A Domain Specific Language for Lessening the Effort Needed to Instantiate Applications Using GRENJ Framework

Vinícius H. S. Durelli[1], **Rafael S. Durelli**[2],
Simone de Sousa Borges[2], **and Rosana T. V. Braga**[1]

[1]Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo (ICMC-USP)
13560-970 – São Carlos – SP – Brasil
{durelli,rtvb}@icmc.usp.br

[2]Departamento de Computação
Universidade Federal de São Carlos (DC-UFSCar)
13565-905 – São Carlos – SP – Brasil
{rafael_durelli,simone_borges}@dc.ufscar.br

Abstract. *GRENJ is a white-box framework implemented in Java. White-box frameworks are reusable designs composed of a set of concrete and abstract classes so that the collaboration among these classes provides support for large-scale reuse of design and source code. However, instantiating applications by using this sort of framework is quite complex and demands detailed architectural knowledge. 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.*

1. Introduction

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 developer has to know where the customization source

code should be written and which sort of code is needed to extend each variation point; *(iii)* variation points may either have interdependencies or be optional; *(iv)* frameworks may provide several ways of adding the same functionality; and *(v)* 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 so that it can manage all application instantiation issues systematically. A similar approach proposed by Fontoura et al. [2000] consists in creating a domain specific language for each variation point. In our approach, only one domain specific language, which acts as a facade over the framework being encapsulated, is implemented. Hence, by using a domain specific language 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 order to describe our approach and the domain specific language developed, the remainder of this paper is structured as follows. Section 2 presents background on domain specific languages, Section 3 describes researches that also focus on ways of easing the application instantiation process, and Section 4 gives an overview of previous research and technologies that played an important role during the development of GRENJ framework. Section 5 presents an evaluation performed to ascertain whether GRENJ framework domain is appropriate for being represented as a textual domain specific language, Section 6 highlights the major features of the domain specific language we have developed, and Section 7 contrasts the major differences between instantiating applications using GRENJ framework and applying our domain specific language. Section 8 concludes the paper with some remarks, limitations of our domain specific language, and future directions.

2. Domain Specific Languages

Domain specific languages (DSLs) are small languages that present limited expressiveness focused on a particular domain [van Deursen et al. 2000]. Usually, DSLs are not turing complete and have no imperative control structures, e.g., conditions and loops. Rather, most of them are declarative, 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 implies in writing a full-fledged parser in order to process it. Internal DSLs use existing general-purpose languages structures and, in most cases, the underlying execution environment, as a hosting base. An 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 ab-

straction 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

Several approaches have been proposed to support framework instantiation. Most of them 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 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. Some of the possible types of approach are based upon: *(i)* studying the framework source code and its documentation; *(ii)* exploration of 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. 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 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 other possible approach. Nevertheless, the exploration of exemplars has the following disadvantages: *(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 behavior. An example of this approach is given by Gangopadhyay and Mitra [1995].

Cookbooks are a sort of 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]. Several 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.

According to Brugali and Sycara [2000], 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] explore 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 [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 toward 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 is resulting from the reengineering of a framework implemented in Smalltalk which has been developed based upon a pattern language; GREN and GRN [Braga 2002], respectively. Therefore, both frameworks and GRN belong to the same domain, business resource management.

GRENJ has more than twenty-nine thousand lines of Java source code and its architecture consists of two layers: persistence and business. 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; not to mention knowledge of several advanced Java features, e.g., generics and reflection application programming interface (API). To overcome these difficulties, we intend to introduce a DSL that encapsulates all details concerning application instantiation. However, before doing this, in the next section we analyze whether the domain, as dealt and represented by GRN, deserves to be denoted as a textual DSL.

5. GRN Domain Evaluation

Sprinkle et al. [2009] present a series of questions intended as a checklist for determining 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 a *"resource"*. After identifying these concepts, for each potential resource, the user iterates throughout the pattern language coherently applying the patterns. Hence, we can conclude that the questions *(i)* through *(iii)* can be positively answered. Nevertheless, taking into consideration the 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 would not be of importance for creating the DSL syntax.

Given that most of the checklist items have been regarded as applicable to GRN and consequently GRENJ domain, we have developed a textual DSL in order to lessen the effort required to instantiate applications using GRENJ framework.

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. Therefore, 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. 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, a prefix character (i.e., #).

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, it is shown a instantiation 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.

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

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

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

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

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

```
...
instance = IdentifyResource.new "Movie"
instance += {:type => :string, :name => :synopsis,
             :access_modifier => :private,
             :generate_getters_and_setters => true}
...
```

As aforementioned, the information needed to instantiate applications is extracted 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.

