Revista Eletrônica de Sistemas de Informação
ISSN 1677-3071

Vol 10, No 2

2011

Table of Contents

This is an electronic journal. There is no printed version of it. However, if you want to print a paper, you can still do it in an environmentally conscious way. It was desktop edited using *Eco Sans*, a font that requires less ink to be printed.

TOWARDS EASING THE INSTANTIATION OF APPLICATIONS USING GRENJ FRAMEWORK BY MEANS OF A DOMAIN SPECIFIC LANGUAGE

(submitted in December 2010)

Vinicius Humberto Serapilha Durelli
Instituto de Ciências Matemáticas e de Computação – Univ. de São Paulo (USP)
durelli@icmc.usp.br

Simone de Sousa Borges
Ciência da Computação – Universidade Federal de São Carlos (UFSCar)
simoneborges@acm.org

Rafael Serapilha Durelli
Ciência da Computação – Universidade Federal de São Carlos (UFSCar)
rafael_durelli@dc.ufscar.br

Rosana Teresinha Vaccare Braga
Instituto de Ciências Matemáticas e de Computação – UNIV. DE SÃO PAULO (USP)
rtvb@icmc.usp.br

ABSTRACT

White-box frameworks are a collection of extensible classes representing reusable designs that can be extended, to varying degrees, to instantiate custom-tailored software systems. Due to its inherent benefits (e.g., large-scale reuse of code, design, and domain knowledge), such domain-specific reuse approach has become a de facto standard to implement business systems. However, in order to fully realize the advantages of white-box frameworks, developers need to have substantial architectural and technical knowledge. In effect, developers must be familiar with the framework's extension points (e.g., hot spots) and how to program those extensions using the programming language in which the framework was implemented. GRENJ is a white-box framework implemented in Java. Thus, instantiating applications through such framework is quite complex and demands detailed architectural knowledge and advanced Java programming skills. In order to lessen the amount of source code, effort, and expertise required to instantiate applications by using GRENJ framework, we have developed a domain specific language that manages all application instantiation issues systematically. This domain specific language facilitates the application instantiation process by acting as a facade over GRENJ framework as well as providing the user with a more concise, human-readable syntax than Java. In this paper, we contrast the major differences and benefits resulting from instantiating applications solely using GRENJ framework and indirectly reusing its source code by applying our domain specific language.

Key-words: object-oriented framework; domain specific language; java; ruby

1 INTRODUCTION

Over the past decades, a myriad of software technologies have been devised for overcoming the intricacies of developing software systems. Among such technologies, reuse has been reckoned as one of the most important software technologies. Towards this end, a number of reuse techniques have been developed: software libraries, components, design patterns, etc. Currently, it is commonplace developing software systems as an assemblage of many preexisting elements and a few new ones pieced together. In practice, the lesser the amount of code developers have to implement from scratch, the shorter the time to market will be.

Object-oriented frameworks are a reuse technique that has been widely used by academics and practitioners alike. An object-oriented framework consists of a collection of several fully or partially implemented components that cooperate among themselves, thereby implementing a software architecture for a family of applications belonging to a specific domain. Frameworks have components that are designed to be either extensible or replaceable, these components are called variation points or hot spots of the framework (JOHNSON, 1997). Developers are able to customize and extend these variation points through application-specific source code, creating applications according to their needs. Thus, frameworks provide support for large-scale reuse of source code as well as their underlying architecture design (FAYAD *et al.*, 1999).

Despite the benefits provided by frameworks, instantiating applications is a complex task for which architectural knowledge is required. Since the most common way to instantiate applications using a framework is to inherit from abstract classes defined in the framework, the following issues hinder the instantiation process: *(i)* the lack of adequate documentation; *(ii)* the need to know where the customization source code should be written and which sort of code is needed to extend each variation point; *(iii)* the fact that variation points may either have interdependencies or be optional; *(iv)* the fact that frameworks may provide several ways of adding the same functionality; and *(v)* the fact that the implementation language compiler cannot verify instantiation restrictions, being unable to report instantiation error messages (FONTOURA *et al.*, 2000).

