When Frameworks Let You Down

Platform-Imposed Constraints on the Design and Evolution of Domain-Specific Languages

Danny M. Groenewegen Zef Hemel Lennart C. L. Kats Eelco Visser

Delft University of Technology, The Netherlands

{d.m.groenewegen,z.hemel,l.c.l.kats,e.visser}@tudelft.nl

1. Introduction

Application frameworks encapsulate knowledge of a particular domain in a reusable library. However, based on a general-purpose language, these do not provide notational constructs for the particular domain, and are limited to the static checks of the host language. Verification of correctness, security, and style constraints, and optimizations in terms of the application domain are not possible or very hard. It is common practice to build a Domain-Specific Language (DSL) as a thin abstraction layer over a framework, providing domain-specific notations and enabling analysis and reasoning at the level of these constructs (5; 6).

Using an established application framework as the platform for a DSL helps in the understanding the domain, and supports reuse of the domain knowledge gathered by the framework developers. By means of a *reference application*, a small application implemented using the framework, it is possible to identify the basic operations that are important for the domain, and map these to language constructs. Rather than designing a complete DSL "on paper," before its implementation, it is good practice to incrementally build higher-level abstractions on top of the basic operations through a process of inductive design (7). This enables quick turn-around time for the development of the DSL and the subsequent gradual extension as new applications are found, and new insights into the domain are acquired.

Application frameworks abstract over a lower-level platform. For instance, the Java Persistence API (JPA) abstracts over persistence operations in relational databases. As frameworks are designed for direct use by programmers, they tend to avoid complex interfaces, hiding underlying details, and limiting the number of concepts a programmer has to deal with. For code generation, these hidden details may become important as a DSL evolves and new features are added to it. A domain-specific language can "outgrow" its platform if it does not provide the elementary operations required to implement new aspects of the language. Despite the benefits of an inductive approach towards designing DSLs, and the merits of using established frameworks, this seems to be a recurring problem in the evolution of DSLs. In this paper, we give examples of occurrences of this problem and possible scenarios that describe how it can be dealt with. The examples we give are taken from our own experience with WebDSL, a domain-specific language for dynamic web applications with a rich data model (7). However, we believe these issues occur in other DSLs as well and therefore pose a more common problem.

2. Examples

The original design of WebDSL largely followed that of its target platform, the Seam web development framework. However, over time, new features were introduced that did not always match with this framework. In this section we give three examples of such extensions that were not directly or sufficiently supported by the framework.

Data Model Modularity WebDSL has first-class language constructs for data model definitions in the form of entity declarations. Data model definitions can be modularized into modules that each represent a different application concern. For example, consider a conference management system that has a generic user management system, defining usernames and passwords. In this system, all users have a list of papers they authored associated with them. Rather than directly modifying the data model for users, entangling the definition with a new concern, we designed a mechanism to augment already defined entities in other modules (see Figure 1). However, these kinds of partial entity declarations do not naturally translate to regular Java classes. They would instead require partial classes (such as in C#) or inter-type declarations (such as in AspectJ), but neither is supported by the current target platform.

Template Mechanism For reuse of page or page element definitions, WebDSL provides a template mechanism. Every page may import one or more templates, and these in turn may import other templates. For increased flexibility, we designed a system based on dynamically scoped template overrides. This template mechanism is inspired by the TEX typesetting system (4), which provides dynamically scoped macro definitions. Consider Figure 2, which demonstrates customizing the default behavior of templates with local template redefinitions. Unfortunately, Seam's templating mechanism (Facelets) is based on the idea of template inheritance, and is incompatible with the dynamic scoping mechanism. Again, this extension posed a mismatch between what is supported by the framework and what we would like to support in the DSL.

Access Control WebDSL provides specialized constructs for access control. While basic access control could be implemented

```
module usermanagement

  entity User {
    username :: String
    password :: Secret
  }

module paper

  extend entity User {
    authored → Set<Paper>
  }
```

Figure 1. Extension of data models in different modules

```
module pagelayout

  define main() {
     header()
     body()
     footer()
  }

  define body() {
     // Default body
  }

  define page home() {
    main()

    define body() {
      // New body, defined in the scope of page home()
    }
  }
```

Figure 2. Dynamically scoped templates.

```
module ac

  rule page viewUser(*) {
    viewingAllowed()
  }

module userpages

  define page viewUser(u : User) {
    // if viewingAllowed()
    output(u.name)
    output(u.authored)
  }

  define page userList() {
    for (u : User) {
      // if viewingAllowed()
      navigate(viewUser(u))
    }
  }
```

Figure 3. Modular access control rules.

through the use of if-statements, which hide certain (parts of) pages from the user if they do not have access to it. This approach, however, would be rather low-level, and would cause tangling of the concern with other aspects of the code.

The Seam framework offers an access control solution, which was deemed too inflexible for use in WebDSL. For instance, it assumes role-based access control policies, while WebDSL should support other policies as well. Seam's access control policy does not take access restrictions into account for the presentation of web pages, and it is difficult to extend with additional behavior. In WebDSL we wanted to support access control as a separate aspect that is woven into the application such that access control rules can have direct impact on page contents, e.g. by hiding navigation links to restricted pagers. Therefore, the language was extended with a specialized access control sub-language that allows the specification of access control as a separate set of rules (1). These rules put access control restrictions on pages, templates or other high-level components (see Figure 3), which serve as join points for the access control aspect language.