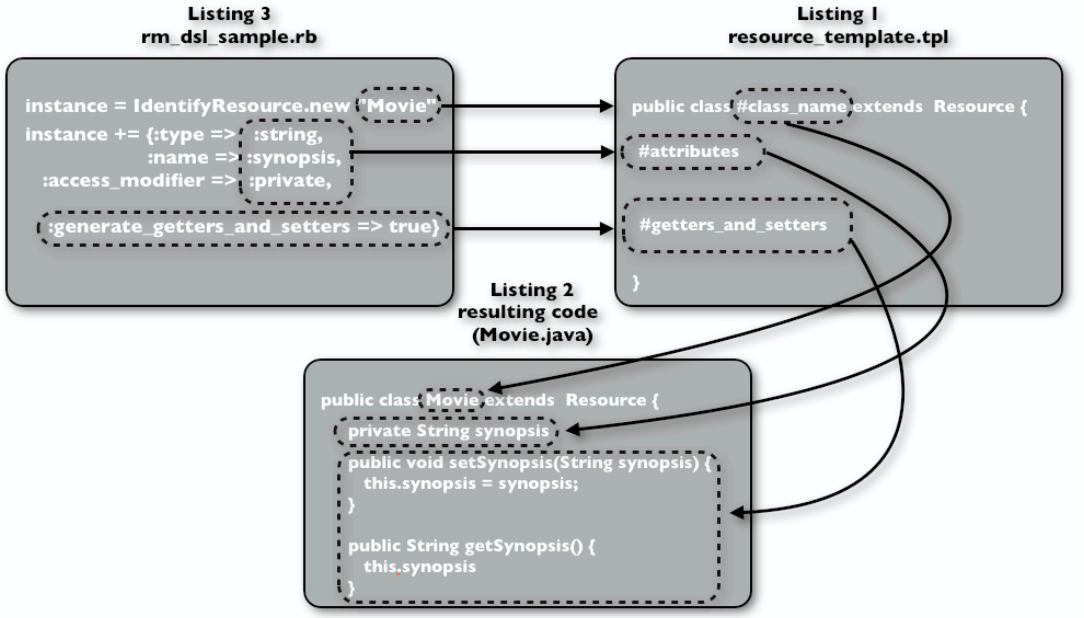

Figure 1. rm-DSL overview: the interaction among the involved files.

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.

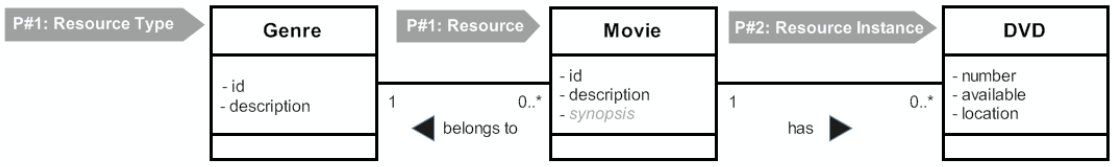

Figure 2. DVD rental store modeled applying GRN patterns.

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 an 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. Moreover, 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 new classes have to be implemented and customized at each hot spot.

8. Concluding Remarks

Learning to use an object-oriented framework effectively requires considerable investment of effort. In addition to it, 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 hiding 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.

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

```
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]
```

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 subclasses 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 limitation of our DSL is that it covers only four 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.

Another limitation of our DSL is that it does not implement complex validation functionalities. We aim at improving on our DSL for the purpose of checking whether a certain instantiation (i.e., combination of patterns) is valid according to GRN criteria, thereby providing the user with instantiation error messages.

References

Braga, R. T. V. (2002). *Um processo para construção e instanciação de frameworks baseados em uma linguagem de padrões para um domínio específico*. PhD thesis, ICMC/USP, São Carlos - SP.

Braga, R. T. V. and Masiero, P. C. (2002). The Role of Pattern Languages in the Instantiation of Object-Oriented Frameworks. In *Advances in Object-Oriented Information Systems*, pages 403–410. Springerlink.

Brugali, D. and Sycara, K. (2000). Frameworks and pattern languages: an intriguing relationship. *ACM Computing Surveys*, 32.

Durelli, V. H. S. (2008). *GRENJ: um framework obtido por um processo iterativo de reengenharia aplicando TDD*. master thesis, UFSCar/DC, São Carlos - SP.

Fayad, M. E., Johnson, R. E., and Schmidt, D. C. (1999). *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. John Wiley & Sons.

Flanagan, D. and Matsumoto, Y. (2008). *The Ruby Programming Language*. O'Reilly Media, Inc.

Fontoura, M., Braga, C., Moura, L., and Lucena, C. (2000). Using domain specific languages to instantiate object-oriented frameworks. *IEE Proceedings – Software*, 147(4):109–116.

Fowler, M. (2009). A Pedagogical Framework for Domain-Specific Languages. *IEEE Software*, 26(4):13–14.

Gangopadhyay, D. and Mitra, S. (1995). Understanding frameworks by exploration of exemplars. *International Workshop on Computer-Aided Software Engineering*.

Johnson, R. E. (1992). Documenting frameworks using patterns. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 63–76. ACM.

Johnson, R. E. (1997). Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42.

Menon, V. and Pingali, K. (1999). A case for source-level transformations in matlab. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*, pages 53–65. ACM.

Ortigosa, A., Campo, M., and Moriyón, R. (2000). 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*, pages 253–263. ACM.

Pree, W., Pomberger, G., Schappert, A., and Sommerlad, P. (1995). Active Guidance of Framework Development. *Software – Concepts and Tools*, 16(3).

Spinellis, D. (2001). Notable design patterns for domain-specific languages. *Journal of Systems Software*, 56(1):91–99.

Sprinkle, J., Mernik, M., Tolvanen, J.-P., and Spinellis, D. (2009). Guest Editors' Introduction: What Kinds of Nails Need a Domain-Specific Hammer? *IEEE Software*, 26(4):15–18.

van Deursen, A. and Klint, P. (1998). Little languages: little maintenance? *Journal of Software Maintenance: Research and Practice*, 10(2):75–92.

van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36.