLs to be experienced in the frameworks for DSL developing such as ANTLR, Ruby, and Curry. We propose design principles for the DSL development (regardless of paradigm) which are sufficient to model the domain in a requirements phase. Furthermore we enunciate a new profile for the requirements analyst and a set of elicitation steps. The use of DSLs not only give us an immediate feedback with the stakeholders; it also allows us to produce part of the real code.

RESUMEN

La Elicitación de Requisitos propicia el entendimiento de las necesidades de los usuarios con respecto a un desarrollo de software. Los métodos que se emplean provienen de las ciencias sociales por lo que se carece de una retroalimentación "ejecutable". Consecuentemente, la primera versión del software podría no cumplir con las expectativas. El uso de DSLs como herramientas para el descubrimiento de requisitos es una idea aceptada, desafortunadamente, muy pocos trabajos en la literatura se enfocan en la definición de principios de diseño de DSLs. En este trabajo planteamos principios de diseño de DSLs orientados a la elicitación de requisitos, enseguida, generamos casos de prueba en ANTLR, Ruby y Curry. También, enunciamos el perfil que debe tener el nuevo analista de software. Con ello, se incrementa la retroalimentación entre los involucrados en el desarrollo de software y se mejora el producto.

Recibido: 10 de Enero de 2012
Aceptado: 14 de Febrero del 2012

INTRODUCTION

Requirements elicitation is the process of uncovering, acquiring, and elaborating requirements for software systems [2].

Requirements elicitation is a complex process that involves many activities with a variety of techniques and approaches for performing them. Many of the techniques for requirements elicitation are borrowed from the social sciences because they emphasize the communication aspects with the stakeholders of the project.

Palabras clave:
DSL; ingeniería de software; requerimientos de elicitación.
Keywords:
DSL; software engineering; requirements elicitation.

Most of the elicitation techniques such as for instance, interviews, questionnaires, introspection, brainstorming, etc. do not produce an executable artifact that permits validation with the future users of the system. As a consequence, it is relatively common to discover that the first release of a system does not fit the requirements which were defined at the beginning of the project. We consider that an early prototype of the system produced during the elicitation phase can help in the definition of the system, although it implies additional costs of codification.

[*]DSC, Instituto Tecnológico de La Piedad, Av. Tecnológico 2000, CP 59300, La Piedad, Mich., México. Email: jgramos@pricemining.com
[**]DSIC, Universidad Politécnica de Valencia, Camino de Vera S/N, CP 46022, Valencia, Spain, Email: jsilva@dsic.upv.es
[***]Centro Interdisciplinario de Inv. y Docencia en Educ. Técnica: CIIDET, Av. Universidad 282, CP 76000, Santiago de Querétaro, México, Email: garroyo@ciidet.edu.mx

Fortunately the field of programming languages provides a variety of frameworks and tools useful for rapid system prototyping like for instance for domain-specific languages (DSLs) development.

A DSL is often a reduced language devoted to a specific domain and it is defined by creating abstractions that model the concepts of the domain and the interrelations between them.

We consider that the DSL development provides support to the elicitation process because it requires a better understanding of the problem. Unfortunately, there are few works in the literature devoted to the definition of design principles for DSLs that can be used in the development of DSLs for the requirement elicitation process.

In this work we sustain the use of DSLs during the elicitation process. Specifically we suggest the creation of DSLs adapted to domains of interest, for this we introduce simple cases of study where we illustrate how it is possible to clarify the concepts and their interrelations of a domain in order to be modeled (or elicited). DSL development is conducted by following a set of DSL design principles which are previously introduced. This is not a trivial task since there are many and clearly different paradigms able to produce DSLs.

At a later stage, we can implement DSL programs (sometimes they can generate code of the final application) that model and approximate the intended final system, and that can be discussed with stakeholders. These discussions provide the feedback that validates or rejects the desired requirements. Note that this proposal of elicitation causes a redefinition of the analyst profile since programming skills are mandatory. For this, we enunciate an analyst profile and the suggested steps for requirements elicitation based on DSLs.