In order to overcome the difficulties related to application instantiation using frameworks, we propose an approach that is based on developing a domain specific language (DSL) so that it can manage all application instantiation issues systematically. A DSL is a programming language geared towards expressing more clearly a particular problem domain; conversely, conventional programming languages are more general purpose. Thus, the syntax and semantics of DSLs reflect concepts of their respective problem domain. A similar approach proposed by Fontoura *et al.* (2000) consists in creating a DSL for each variation point. In our approach, only one DSL, which acts as a facade over the framework being encapsulated, is implemented. Hence, by using a DSL the developer is able to describe the application being instantiated as concepts from the

application domain, not concepts of a general-purpose programming language. In summary, framework instantiation implies investment of the organization to make sure that software developers know enough details about the framework, but this might not guarantee that the instantiation is free of errors, as it is often a complex, repetitive and error-prone task.

Aimed at describing our approach and the DSL we have developed, the remainder of this paper is structured as follows. Section 2 presents background on DSLs. Section 3 describes researches that also focus on ways of easing the application instantiation process. Section 4 presents the GRENJ framework and gives an overview of previous research and technologies that played an important role during GRENJ development. Section 5 presents an evaluation performed to ascertain whether GRENJ framework domain is appropriate for being represented as a textual DSL, Section 6 highlights the major features of the DSL we have developed, and Section 7 contrasts the major differences between instantiating applications using GRENJ framework and applying our DSL. Section 8 takes a step back and presents the pros and cons of using DSLs in general. It describes such information in a high-level fashion so that managers and senior developers can reflect and decide whether they should invest in the implementation of DSLs for their underlying problem domains. Section 9 concludes the paper with an outlook on future work, some concluding remarks, and limitations of our DSL.

2 DOMAIN SPECIFIC LANGUAGES

Domain specific languages (DSLs) are small languages tailored towards better expressing concepts of a particular domain (VAN DEURSEN *et al.*, 2000). Usually, most of them are declarative and, consequently, they can be regarded as specification languages. These small, declarative, special-purpose languages have a simplified suite of notations that is tailored toward their domain abstractions, features, semantics, and jargon. Hence, by using DSLs, developers perceive themselves as dealing directly with domain concepts (SPRINKLE *et al.*, 2009).

DSLs can be divided into two groups: external and internal (FOWLER, 2009). External DSLs have their own custom-built syntax. As a consequence, developing an external DSL entails writing a full-fledged parser in order to process it. Internal DSLs use existing general-purpose language structures and, in most cases, the underlying execution environment, as a hosting base. A clear advantage of this approach is that the compiler or interpreter of the base language is reused. The main limitation is related to the limited expressiveness that can be achieved by using the base language syntactic mechanisms (VAN DEURSEN *et al.*, 2000).

The benefits of using DSLs include: *(i)* solutions can be expressed in a high abstraction level that encompasses domain idioms and jargons; *(ii)* DSL programs are concise and self-documenting; *(iii)* DSLs embody domain knowledge; *(iv)* it is possible to perform validation and optimization at the

domain level (MENON and PINGALI, 1999); and *(v)* DSLs enhance productivity, reliability, and maintainability (VAN DEURSEN and KLINT, 1998). However, it is worth noting that not all domains are appropriate for being represented as a DSL. A DSL approach is more suitable when: *(i)* the domain is well defined and it has repetitive elements; *(ii)* there is an intuitive or well accepted representation of the domain concepts; and *(iii)* the abstractions of the general-purpose language being used do not provide the required expressiveness (SPRINKLE *et al.*, 2009).

3 RELATED WORK IN FRAMEWORK INSTANTIATION

Frameworks have become a de facto standard to implement business systems. However, instantiating applications using frameworks entails a fair amount of technical and domain knowledge. Due to the fact that white-box frameworks are a collection of extensible classes representing reusable designs, they demand even more knowledge in order to be instantiated. As mentioned, even architectural knowledge is required. Thus, such complexity has spurred a growing interest in approaches tailored to overcoming the technical hurdles of instantiating white-box frameworks. Most of these approaches draw the information required for instantiating applications from the framework documentation.