3. Scenarios

The examples described in the previous section indicate clear mismatches between frameworks and DSLs. In this section we describe a number of scenarios that can be explored to deal with these issues.

3.1 Introducing Intermediate Transformation Steps

Some platform mismatches can be dealt with by introducing model-to-model transformations, resolving certain shortcomings of the platform. For instance, dynamically scoped templates can be implemented through a transformation that statically expands templates. Template calls can be resolved statically and replaced by the template contents at generation time. A drawback of this solution is that it leads to a significant increase in the size of the generated code and cannot support recursive template calls. DSL features that require a type of aspect weaving can also be implemented through model transformations. Access control checks are woven into the pages they apply to, so that all that is left is DSL code that can be directly mapped to the underlying framework (1). Strictly adhering to the implementation style prescribed by the framework, these model-to-model transformations can cause abstraction inversion in the generated code. A feature such as templating is reimplemented on top of the existing mechanism, possibly incurring a performance penalty.

Likewise, if a target language does not suffice for a particular task, because it lacks certain features, it is possible to introduce these as extensions to the language. In particular, features for modularity such as partial classes and methods or multi-methods can be beneficial for code generation, and are not supported for all languages. By treating the target language as a first-class model, rather than plain text, it is possible to use the same techniques for model transformations as used in the rest of the DSL for realizing these extensions (2). In this manner the WebDSL generator internally generates partial Java classes, which in a later step of the generator are merged through a set of model transformations.

3.2 Adapting the platform

Another option is to adapt the application framework to better support the DSL. The JavaServer Faces library could be adapted to support WebDSL's dynamic scope templates for instance, and Seam's access control framework could be extended to suit our particular needs.

However, the developers of the DSL should be very well acquainted with the platform they are adapting. Often the developers are not motivated to adapt other people's work (the "not invented here" attitude).

Once changes have been made to the target platform, the original developers of that platform may evolve it over time (assuming the framework developers are not the same group as the DSL developers). In this case there are three options: either stick to the original adapted target platform, apply the changes made to the old version to the new one, or try to convince the framework developers to adopt the changes. Clearly, this is a maintenance issue.

3.3 Switching to a Different Target Platform

If a given platform really does not suffice for a particular task, it may also be an option to replace it with an alternative. This can be another, similar framework, or may be a lower-level alternative, for instance by generating plain Java Servlets code.

The approach of compilation by normalization (3) can be used to guide the generation to a low-level target. It iteratively abstracts from a low-level target platform, while not restricting access to the underlying primitives.

One issue that may come up is having introduced leaky abstractions. The DSL was initially built as a thin layer on top of a framework, so is likely to be fairly specific to that platform. It may be very difficult or even impossible to replace the platform with another one, because it would imply changes to the DSL.

When it is decided to generate lower-level code it is likely that much more code has to be generated, mimicking the behavior of the framework that was previously used, because the semantics of the DSL should not change. Thus, developing the generator is likely to require a larger effort than when targeting a high-level framework. Frameworks often encapsulate years of experience in their particular domain. Replacing them with a home brewed framework should not be underestimated, neither in terms of maintenance or performance optimizations. On the other hand, frameworks are usually optimized to make writing code using them as simple as possible for the developer, often at the expense of performance. For instance, many Java frameworks depend on reflection to reduce the size of client code. However, this incurs a runtime overhead, which can be avoided when code is generated.

3.4 Cutting Your Losses

It may turn out that the previously described scenarios are simply too expensive. The time and effort required to switch to a different platform, adapt it, or to add additional model transformations can be too costly and the gain too small. Settling for Facelet templates with inheritance, and the somewhat inflexible access control solution of Seam could be an acceptable solution if this is the case. In this scenario, the DSL and the framework are kept in sync. No features are added to the DSL unless fully supported by the framework.

This raises the question of what expectations one has of DSLs. Are they intended as only a thin layer of syntax and set of checks on top of a framework, or is that just a point of departure that leads to an independently evolving language? Our experience from designing and implementing WebDSL has taught us that frameworks form a very useful foundation to build upon, but that at some point the DSL may simply outgrow its underlying platform. In this paper we outlined four scenarios to deal with this problem, each having their respective advantages and drawbacks. Future research is required to reveal best practices in this area.

Acknowledgments This research was supported by NWO/JACQUARD projects 638.001.610, *MoDSE: Model-Driven Software Evolution*, 612.063.512, and *TFA: Transformations for Abstractions*.

References

[1] D. Groenewegen and E. Visser. Declarative access control for WebDSL: Combining language integration and separation of concerns. In D. Schwabe and F. Curbera, editors, *International Conference on Web Engineering (ICWE 2008)*. IEEE CS Press, July 2008.

[2] Z. Hemel, L. C. L. Kats, and E. Visser. Code generation by model transformation. A case study in transformation modularity. In J. Gray, A. Pierantonio, and A. Vallecillo, editors, *International Conference on Model Transformation (ICMT 2008)*, Lecture Notes in Computer Science. Springer, June 2008.

[3] L. C. L. Kats, M. Bravenboer, and E. Visser. Mixing source and bytecode. A case for compilation by normalization. In G. Kiczales, editor, *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2008)*, Nashville, Tenessee, USA, October 2008. ACM Press.

[4] D. E. Knuth. *The TEXbook*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[5] T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[6] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[7] E. Visser. WebDSL: A case study in domain-specific language engineering. In R. Lammel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, Lecture Notes in Computer Science. Springer, 2008. Tutorial for International Summer School GTTSE 2007; to appear.