The rest of the paper is organized as follows: In Section *Domain-Specific Languages* we introduce the DSL concept and discuss the newest paradigms for DSL development. In Section *Cases of Study in DSL Development* we describe principles of DSL design oriented to requirements elicitation and present three examples. In Section *DSLs and Requirements Elicitation* we propose a profile and a procedure for requirements elicitation based on DSLs. Finally, we conclude in Section *Conclusion.*

DOMAIN-SPECIFIC LANGUAGES

A domain-specific language is a programming language tailored for a particular application domain (e.g, Tex and Latex for document preparation, HTML for document markup, etc.). The main purpose of DSL design is the correct abstraction of the domain (Hudak [6]). This fact is very useful in software engineering.

Currently, the development of a new (domain) specific language does not represent an extremely hard work, like years ago. Today, there are many modern frameworks on different paradigms that are able to support DSL development. For instance, we experienced the multi-paradigm functional-logic language Curry, specifically we developed a DSL for routers specification [11] and a DSL for CNC (a manufacturing language) code generation [1] which is referenced in the following section.

In the field of formal languages there exist the well known tools: LEX (FLEX) and YACC (BISON) for C compilers generation. Recently a modern tool called ANTLR [10] for Java compilers generation became popular. We will show in a later section how ANTLR can be useful for DSL development. Furthermore, Ruby is a modern object-oriented language that provides advantages of higher level than Java, in next section we describe an example on Ruby. We will explore this kind of frameworks with an example of use.

CASES OF STUDY IN DSL DEVELOPMENT

In this section we describe three examples for DSL development employing different languages and frameworks. They were developed in order to define design principles which could be useful in any paradigm and also to analyze advantages and disadvantages of using DSLs in several programming frameworks.

The DSL development showed us that the process of DSL design is similar to the steps that an analyst follows in order to discover and clarify the domain which is subject of analysis for a new system development. In this section we reason about this similarity and its fundamentals, then we propose DSL development as an auxiliary and useful activity for requirements elicitation.

Design principles

Any application development should be envisioned following the next principles proposed by Bentley [3]:

- **Design.** We should use a language that minimizes the distance between the problem-solving strategies we have in mind and the program.

- **Analysis.** We should use a language that helps us to reason about its correctness.

- **Maintenance.** We should keep in mind that the next developer of a program might be someone who is totally unfamiliar with it.

These principles are naturally covered by DSLs. A DSL is composed of abstractions that model the domain concepts (vocabulary) and their interactions. DSL programs are simple and can reduce the hardness of the analysis, design and maintenance [6] of system prototypes.

The main decision is to determine what aspects of a domain should be modeled by the DSL. The requirements modeling with respect to the Objectory methodology [7] is described by three particular models: behavior, information and presentation. Following this basis we conduce the DSL design by focusing on the *static* (w.r.t. the model of information) and *dynamic* (w.r.t. the model of behavior) aspects of a domain.

Static aspect

A particular domain is formed by a set of *citizens* that perform the activities proper of the domain. For instance, in a bank an account or an account holder are common domain concepts. Some of them are not physical elements but abstractions intended to model a reality. Similarly, in object-oriented languages it is common to talk about classes; and in software engineering the concept of *domain model* is also familiar.

Dynamic aspect

It represents the set of actions performed by each element of the static aspect. It is usually known as the *business rules* of a software system. In the setting of object-oriented languages each class method implements a particular behavior of a class.

In architectural design of software, normally both the static and dynamic aspects are divided into more than one view. For instance the common views business rules and web services could be considered as the dynamic aspect in the analysis of a system. Similarly the domain model view and the persistence view represent the static aspect of a system. Nevertheless, these design considerations of a system are in a different level of abstraction than the elicitation phase, and thus, in the following we will only focus on the static and dynamic aspects of a domain.

Regularly, it is during the requirements analysis when the system analyst translates the stakeholder expressions in technical documents or diagrams that materialize the requirements specification. Such specifications are class diagrams, entity-relation diagrams, etc. which properly model domain abstractions and their interactions. Thus, developing a DSL implies (in an early phase) the discovering of requirements.

In the following we introduce some cases of study focusing in static and dynamic aspects of the corresponding domain.