In the context of white-box frameworks, this information basically consists of the framework class hierarchy, the inter-dependent abstract classes that need to be subclassed in the new application, the methods to be overridden in these classes, and examples of applications derived from the framework. The methods that act as extension points of the framework are called hook methods (PREE *et al.*, 1995). By overriding these methods developers can add custom-tailored behavior. Some of the possible types of instantiation approaches are based upon: *(i)* studying the framework source code and its documentation; *(ii)* exploring exemplars; *(iii)* cookbooks; *(iv)* patterns; and *(v)* pattern languages.

The first approach consists in studying the framework documentation and the framework itself, i.e., its class hierarchy, source code, and other documents (Johnson, 1992). Conventional training or special tutorials are ways of achieving the required knowledge. The main drawbacks of this approach are the time required to properly learn the framework from the ground up and the difficulty to determine whether the newly acquired comprehension is enough to begin to use the framework.

Examining existing applications built with the framework in order to identify what needs to be adapted to obtain the custom-application is another possible approach. Nevertheless, the exploration of exemplars has two shortcomings: *(i)* its is difficult to find an application that has all the particular functionalities that need to be implemented and *(ii)* when the functionality is present in an example it may have additional features that are not needed, thus, the user has to know what can be removed without

affecting the functionality. An example of this approach is given by Gangopadhyay and Mitra (1995).

Cookbooks are a sort of clear-cut documentation that describes the tasks and configurations required to instantiate applications. Usually, this information is conveyed in a stepwise fashion – like in a recipe. Several researches have been conducted aimed at evaluating this instantiation approach (PREE *et al.*, 1995; ORTIGOSA *et al.*, 2000). Two limitations of this approach are as follows: *(i)* difficulty in finding the correct "recipe" and *(ii)* some tasks and configurations cannot be performed step by step.

According to Johnson (1992), patterns document frameworks and help to ensure the correct use of their functionalities. Nevertheless, patterns are situated in a lower abstraction level than frameworks. Moreover, since frameworks may be quite complex, usually it is not possible to document the overall design as a set of unrelated patterns, instead they should be related to each other in the documentation. Thus, pattern languages are a more suitable technique for documenting frameworks.

Brugali and Sycara (2000) argue that if a framework is developed based on a pattern language, this pattern language can be used to guide the instantiation process by providing: *(i)* domain-specific advices and *(ii)* information on the design of the framework in terms of objects and their relationships. Braga and Masiero (2002) capitalize on this idea and try to support framework development and instantiation based on pattern languages and a well-defined process. The proposed process encompasses: *(i)* analysis by means of following and applying the patterns of the underlying pattern language; *(ii)* mapping between the analysis model, produced during the previous step, and corresponding framework classes; *(iii)* details concerning the implementation of specific classes according to the requirements of the application under development; and *(iv)* testing.

An advantage of Braga and Masiero's (2002) approach is that the framework user knows exactly where to begin the instantiation since the pattern language guides him/her through the several parts that need to be adapted in the framework hierarchy. The instantiation is focused on the functionality required and there is a clear notion of which requirements are attended by each pattern. However, applying this approach does not help to overcome technical problems associated with the instantiation process, i.e., properly using the programming language at each framework hot spot.

In another related work Fontoura *et al.* (2000) also propose using DSLs in order to overcome difficulties from instantiating applications using frameworks. The proposed approach uses DSLs only to describe hot spots, thereby instantiating applications involves describing the desired functionality by means of several DSLs. During instantiation time, DSLs are transformed to generate the framework instantiation code.

As for our approach, it relies on introducing just one DLS atop a framework, aiming at providing a suite of notations that is tailored towards

the underlying domain abstractions. Thus, a clear advantage of our approach is that it obviates the need for knowing and using more than one DSL.

4 GRN, GREN, AND GRENJ

GRENJ (DURELLI, 2008) is a white-box framework that results from the reengineering of a framework implemented in Smalltalk (i.e., GREN) which has been developed based on a pattern language (i.e., GRN) (BRAGA, 2002). Therefore, both frameworks and GRN belong to the same domain, namely, business resource management. This domain encompasses applications where resources (e.g., assets or services) can be purchased, sold, rented, or fixed; thereby many different systems can be instantiated from it.

