The Relationships between Domain Specific and General-Purpose Languages

Oded Kramer and Arnon Sturm

Department of Information Systems Engineering, Ben-Gurion University of the Negev
Beer-Sheva, Israel
odedkr@bgu.ac.il, sturm@bgu.ac.il

Abstract. Domain Specific Languages (DSLs) are used for increasing productivity, improving quality and maintainability, and reusing experts' knowledge. Current DSL development approaches reveal a tradeoff between the efforts required for developing a DSL to the benefits the DSL will introduce. This tradeoff is directly linked to the relationship between the DSL to general purposed programming languages (GPPL). In this paper, we proposed a framework for integrating the benefits of various DSL approaches.

Keywords: domain specific language, general-purpose programming language

1 Introduction

Domain Specific Languages (DSLs) are computer languages that are tailored to specific domains [1, 6]. The use of such languages leads to increased productivity, improved quality, better maintainability and reuse of experts' knowledge.
According to Jones [4], Basic and COBOL improved the productivity of programming languages by 400% compared to assembly (measured by the average number of source statements per function points). The explanation for this is that the new programming languages raised the level of abstraction that was in assembly. This allows the programmers to think and act in "higher" terms and avoid the complexity of assembly. This is possible since complexity is handed over to the compiler, which automatically transforms the high level source code to low level byte code. Moreover, DSLs are tailored to specific domains, reducing efforts of mapping general purposed concepts to the relevant domain. Thus, the ease of use and reuse increases and leads to better productivity and maintainability [6].

Current DSL development approaches reveal a tradeoff between the efforts required for developing a DSL to the benefits the DSL will introduce. This tradeoff is directly linked to the relationship between the DSL to general purposed programming languages (GPPL). On the one hand DSLs which are not related to GPPLs introduce significant benefits but are extremely difficult to develop [3]. On the other hand, DSLs which are embedded in GPPLs are relatively easy to develop but their benefits are questionable and unproven.

In this paper, we advocate for examining the integration of the two aforementioned approaches for increasing the usage of DSLs. The structure of the paper is as follows. Section 2 discusses related work concerning internal and external DSLs, exploring their benefits and their limitations. Section 3 provides an overview of the proposed framework based on the idea of engineering DSL over a GPPL and Section 4 concludes.

2 Related Work

As mentioned before, DSL development approaches are different with respect to the extent that they are related to GPPLs. These can be classified as external and internal DSLs. In the following we elaborate on these DSLs.

2.1 External DSLs

The basic premise of external DSLs is that the underlying principles of higher abstraction level and tailoring to specific domain necessitate the development of DSL from scratch. Typically, there would be a domain expert whose expertise is on the semantics of the domain and an expert programmer whose expertise is on developing complicated and sophisticated software[1] working on this process [1]. The design process includes defining the domain concepts their relationships, semantics, notations, and constraints. The implementation process includes building a code generator, an optional domain specific framework (domain specific code which is designed to be reused across the domain's applications), and the DSL's integrated development environment (IDE) which includes the DSL’s supporting tools.

The main two advantages of external DSLs are the improved productivity; reports have shown of increase in productivity of 300%-1000% [1] and enhanced application quality; due to be a preliminary check of the model's consistency according to domain rules. Ideally the domain rules were determined by the domain experts, and the pre-checking tool was built by an expert programmer. This means that much of the programmers' mistakes can be detected and thus avoided at this early stage of development. The modelers specify the solution on a higher level, which is then transformed automatically to the source code. This means that they can avoid dealing with important but complicated issues such as design principles, and architecture, as they are handled by the code generator. An empirical study by Kieburtz et al. [5], which compared DSLs to manual programming found an improvement in quality of code (measured by number of failed tests) of more than 50% in favor of the DSLs.

Yet, external DSLs also suffer from various limitations. As mentioned the design and implementation of external DSL is by no means simple, it is complicated and time consuming. Even if the work is done by experts (both domain and programming), and some supporting tools are available it might not be enough to

[1] obviously, they could be the same person, however both kinds of expertise are required

ensure a successful working DSL. According to [3] most DSLs are usually abandoned in the development process and the work is done eventually in regular general purpose language. Additionally, to justify economically the investment of the DSL development process a quota of applications has to be exceeded. While this is true for all domain engineering techniques it is as harsh as the amount of emphasis that is put on the domain engineering process. As [2, 3] found, external DSL approaches put more emphasis than others. Moreover, introducing the notion of DSL based development into an organization requires a significant change in the organization's development paradigm. This change requires both new tools and new processes. While some managers will be able to see the long terms advantages of DSLs, other might be reluctant to introduce radical, expansive and time consuming changes to their natural development process. Another limitation of external DSL approaches is the narrow of expressiveness given to the application developer. While this frequently is considered to be an advantage – limiting the application developer expressiveness means guiding him and controlling the quality of his work, by that increasing his productivity and the overall quality of the end products – we consider this to be a disadvantage as well. The restrictions imposed on the application developer are achieved by designating the domain to include a set of pre formulated commonality and variability. This means that the application developer is restricted to a close set of variants that he can express. In case the application developer wishes to express a newly encountered feature she has to inform the DSL developer to update the DSL and wait for the change to be done. This process is time consuming and more importantly will make the procedure of incorporating new variants into the domain difficult, ultimately leading to narrow domains.