A DSL with Curry

Curry is a universal programming language aiming to amalgamate the most important declarative programming paradigms, namely functional programming and logic programming.

Example 1 Problem description: *The Computerized Numeric Control (CNC) is an industrial language for the manufacturing of products. CNC programs are series of codes that consist of assembler-like instructions. Hence, they are low-level programs that require specialized developers in order to gain productivity. A system for CNC code generation is usually mandatory. We required a DSL that resemble common CNC operations like drilling holes, performing linear and circular movements, defining units of measure, doing canned cycles, etc. These operations were going to be interpreted by a CNC machine with mechanical tools able to move in the X, Y, and Z axis with respect to a static surface.*

The first step of the DSL development is to determine the static and dynamic aspect of the domain.

1. **Static.** The citizens of this domain are tools, geometric figures, measures, reference axis, etc.

2. **Dynamic.** A main requirement of this DSL is to produce CNC code. In this way, we should produce real code, i.e., a DSL program is an executable requirement. Once a tool is defined in CNC all kind of movements are actions that represent the behavior of a physical tool; thus, drawing arcs, drawing lines, performing canned cycles, etc. constitute the dynamic part of the system.

In Curry a program is composed of a set of functions. In this way all static abstractions to be modeled

are represented with functions or with data structures, while the dynamic aspect of the system is modeled by employing functions and sometimes chaining function invocations.

For instance, a function to make a circular cylinder is the following: `DrawCylinder (i, j, k, Height, Radius,Feedrate, dx)` whose arguments `i,j,k` indicates the center of the circle; `Height` represents the depth of the drilling; `Feedrate` represents the cutting speed and `dx` represents the width of the cutting tool. Each argument represents a static element of the domain, while the function is the dynamic face when it is invoked. A DSL program is as follows:

```
DrawCylinder (2.0,2.0,0.0,0.25,1.0,0.5)
```

which makes a circular canned (see figure 1) by defining a center of the circular sector in (2,2,0) with a 0.25 units depth, with 1.0 units radius and a tool 0.5 units wide. This DSL function produce de following CNC code:

```
[(G "00"),(X 2.0),(Y 2.0),(Z 0.0)]
[(G "01"),(Z (-0.25)),(F 40.0)]
[(G "01"),(X 2.5),(F 120.0)]
[(G "03"),(X 2.5),(I 2.0),(J 2.0),(K 0.0),
                                  (F 120.0)]
[(G "01"),(X 3.0),(F 120.0)]
[(G "03"),(X 3.0),(I 2.0),(J 2.0),(K 0.0),
                                  (F 120.0)]
[(G "01"),(X 3.5),(F 120.0)]
[(G "00"),(X 2.0),(Y 2.0),(Z 0.0)]
```

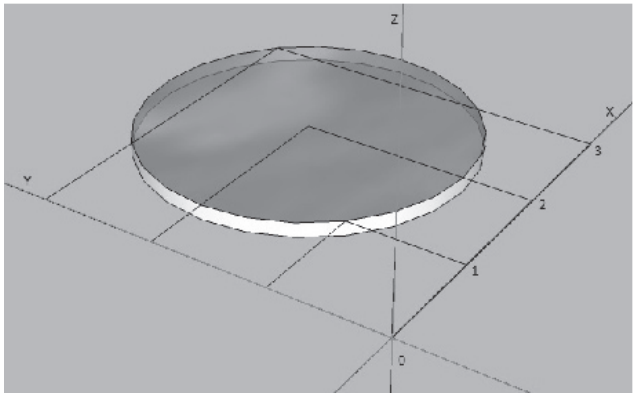

Figure 1 . Circular canned.

The description of the DSL can be found in [1]. In this case of study the goal was the modeling of a domain and also to produce real code. Here, it is evident that the DSL development conduced to a better understanding of the domain and, besides, it was possible to yield the code of the final application.

An ANTLR DSL

ANTLR [10] is the most modern language tool that provides a framework for constructing compilers from grammatical descriptions. It is possible to define actions for a variety of target languages.