GRENJ has more than twenty-nine thousand lines of Java source code and its architecture consists of two layers: persistence and business. Moreover, unit tests cover almost 85% of the framework's source code. In the business layer, there are implementations of each of the fourteen GRN patterns. Most of the classes in this layer represent elements of some GRN pattern and are abstract so that they can be extended for generating specific applications. To properly instantiate applications, the user must be familiar with GRN and should have a fair knowledge of GRENJ architectural details; let alone having knowledge of several advanced Java features, e.g., generics and reflection application programming interface (API). To cope with these difficulties, we devised a DSL that encapsulates all details concerning application instantiation. However, before delving into details of our DSL, in the next section we canvass whether the domain, as dealt and represented by GRN, deserves to be referred to as a textual DSL.

5 GRN DOMAIN EVALUATION

Sprinkle *et al.* (2009) discuss a series of questions intended as a checklist for ascertaining whether a problem merits a DSL approach. The items of such a list that have been considered can be summarized by the following questions: *(i)* "Is the domain well-defined?"; *(ii)* "Does the domain have repetitive elements or patterns, such as multiple products, features or targets?"; *(iii)* "Is there a clear path from requirements' analysis and specification to execution?"; and *(iv)* "Is there an intuitive and well-accepted representation?".

GRN patterns and the way they are organized capture and concisely convey information on the business resource domain. In addition to it, GRN provides a path that emphasizes the identification of concepts that can be regarded as "*resources*". After identifying these concepts, for each potential resource, the user iterates throughout the pattern language coherently applying the patterns. Hence, we can conclude that questions *(i)* through *(iii)* can be positively answered. Nevertheless, taking into consideration item *(iv)*, it is worth noting that there is no available representation apart from the analysis-level class diagrams provided by

GRN to illustrate each pattern. We have not emphasized item *(iv)* since we intend to implement a textual DSL. We argue that an intuitive, well-accepted graphical representation is not imperative for creating the DSL syntax.

Given that most of the checklist items have been regarded as applicable to GRN and consequently to the GRENJ domain, we have developed a textual DSL in order to lessen the effort required to instantiate applications using GRENJ. Information on the implementation of such DSL is presented in the next section.

6 RM-DSL IMPLEMENTATION

Our domain specific language is called *resource management Domain Specific Language* (rm-DSL). We have chosen to implement an internal DSL (i.e., adapting an existing general-purpose language by adding or changing methods, operators, and other structures), thus rm-DSL was built on top of the Ruby programming language (FLANAGAN and MATSUMOTO, 2008). Moreover, in order to support the development and design of our DSL, we have consulted several DSL design patterns described by Spinellis (2001). Along this section, as we describe the DSL implementation, we also briefly mention the patterns applied.

The most important points concerning the implementation of a DSL on top of an existing language are described by the structural pattern *Piggyback* (SPINELLIS, 2001). The use of this pattern consists simply in obtaining all standardized support for common syntactical elements from the hosting language. Hence, taking advantage of several Ruby language structures, we have designed rm-DSL so that it provides a notation intended to reduce the semantic distance between the problem domain and the solution domain, easing the instantiation of applications using GRENJ framework by hiding details related to the framework and its intricacies.

Rm-DSL uses code templates containing valid subclasses of GRENJ framework classes which, usually, are extended and have their hook methods overridden during application instantiation. These code templates have *lexical hints*, which point out chunks of code that must be customized according to the application being instantiated. The notation used is as follows: every element preceded with # is replaced by a value provided by the user during application instantiation by means of the rm-DSL. In Listing 1 we show an example of the sort of code template used by the DSL. In this chunk of code, all occurrences of *#class_name* are replaced by the resource name supplied during instantiation. In this example GRENJ is being instantiated to a DVD rental store, where Movie is playing the role of Resource. The pattern classes have fixed attributes, but during instantiation new attributes can be added. The lexical hints *#attributes* and *#attribute_initializations* are replaced by attribute declarations and attribute initializations, respectively. These lexical hints represent the added

attributes in order to customize the resource being instantiated. It is worth noting that the code templates used by rm-DSL can also be considered a DSL. More specifically, it can be regarded as an external DSL that applies the *Lexical Processing* pattern (SPINELLIS, 2001) since it is geared towards lexical translation by using a notation based on lexical hints; in this case, the prefix character #.

