Chapter 17
Design Patterns and Design Principles for Internal Domain-Specific Languages

Sebastian Günther
Vrije Universiteit Brussel, Belgium

ABSTRACT

Internal DSLs are a special kind of DSLs that use an existing programming language as their host. To build them successfully, knowledge regarding how to modify the host language is essential. In this chapter, the author contributes six DSL design principles and 21 DSL design patterns. DSL Design principles provide guidelines that identify specific design goals to shape the syntax and semantic of a DSL. DSL design patterns express proven knowledge about recurring DSL design challenges, their solution, and their connection to each other – forming a rich vocabulary that developers can use to explain a DSL design and share their knowledge. The chapter presents design patterns grouped into foundation patterns (which provide the skeleton of the DSL consisting of objects and methods), notation patterns (which address syntactic variations of host language expressions), and abstraction patterns (which provide the domain-specific abstractions as extensions or even modifications of the host language semantics).

INTRODUCTION

Domain-Specific Languages (DSLs) are languages specifically tailored to express the concepts and notations of a domain by embodying suitable abstractions and notations (van Deursen et al., 2000). Using DSLs increases the productivity, reduces errors, and allows to better focus on the problem space (Czarnecki & Eisenecker, 2000; Greenfield et al., 2004). DSLs are used for a wide spectrum of domains, for example in telephone services (Latry et al., 2007), for healthcare systems (Munelly & Clarke, 2007), and for magnet tests at the Large Hadron Collider at CERN (Arpaia et al., 2009). This chapter focuses on one particular type of DSLs. Internal DSLs are based on an existing host language, they are built by carefully combining syntactic options of the host with well-scoped semantic modifications using the host's support for metaprogramming. In the remainder of this chapter, we mean internal DSLs whenever we talk of DSLs.

DOI: 10.4018/978-1-4666-6042-7.ch017

Considering existing open-source and research DSLs, which can be especially found in languages such as Ruby, Python, and Scala, we can study the applied syntactic and semantic modifications of the host. Although there are plenty examples of DSLs, two problems are apparent. First, it remains difficult for the individual developer to explain his particular design because the relation of used host language constructs and DSL design questions is not clear. Second, the host language constructs are language specific, so it is difficult to use designs from one host language in another one. Therefore, we think that only a common language to describe the syntactic and semantic modifications, as well as to understand how a modification affects the characteristic of a DSL, allows developers to communicate and plan DSL design. To achieve this goal, developers need to understand design principles and design patterns of DSLs.

Design principles are guidelines that provide design goals to shape the syntax and semantic of a DSL. The choice of principles influences how a DSL is developed, which results in DSLs that are distinguishable from each other and from host language code. In related work, two limitations become apparently. First, most case studies, including recent ones, do not consider them (Groote et al., 1995; Thibault et al., 1997; Barreto et al., 2002; Agosta & Pelosi, 2007; Dinkelaker & Mezini, 2008; Bennett et al., 2006; Havelund et al., 2010). And second because those DSL case studies that treat principles only define them, but do not show how to actually achieve them (Atkins et al., 1999; Oliveira et al., 2009; Bentley, 1986).

Design patterns are the essential way to provide a vocabulary for communicating and planning DSL designs. Although the implementation of a DSL is host language specific, a careful analysis of DSLs shows that there are language-independent techniques to form the syntax and semantics of a DSL. We refine these techniques to patterns, which identify a common DSL design challenge and a common solution that can be implemented by using host language specific mechanisms. Related work about DSL design patterns shows two limitations. One the one hand, case studies about internal DSL offer diverse techniques for design and implementation, ranging from modifications of a host language's metaobject protocol to object-oriented mechanisms (Agosta & Pelosi, 2007; Cannon & Wohlstadter, 2007; Cunningham, 2008; Dinkelaker & Mezini, 2008; Sloane, 2008). However, the explained techniques are not generalized outside the context of their DSL, which makes them hard to reuse for other case studies. And on the other hand, most existing work about DSL design patterns suggests very general patterns (Haase, 2007; Mernik et al., 2005; Spinellis, 2001; Zdun & Strembeck, 2009), which are not detailed enough for finer aspects of syntax and semantic design.

To overcome the mentioned limitations, we contribute with a set of 6 DSL design principles and an explanation of 21 DSL design patterns. We identify three design principles related to the syntax of a DSL (*notation*, *compression*, and *absorption*) and three principles related to the semantics (*abstraction*, *generalization*, and *optimization*). The patterns are grouped into *foundation patterns*, which provide the skeleton of the DSL consisting of objects and methods, *notation patterns*, which address syntactic variations by using host language expressions, and *abstraction patterns*, which provide the domain-specific abstractions as extensions or even modifications of the host language semantics. We show how to apply the principles by identifying individual design goals of the patterns – each goal giving the pattern a unique way of being applied. The patterns have been found and implemented in the context of languages with an object-oriented core and strong support for reflection, and therefore are especially applicable in similar programming languages. Furthermore, in the concluding section, we explain how programming languages with another core paradigm, for example functional languages, can benefit from these patterns.