Example 2 Problem description*: A software developer requires a tool for expressing the user requirements of new projects to be developed. In a typical meeting with a customer, she clarifies which are the abstractions to be represented and the actions they should perform. Regularly, abstractions are classified as: logic, for actions or services; data for concepts to be recorded and visual for indicating the future system interfaces. It should be desirable a tool for registering the meaningful software requirements.*

In order to solve the problem we focus in the following aspects of the problem:

1. **Static.** In this case, we identify the concept of *abstraction* and its different types like the static citizens of the domain, i.e., in a meeting is mandatory to register the following abstractions: logic elements, for actions representing; data elements, for database tables and visual elements for interfaces specifying.

2. **Dynamic.** Each abstraction and its attributes should have a method to interact with each other. In an object-oriented environment there is a sequence of method calls which are contained inside another method. Hence, once abstractions are defined it is required a mechanism for combining them and for calling them.

In the following we introduce a grammar which is composed by a set of syntactical rules in lowercase and a set of lexical rules.

```
prog          :  abstraction+ ;
abstraction   :  dataElement   | logicElement
                               | visualElement ;
dataElement   :  'data' 'element' IDABSTRACTION
                   '{' dataAttrib*   '}';
dataAttrib    :  'attrib' ID;
```

```
logicElement  :  'logic' 'element' IDABSTRACTION
                   '{' logicService* '}';
logicService  :  'query' ID
              | 'action' ID ;
visualElement :  'visual' 'element' IDABSTRACTION
                   '{' visualItem*   '}';
visualItem    :  vDataItemOp    | vLogicItemOp
              | vVisualItemOp;
vDataItemOp   : dataOp dataGranule
                          (ID|IDABSTRACTION);
vLogicItemOp  : 'call' logicOp  ID;
vVisualItemOp : 'call' 'visual' IDABSTRACTION;
dataOp        : 'add'   | 'update' | 'del'  ;
dataGranule   : 'data'  | 'attrib' ;
logicOp       : 'action'| 'query'  ;
//lexical rules
IDABSTRACTION : ('A'..'Z')
                ('A'..'Z'|'a'..'z'|'0'..'9')*;
ID            : ('a'..'z')
                ('A'..'Z'|'a'..'z'|'0'..'9')*;
```

Both lexical and syntactical rules define the set of concepts of the domain. For instance, a DSL program is composed of abstractions. An abstraction is composed of data, logic or visual elements; and they have attributes, services or items respectively. Relations between concepts are defined by sequences of compositions which are explored when an expression is decomposed (by a compiler) following grammar rules.

A DSL program is as follows:

```
data element Book
   { attrib bTitle
     attrib bAuthor
     attrib bIsbn
   }
logic element BookUsing
   { query mostUsed
     query lessUsed
     action avgUsing
   }
 visual element Interface
   {  add data Book
      add attrib bAuthor
      del attrib bIsbn
      call action avgUsing
      call query lessUsed
      call visual Window
   }
```

We observe in the DSL program that all type of abstraction can be classified as data, logic or visual. The visual element Interface encloses the behavior of the abstractions since it models the interactions between them inside of an interface. The figure does not include instructions for real code generation, however this is not a hard task because they could be added to the syntactical rules [10].

A DSL with Ruby

In this section we analyze an instance of DSL development in Ruby. Ruby is an object-oriented programming language very useful for DSL development [12].

Example 3 Problem description*: let us consider a domain relative to house building. Frequently a house is constructed by taking into account diverse materials; for instance, walls can be constructed with many type of bricks: handcrafted or industrial and the construction activities imply labor hours. A system that allows us to define types of materials and to compose segments (from windows, doors, etc.) in order to build a house and to calculate its cost is desirable.*

Again, we clarify the static and dynamic aspect of the domain.

1. **Static.** In the house building domain we can discover the following citizens or concepts: brick, wall, labor hour, segment, window, door, etc.
2. **Dynamic.** In order to construct segments we should be able to add materials, compose segments, add segments, etc. Each time an element is added it must calculate its cost, and this, should be automatically considered for the total cost of the construction.

In summary, the goal is to develop a system for calculating the implicit costs derived from the construction of a house.

In the following we describe the DSL whose design was inspired by Martin Fowler in [12].