For instance, the chunk of code shown in Listing 2 can be generated from the rm-DSL code shown in Listing 3. The utilization of our DSL consists in instantiating implementations of GRN patterns and adding attributes to these instantiations in order to customize them. At line 4 of Listing 3, an instantiation it is shown of the *Identify the Resource* pattern from GRN (BRAGA, 2002). In such a context, the resource being instantiated is a *movie* and it has a string as attribute which describes its *synopsis*. During the addition of attributes, the user is able to specify other properties related to them, e.g., access modifier and whether it is required to generate getters and setters methods. As can be seen from lines 5 to 7 of Listing 3, attributes are added using the += operator. Our DSL takes advantage of the fact that Ruby implements a number of its operators as methods (FLANAGAN and MATSUMOTO, 2008), allowing classes to define new meanings for these operators.

```
9  ...
10 public class #class_name extends Resource {
11   #attributes
12   public #class_name() {
13     super();
14     #attribute_initializations
15   }
16 ...
```

Listing 1. Chunk of a code template used by rm-DSL

```
9  ...
10 public class Movie extends Resource {
11   private String synopsis;
12   public Movie() {
13     super();
14     synopsis = "";
15   }
16 ...
```

Listing 2. Resulting code from the rm-DSL code in Listing 3

doi:10.5329/RESI.2011.1002007

```
3 ...
4 instance = IdentifyResource.new "Movie"
5 instance += {:type => :string, :name => :synopsis,
6              :access_modifier => :private,
7              :generate_getters_and_setters => true}
8 ...
```

Listing 3. Instantiating the *Identify the Resource* pattern and adding an attribute to it

As mentioned, the information needed to instantiate applications is drawn from certain key points of rm-DLS programs (*.rb* files). In order to generate the code shown in Listing 2, information that varies according to the application being instantiated has to be explicitly specified, e.g., *(i)* name of the class to be generated, *(ii)* its attribute names, *(iii)* and whether it is necessary to generate methods to get and set the value of each attribute. As illustrated in the overview in Figure 1, such information is used to replace the code template's lexical hints, thereby generating the resulting Java code.