2.2 Internal DSLs

Internal DSLs drew their inspiration from the recognized drawbacks of external DSLs. Their basic premise is that DSLs should not be developed from scratch; rather they should be embedded on existing proven general purposed programming languages (GPPLs). In this sense internal DSLs are no different than regular domain specific application programming interfaces, also known as library code. However they are different in the sense that the APIs are designed to have a language like flow to them. This is achieved by advanced coding techniques such as method chaining, expression builders, interface chaining, generics, etc. When these techniques are used correctly some domain semantics could be validated in compile time.

The main advantage of internal DSLs is that they do not suffer from the above mentioned drawbacks of external DSLs. This is caused by three main reasons: (1) The development of internal DSLs is much easier with respect to external DSLs mainly because the GPPL facilities (such as IDEs and compilers,) already exist; (2) Internal DSLs do not necessitate a radical change in the organization's natural development paradigm as they permit using the same set of tools (such as a programming languages, IDEs, and compilers); and (3) Internal DSLs do not limit the application developer expressiveness as she is free to use the GPPL regularly anytime she wants.

However, internal DSLs introduce the following limitations: (1) Current reports [2, 3] of internals DSLs focused on code readability and maintainability. Although this should have positive effects over productivity it is hard to see how sophisticated APIs raise the level of abstraction similarly to specially tailored DSLs; (2) External DSLs achieved improved code quality through pre code generating validation algorithms and higher abstraction levels. Although internal DSLs can exploit coding techniques to assure some domain semantics with type safety and compile time error reports, they cannot implement validation algorithms that examine the specified code according to domain constraints. The application programmer can use the API the way she sees fit.

In this paper, we proposed to integrate both internal and external DSLs benefits into one coherent framework.

3 Domain Specific Language based on A General-Purpose Language

In order to gain benefits from both approaches we propose to follow the underlying principles:

1. A framework which is embedded into a GPPL thus gaining the benefits of internal DSLs.
2. A framework which enables capturing and reusing domain experts' knowledge in the form of domain rules and constraints.
3. A framework which enables validating the application according to domain rules and constraints, thus detecting coding errors at an early stage of the development and increasing productivity.

To address these principles we adopt a domain modeling approach called Application-based DOmain Modeling (ADOM) [7, 8] and apply it to Java as a GPPL.

3.1 The ADOM Approach

The Application-based Domain Modeling (ADOM) is rooted in the domain engineering discipline, which is concerned with building reusable assets on the one hand, and representing and managing knowledge in specific domains on the other hand. ADOM supports the representation of reference (domain) models, construction of enterprise-specific models, and validation of the enterprise-specific models against the relevant reference models.

The architecture of ADOM is based on three layers: The ***language layer*** comprised of metamodels and specifications of the used languages. The ***domain layer*** holds the building elements of the domain and the relations among them. It consists of specifications of various domains; these specifications capture the knowledge gained in specific domains in the form of concepts, features, and constraints that express the commonality and the variability allowed among applications in the domain. The structure and the behavior of the domain layer are modeled using a language defined in the language layer. The ***application layer*** consists of domain-specific applications, including their structure and behavior. The application layer is specified using the knowledge and constraints presented in the domain layer and the constructs specified in the language layer. An application model uses a domain model as a validation template. All the static and dynamic constraints enforced by the domain should be applied in any application of that domain. In order to achieve this goal, any element in the application is classified according to the elements declared in the domain.

For describing variability and commonality, ADOM uses a multiplicity indicator that can be associated to all elements, including classes, attributes, methods, and more. The multiplicity indicators in the domain aim to represent how many times an element of this type may appear in an application. This indicator has two associated tagged values - min and max - which define the lowest and the upper most multiplicity boundaries.

The relations between a generic (domain) element and its specific (application) counterparts are maintained by a classification mechanism: each one of the elements that appears in the domain can serve as a classifier of an application element of the same type (e.g., a class that appears in a domain may serve as a classifier of classes in an application). The application elements are required to fulfill the structural and behavioral constraints introduced by their classifiers in the domain. Some optional generic elements may be omitted and not be included in the application, while some new specific elements may be inserted in the specific application; these are termed application-specific elements and are not classified in the application.

ADOM also provides validation mechanism that prevents application developers from violating domain constraints while (re)using the domain artifacts in the context of a particular application. This mechanism also handles application-specific elements that can be added in various places in the application in order to fulfill particular application requirements.

While ADOM is general and language-independent, a specific language needs to be selected as a basis for a workable dialect of ADOM. In order to apply ADOM, the only requirement from the associated language is to have a classification mechanism that enables categorization of elements.