Typically a house is built from different *segments* such as floor, walls, windows, doors, etc., and all segments contribute to the total cost computation.

```
class Construction
 def initialize
   @segments = {}
 end
 def addSegment arg
   @segments[arg.id] = arg
 end
```

```
  def [] arg
    return @segments[arg]
  end
  def allsegs
    @segments.values
  end
  def ccost
      tcost=0
      allsegs.each do |s|
           tcost += s.scost
      end
      return tcost
  end
  def to_s
    "\n This Construction is composed of \n" +
       allsegs.join("\n") + "\n
    And the total cost is: %.2f "%[ccost]
  end
end
```

The concept `Construction` is the main abstraction. It resembles a construction project, which includes all segments that compose an entire house.

Creating a system for costing requires to determine particular concepts to be taken into account. A brick is modeled as it is shown below:

```
class Brick
  attr_accessor :type, :number
  def initialize
    @@costh = 2.5
    @@costi = 2.0
  end
  def cost
    if type==:industrial
      return @@costi * number
    elsif type==:handcrafted
      return @@costh * number
    end
  end
  def to_s
    "Brick requirements <<type: %s, quantity: %d,
                          cost %.2f>>" %
    [@type,@number,cost]
  end
end
```

A brick is modeled in its corresponding class, let us observe the class variables (`@@costh`, `@@costi`) that define prices for all instances of `Brick`.

Example 4 *A DSL program and its execution is presented in the figure 2. In the code we can observe a construction composed of a wall, and the wall has some requirements, e.g., industrial and handcrafted bricks and units of labor. Let us observe the domain language instead of computational language.*

When the domain user begins a program (`begBuilding`), a global variable (as in [12]) is established to register the construction and all its segments. When a segment is declared, again a global variable defines the current segment. Variables predefined with ":" are denominated symbols in Ruby, and they are associated to a new object.

DSL development shows how the understanding of the project becomes better since it implies detection and programming of domain concepts just as an analyst should face the elicitation phase.

DSLS AND REQUIREMENTS ELICITATION

In this section we present a set of skills necessary for the analyst for requirements elicitation based on DSLs. We also describe a number of steps used to achieve a successful elicitation.

Elicitating steps

The requirements process is often described as a series of activities such as elicitation, modeling, triage (determining which subset of the requirements ascertained by elicitation are appropriate to be addressed in specific releases of a system),

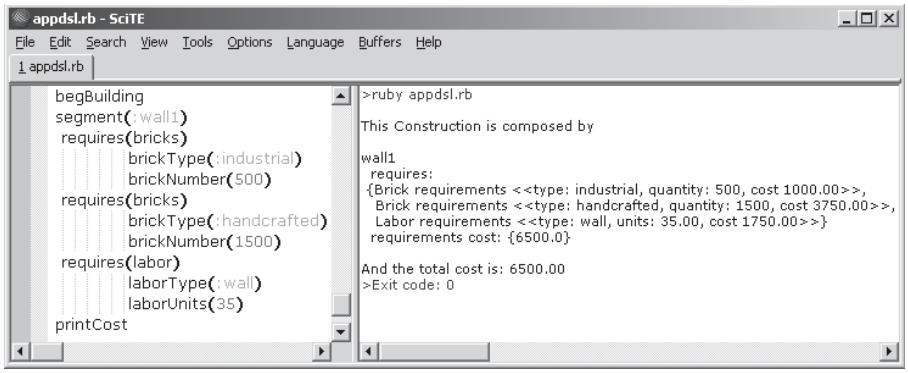

Figure 2 . Execution of the Ruby DSL program of Example 3.

specification, and verification; although there is little uniformity in the industry concerning names given to these activities, see [5] for a survey.

Next, based on our experience in DSL development, we propose some steps that lead to a successful requirements elicitation.