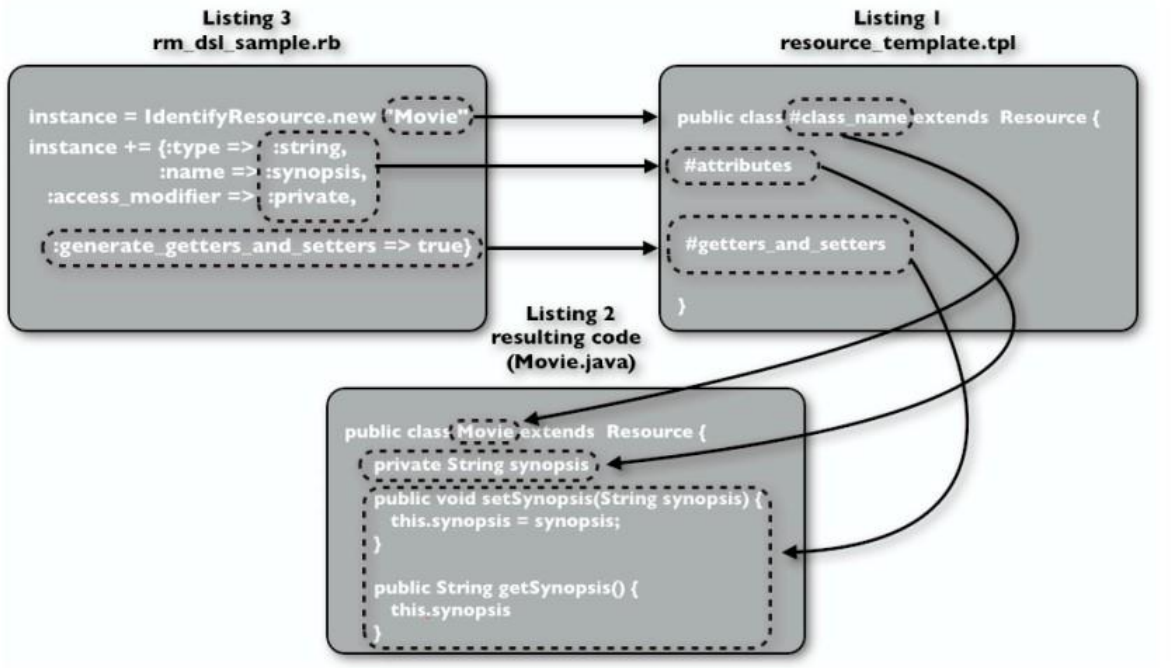

Figure 1. rm-DSL overview: the interaction among the involved files

7 CONTRASTING INSTANTIATION USING GRENJ AND RM-DSL

In this section we highlight the main particularities of instantiating applications both using GRENJ framework (i.e., through extending framework superclasses and overriding hook methods) and rm-DSL. In order to do that, we have instantiated the class diagram shown in Figure 2 using both

foregoing approaches. The underlying class diagram represents part of the functionalities required by a DVD rental store and has been created applying GRN patterns. Inside the arrows, the following format has been adopted: P#n: *role*, where *n* is the pattern number in the context of GRN and *role* is the "*role*" played by this class in the underlying pattern. The added attribute is depicted in a lighter shade of gray.

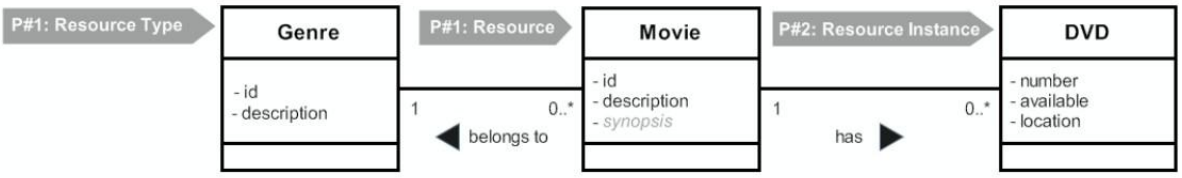

Figure 2. DVD rental store modeled applying GRN patterns

Implementing such an application using GRENJ requires extending three classes. In each extended class, it is necessary to implement three distinct constructors, i.e. a default constructor that has no parameters, one that has all added attributes passed as parameters, and one that receives instances of *java.sql.ResultSet* and *grenj.util.Index*. It is also necessary to implement all getters and setters methods. Moreover, for implementing persistence, in each class, the following methods have to be overridden: *insertionFieldClause*, *insertionValueClause*, and *updateSetClause*. In the context of the *Movie* class, it is also necessary to override the method *getResourceInstanceClass* in order to indicate which class represents a resource instance; in this case, the method must be overridden so that it returns an instance of *DVD*. The number of classes and methods that have to be implemented are summarized in Table 1. Given that the developer needs to implement many methods, this approach results in a lot of effort and source code. This large amount of Java source code that needs to be implemented makes this approach error-prone.

Table 1. Classes and methods that have to be implemented during the instantiation using GRENJ framework

Class	Added Attributes	Added Methods	Lines of Code
Movie	1	12	279
DVD	0	6	134
Genre	0	6	89
Total	1	24	502

By applying our DSL the user needs to have knowledge of neither GRENJ architecture nor Java programming language. In addition, application instantiations using rm-DSL have less lines of code and the resulting source code is more human-readable. The DVD rental store depicted in Figure 2 can be instantiated using rm-DSL as shown in Listing 4. In the context of rm-DSL, it consists simply in creating instances of each pattern as if they were simple classes (e.g., lines 6 and 8), whereas using GRENJ framework mandates the implementation of new classes that have

doi:10.5329/RESI.2011.1002007

to be fairly customized at each hot spot. Therefore, instantiating applications using GRENJ may be a rather cumbersome activity.

```
require "identify_resource"
require "code_generator"
require "resource_instance"

instance =
   IdentifyResource.new("Movie", {:type => :instantiable})

instance.resource_instance = ResourceInstance.new "DVD"

instance+= {:type => :string, :name => :synopsis,
            :access_modifier => :private,
            :generate_getters_and_setters => true}

instance.types= ["Genre"]

code_generator = CodeGenerator.new
code_generator.generate_without_validation [instance]
```