3.2 ADOM-Java Dialect

Selecting ADOM for the purpose of devise an integrated framework for DSLs, we select Java as the language and the classification mechanism that we used in that case is the Java annotation. The annotation within the domain code consists of a multiplicity annotation that determines the number of elements of a type that should appear any application in that domain. This is akin to the multiplicity indicator of ADOM. In Figure 1 a partial code of the domain of access control is defined as discussed in [9]. The code specifies the domain knowledge and constraints. The `MachineInfo` class should appear at least once within an application within that domain as the multiplicity annotation dictates. Within that class there should be between one to five identifiers and exactly one balance attribute. In addition, an identification method is optional in that domain, yet it appears in the domain as a guideline. The `itemAvailabilityChecking` should appear between one to two times in each class of type `MachineInfo` and should consist of maximum one parameter.

```
@multiplicity(min = 1, max = -1)
public class MachineInfo {

    @multiplicity(min = 1, max = 5)
    private String machineIdentifier;

    @multiplicity(min = 1, max = 1)
    private double balance;

      @multiplicity(min = 0, max = 1)
    public boolean identification(@multiplicity(min = 1, max = 1)
                                  int itemIdentifier) {}
```

Figure 1. A domain level code

```
@MachineInfo
public class DVMInfo {

    @machineIdentifier
    String location;

    @balance
    double amountInDVM;

    @itemAvailabilityChecking
    public int isProductAvailiable(int productID) {}

    @itemAvailabilityChecking
    public int isChangeAvailable() {}
```

Figure 2. An application level code

The annotation within the application code uses the domain element name in order to associate the application element with a domain element. In **Figure 2** a partial code of a drinking vending machine application is depicted. This code is derived (or instantiated) from the domain appears in Figure 1. The `DVMInfo` is classified as a `MachineInfo`, the `location` attribute is classified as `machineIdentifier`, the `amountInDVM` attribute is classified as `balance`, the `isProductAvailable` and `isChangeAvailable` methods are classified as `itemAvailabilityChecking`. The `operate` method is not classified as is specific to the application. The classification aforementioned tied the application to the domain enabling the validation of the application versus the domain constraints. In this simple example, it is clear that the application code adheres with the constraints appear within the domain code. Even though the identification method in the domain code does not appear in the application code and that the application code consists of additional method (i.e., operate), the application code is valid with respect to the domain code as it does violate the domain code constraints.

4 Summary

Current DSL frameworks suffer from the tradeoff between the efforts required for developing a DSL and using that DSL. In this paper, we propose a framework for developing and using DSL based on a domain modeling approach, called ADOM. The framework is demonstrated on Java as a representative to a GPPL.

Having introduced that framework it gains the benefits of an internal DSL as it is embedded into Java, as well as the benefits of an external DSL as the ADOM approach supports the guiding and validating applications with respect to a specific domain.

Although the new framework seems promising for the task of designing and using DSLs, it is clear that additional annotations should be added in order to fully explore its capabilities for the purpose of increasing programming productivity and code quality. Also, additional evaluation of the proposed approach should be made.

References

1. Kelly, S. and Tolvanen, J-P. Domain-Specific Modeling: Enabling Full Code Generation, Wiley, 2008.
2. Kabanov, J. and Raudjärv, R. Embedded typesafe domain specific languages for Java. In *Proceedings of the 6th international Symposium on Principles and Practice of Programming in Java* (Modena, Italy, September 09 - 11, 2008). PPPJ '08, vol. 347. ACM, New York, NY, 189-197, 2008.
3. Freeman, S. and Pryce, N. Evolving an embedded domain-specific language in Java. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA, October 22 - 26, 2006). OOPSLA '06. ACM, New York, NY, 855-865, 2006.
4. Jones, C., SPR Programming Languages Table. Release 8.2, 1996.
5. Kieburtz, R. B., McKinney, L., Bell, J. M., Hook, J., Kotov, A., Lewis, J., Oliva, D. P., Sheard, T., Smith, I., and Walton, L. 1996. A software engineering experiment in software component generation. In *Proceedings of the 18th international Conference on Software Engineering* (Berlin, Germany, March 25 - 29, 1996). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 542-552.
6. Mernik, M., Heering, J., and Sloane, A. M.,When and how to develop domain-specific languages. *ACM Comput. Surv.* 37, 4 (Dec. 2005), 316-344, 2005.

7. Reinhartz-Berger, I. and Sturm, A., Enhancing UML Models: A Domain Analysis Approach, Journal of Database Management (JDM), vol. 19 (1), 2007, pp. 74-94.
8. Reinhartz-Berger, I. and Sturm, A., Utilizing Domain Models for Application Design and Validation, Information & Software Technology, vol. 51 (8), 2009, pp. 1275-1289.
9. Duffy, D. J. Domain Architectures: Models and Architectures for UML Applications. John Wiley & Sons, 2004.