1. **Organize** meetings to motivate interaction between stakeholders, users and analysts and produce abstractions from the problem domain.
2. **Classify** the static part of the domain by uncovering abstractions either physical or conceptual.
3. **Discover** (1) the dynamic behavior of each abstraction, (2) how domain citizens interact, and (3) how each domain citizen affect the others; i.e., how abstractions coexist.
4. **Discuss** with stakeholders about abstractions and their behavior. Take into account the feedback to improve the static and dynamic domain identification.
5. **Build** a DSL.
6. **Write** DSL programs for modeling the required system.
7. **Present** the DSL programs to stakeholder for tuning and validating the system model.

Once the steps are concluded the understanding of the domain is fully achieved and depending on the paradigm used in the DSL development it is possible to yield code fragments of the real system.

Analyzing elicitation by means of DSLs

A mean for arguing the validity of our elicitation steps could be based on a simple comparison between the number of abstractions uncovered by typical requirements elicitation and those based on our procedure. However, it seems clear, that developing an executable artifact that models a set of domain abstractions and their behavior produces a best understanding of the domain, just like elicitation requirements pursues.

When feedback with stakeholders is performed, perhaps, someone could argue that they are unable to discuss about programs with an analyst, however DSL programs are simpler and understandable and frequently they can be read in natural language.

CONCLUSIONS

This work presented three examples of DSL development which were mainly oriented to present a methodology for DSL design and to demonstrate how it can be useful despite of significative differences between the paradigms (declarative, formal or object-oriented frameworks).

The idea of using software prototypes for software requirements is not new. [8] proposes an elicitation method based on a tool that uses XML to record the answers exposed by a domain user. Several authors propose specification languages for particular domains such as, e.g., [9, 4]. However they do not consider the use of DSLs as a main procedure for the definition of a general method of requirements elicitation. We have identified a set of necessary abilities for the analyst and a procedure composed of a set of steps that are necessary for a successful elicitation process. The profile of the analyst ensures that a DSL development is accomplished within the time limits given for the requirements phase.

ACKNOWLEDGMENTS

This work has been partially supported by the Mexican "Dirección General de Educación Superior Tecnológica" under grant 2369.09-P and by the Spanish "Ministerio de Ciencia e Innovación" under grant TIN2008-06622-C03-02 and by the "Generalitat Valenciana" under grant PROMETEO/2011/052.

REFERENCES

[1] Arroyo, G., Ochoa, C., Silva, J., Vidal, G. (2004). Towards CNC Programming Using Haskell. *Proc.of the IX Ibero-American Conference on Artificial Intelligence (IBERAMIA 2004),* pp.386-395.

[2] Aurum, A., Wohlin, C. (2005). *Engineering and Managing Software Requirements.* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[3] Bentley, Jon. (1986). *Programming Pearls.* ACM, New York, NY, USA.

[4] Bhattacharjee, A. K., Shyamasundar, R. K. (2008). A Specification Language for Web Service Choreography. *APSCC,* pp,1089-1096.

[5] Hickey, A. M., Davis, A. M. (2004). A Unified Model of Requirements Elicitation. *Journal of Management Information Systems,* 20:65.

[6] Hudak, P. (1998). Modular Domain Specific Languages and Tools. *Proc. of the 5th Int'l Conf. on Software Reuse (ICSR '98),* pp.134. IEEE Computer Society.

[7] Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G. (1992). *Object-Oriented Software Engineering --- A Use Case Driven.* Addison-Wesley.

[8] Kassel, N.W., Malloy, B. A. (2003). An Approach to Automate Requirements Elicitation and Specification.*Proceedings of the 7th International Conference Software Engineering and Applications.*

[9] Khwaja, A. A., Urban, J. E. (2009). Realspec: An Executable Specification Language for Modeling Control Systems. *ISORC,* pp.219-227.

[10] Parr, T. (2007). *The Definitive ANTLR Reference: Building Domain-Specific Languages.* Pragmatic Programmers. Pragmatic Bookshelf.

[11] Ramos, J. G., Silva, J., Vidal, G. (2004). An Embedded Language Approach to Router Specification in Curry. *Proc. of the 30th Conference on Current Trends in Theory and Practice of Informatics (SOFSEM 2004),* pp. 277-288. Springer-Verlag.

[12] *ThoughtWorks (2008). The ThoughtWorks' Anthology. Essays on Software Technology and Innovation.* Pragmatic Bookshelf.