Listing 4. Instantiating the DVD rental store depicted in Figure 2 using rm-DSL

8 THE BENEFITS DSLS MAY PROVIDE FOR ENTREPRENEURS: DRIVING THE INSTANTIATION PROCESS FROM THE PERSPECTIVE OF NONTECHNICAL PERSONNEL

This section is intended to outline the benefits of using DSLs from either a manager or a businessman viewpoint. It is important to emphasize that the following discussion presents such benefits in a broader sense. That is, we do not emphasize only the advantages of DSLs that act as facades for frameworks (e.g., rm-DSL), rather we center around describing general advantages. We also outline some of the drawbacks of using DSLs so that entrepreneurs can weigh the pros and cons and decide whether they are applicable to their circumstances.

One of the advantages of using a DSL is that it improves the communication between developers and project stakeholders. In fact, DSLs make the collaboration with business users more approachable by acting as a common bridge of understanding between the developers and the domain experts. Due to the fact that DSLs share a common vocabulary with the problem domain, nontechnical people as business users can cooperate with developers and programmers alike. This opens up possibilities for having domain experts validate the domain rules as they are being programmed without relying upon high-level documentation tailored to nontechnical people. For instance, the domain expert group can verify test cases as they are developed, further ensuring that the underlying software system is in conformance with the intricate domain

rules. Moreover, such a common terminology makes it possible to easily trace artifacts from the problem domain to its respective representation in the solution domain. As far as we are concerned, establishing an improved communication medium throughout the development cycle is the major benefit of using DSLs.

As previously mentioned, there is also evidence that using DSLs improves productivity. Such improvements have been reported in a myriad of domains, including digital signal processing, telecommunications, electrical utilities, and home automation. As described by Sprinkle *et al.* (2009), Nokia and Panasonic have reported significant productivity improvements. Therefore, managers seeking to boost productivity may be willing to sacrifice more time up-front in order to develop a DSL comprising the problem domain terminology; the resulting DSL is likely to boost productivity as well as bring other of its inherent benefits. However, how much of such improvements in productivity can be attributed to better communication between the involved parts are unknown. Thus, we argue that this is a topic of much-needed exploration.

As for disadvantages, the most obvious one is poor performance. Since DSL entails an additional layer it may incur in not-so-negligible overheads during runtime. Thus, before considering creating a DSL, managers and senior developers must be able to ascertain whether the computation overhead introduced by DSLs is not going to get in the way of achieving the expected response times and performance.

After determining that performance is not a central issue, managers and senior developers have to take stock of the complexity of the domain being dealt with. Given that DSL have an upfront-cost, it may not make sense to waste time creating DSLs for straightforward domains. We contend that only marginal benefits would be achieved in cases where the domain rules are well-known.

Another issue regarding DSLs is the lack of tooling support. For instance, since DSLs are in-house creations, usually they are not supported by mainstream integrated development environments (IDEs). Therefore, features geared towards simplifying and speeding up the creation of source code such as syntax highlighting, autocomplete, and refactoring are not available and, when required, have to be developed from the ground up. Naturally, this lack of support is ameliorated when internal DSLs are employed: a subset of the support provided to the underlying programming language can be used.

9 CONCLUDING REMARKS

Learning to use an object-oriented framework effectively requires considerable investment of effort. Besides, due to the large amount of customization source code required for instantiating each application, this process tends to be error-prone. Aiming at overcoming these problems, we propose the use of a DSL as a facade over the framework being

encapsulated, thereby concealing details related to the underlying framework and its intricacies. Such a DSL must be sufficiently expressive to support the description of all possible combinations of valid instantiations.

In order to prove the feasibility of the proposed approach, we have presented a DSL to lessen the amount of Java source code and effort needed to instantiate applications using GRENJ framework. Such a DSL, which is called rm-DSL, encompasses domain concepts and provides the user with a more concise, human-readable syntax than Java. Through rm-DSL we have shown that it is possible to reuse GRENJ framework source code indirectly through code templates containing valid chunks of GRENJ subclasse code and lexical hints that are replaced according to instantiation needs. Hence, rm-DSL and GRENJ framework synergistically produce a more flexible approach for instantiating applications.

A shortcoming of our DSL is that it covers only six of the fourteen patterns implemented on GRENJ framework. Therefore, as a future work, we intend to implement the remaining patterns. Another considered extension is to add validation functionalities, allowing rm-DSL to determine whether an instantiation is in compliance with GRN criteria, thereby providing the user with instantiation error messages. Moreover, we aim at conducting case studies for evaluating the effectiveness and the amount of reuse that can be achieved by using our DSL in contrast with solely using GRENJ framework.

REFERENCES

BRAGA, Rosana Teresinha Vaccare. A process for creating and instantiating frameworks that are based on domain-specific pattern languages (In Portuguese). 2002. PhD thesis - ICMC/USP, São Carlos –SP. 2002.

BRAGA, Rosana Teresinha Vaccare. MASIERO, Paulo César. The role of pattern languages in the instantiation of object-oriented frameworks. *Advances in Object-Oriented Information Systems*, 20426, p. 403–410, 2002.

BRUGALI, David; SYCARA, Katia. Frameworks and pattern languages: an intriguing relationship. *ACM Computing Surveys*, v. 32, March, 2002.

DURELLI, Vinicius Humberto Serapilha. GRENJ: A framework resulting from a iterative reengineering process applying TDD (In Portuguese). 2008. Master thesis - Departamento de Computacao, Universidade Federal de São Carlos (UFSCar), São Carlos, 2008.

FAYAD, Mohamed; SCHMIDT, Douglas. Building application frameworks: object-oriented application frameworks. *Communications of ACM*, v. 40, n. 10, p. 32-38, October, 1997.

FLANAGAN, David; MATSUMOTO, Yukihiro. *The Ruby programming language*. O'Reilly Media Inc., 2008.

FONTOURA, Marcus; BRAGA, Christiano; MOURA, Leonardo; LUCENA, Carlos. Using domain specific languages to instantiate object-oriented frameworks. *Software*, v. 147, n. 4, p. 109-116, August, 2000.

FOWLER, Martin. A pedagogical framework for domain-specific languages. *IEEE Software*, v. 26, n. 4, p. 13–14, August, 2009. doi: http://dx.doi.org/10.1109/MS.2009.85

GANGOPADHYAY, Dipayan; MITRA, Subrata. Understanding frameworks by exploration of exemplars. In: International Workshop on Computer-Aided Software Engineering, 7., Toronto. 1995.

JOHNSON, Ralph. Documenting frameworks using patterns. In: OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications, p. 63–76. ACM, 1992.

JOHNSON, Ralph. Frameworks = (components + patterns). *Communications of the ACM*, v. 40, n. 10, p. 39–42, October, 1997. doi: http://dx.doi.org/10.1145/262793.262799

MENON, Vijay; PINGALI, Keshav. A case for source-level transformations in matlab. In: PLAN '99: Proceedings of the 2nd conference on Domain-specific languages, p. 53–65. ACM, 1999.

ORTIGOSA, Alvaro; CAMPO, Marcelo; MORIYON, Roberto. Towards agent-oriented assistance for framework instantiation. In: OOPSLA '00: Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications, p. 253–263. ACM, 2000.

PREE, Wolfgang; POMBERGER, Gustav; SCHAPPERT, Albert; SOMMERLAD, Peter. Active guidance of framework development. *Software – Concepts and Tools*, v. 16, n. 3, p. 94-103, 1995.

SPINELLIS, Diomidis. Notable design patterns for domain-specific languages. *Journal of Systems Software*, v. 56, n. 1, p. 91–99, February, 2001. doi:http://dx.doi.org/10.1016/S0164-1212(00)00089-3

SPRINKLE, Jonathan; MERNIK, Marjan; TOLVANEN, Juha-Pekka; SPINELLIS, Diomidis. Guest Editors' Introduction: what kinds of nails need a domain-specific hammer? *IEEE Software*, v. 26, n. 4, p. 15–18, July/August, 2009. doi: http://dx.doi.org/10.1109/MS.2009.92

VAN DEURSEN, Arie; KLINT, Paul. Little languages: little maintenance? *Journal of Software Maintenance: Research and Practice*, v. 10, n. 2, p. 75–92, 1998.

VAN DEURSEN, Arie; KLINT, Paul; VISSER, Joost. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, v. 35, n. 6, p. 26–36, June, 2000.