ELSEVIER

The Journal of Systems and Software 56 (2001) 91–99

The Journal of Systems and Software

www.elsevier.com/locate/jss

Notable design patterns for domain-specific languages

Diomidis Spinellis

Department of Information and Communication Systems, University of the Aegean, GR-83 200 Karlovasi, Greece

Received 18 August 1999; received in revised form 17 December 1999; accepted 14 February 2000

Abstract

The realisation of domain-specific languages (DSLs) differs in fundamental ways from that of traditional programming languages. We describe eight recurring patterns that we have identified as being used for DSL design and implementation. Existing languages can be extended, restricted, partially used, or become hosts for DSLs. Simple DSLs can be implemented by lexical processing. In addition, DSLs can be used to create front-ends to existing systems or to express complicated data structures. Finally, DSLs can be combined using process pipelines. The patterns described form a pattern language that can be used as a building block for a systematic view of the software development process involving DSLs.

Keywords: Design patterns; Domain-specific languages

1. Introduction

The realisation of domain-specific languages (DSLs) differs in fundamental ways from that of traditional programming languages. Although the idea of DSLs is more than mature (Landin, 1966), their role in the architecture, design, and implementation of software systems has only recently been acknowledged (Ramming, 1997). Some DSLs are being designed as full-flavoured programming languages (Wirth, 1974) and implemented as interpreters or compilers using traditional programming language implementation techniques and tools (Aho et al., 1985). However, the software process and economics behind the realisation of a DSL are, more often than not, entirely different from those that drive the implementation of a traditional programming language. Specifically, DSLs are by definition part of a larger system and often implemented for a narrow usage domain. The resources available for designing and implementing them are therefore constrained to a small percentage of those available for the system they belong to, and difficult to amortise over a large user base. The constraints on the design and implementation effort and talent that can be devoted to the realisation of a DSL have brought forward a number of distinct and reusable strategies. These DSL realisation strategies solve specific problems of design and can be applied to many similar problems. The description of such reusable designs, often referred to as *patterns* (Alexander et al., 1977; Coplien and Schmidt, 1995; Gamma et al., 1995), allows their dissemination and conscious reuse by DSL designers and software practitioners.

The remainder of this paper is structured as follows: in Section 2, we introduce DSLs and outline their differences from executable specification and general purpose languages while in Section 3, we present the formalism of *design patterns* that we use for describing the DSL realisation strategies in Section 4. Finally, Section 5 concludes this paper with a discussion of the relationships between the outlined design patterns and directions of future research.

2. Domain-specific languages

A DSL is a programming language tailored specifically to an application domain: rather than being for a general purpose, it captures precisely the domain's semantics. A DSL-based development methodology addresses the need for increasing domain specialisation in the software engineering field (Jackson, 1999). Examples of DSLs include *lex* and *yacc* (Johnson and Lesk, 1987) used for program lexical analysis and parsing, HTML (Berners-Lee and Connolly, 1995) used for document mark-up, and VHDL used for electronic hardware descriptions. DSLs allow the concise description of an application's logic reducing the semantic distance between the problem and the program (Bell et al., 1994; Spinellis and Guruprasad, 1997).

E-mail address: dspin@aegean.gr (D. Spinellis).

0164-1212/01/$ - see front matter
PII: S0164-1212(00)00089-3

DSLs are, by definition, special purpose languages. Any system architecture encompassing one or more DSLs is typically structured as a confederation of modules; some implemented in one of the DSLs and the rest implemented using a general purpose programming language (Fig. 1). As a design choice for implementing software systems, DSLs present a number of distinct advantages over a "hard-coded" program logic:

Concrete expression of domain knowledge. Domain-specific functionality is not coded into the system or stored in an arcane file format; it is captured in a concrete human-readable form. Programs expressed in the DSL can be scrutinised, split, combined, shared, published, put under release control, printed, commented, and even be automatically generated by other applications.

Direct involvement of the domain expert. The DSL expression style can often be designed so as to match the format typically used by the domain expert. This results in keeping the experts in a very tight software lifecycle loop where they can directly specify, implement, verify, and validate, without the need of coding intermediaries. Even if the DSL is not high-level enough to be used as a specification language by the domain expert, it may still be possible to involve the expert in code walkthroughs far more productive than those over code expressed in a general purpose language.

Although the DSL concept bears similarity to executable specification languages (Sommerville, 1989, p. 125; Turski and Maibaum, 1987, p. 135) such as OOSPEC (Paryavi and Hankley, 1995), the DSL approach exhibits some important advantages:

Expressiveness. Executable specification languages taking a Swiss army knife approach towards the problem of specification offer facilities for specifying all types of systems, but often at a cost of clearness of expression. As an example, OBSERV (Tyszberowicz and Yehudai, 1992) provides a multiparadigm environment allowing the system specification using object-oriented constructs, finite state machines, and logic programming. In contrast, DSLs being tailored towards a narrow, specific domain can be designed to provide the exact formalisms suitable for that domain.

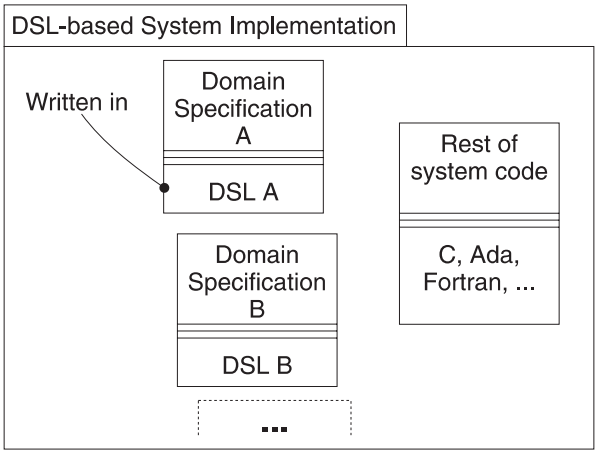

Fig. 1. UML diagram of a DSL-based system architecture.

Runtime efficiency. The possible interactions between different elements of a general purpose specification language such as its type system and its support for concurrency result in runtime inefficiencies. A narrowly focused DSL can employ the most efficient implementation strategy and specialised optimisations for satisfying the expressed specification.

Modest implementation cost. DSLs are typically implemented by a translator that transforms the DSL source code into source or intermediate code compatible with the rest of the system. Such an approach can often be implemented using string processing languages such as *awk* (Aho et al., 1979) and Perl, language development tools such as *lex* and *yacc*, specialised systems such as TXL (Cordy et al., 1991) and KHEPERA (Faith et al., 1997), or declarative languages such as Prolog and ML. The DSL implementation cost is – and should always be – modest.

Reliability. As described in the previous paragraph, the limited scope of a DSL often allows a source-to-source transformation type of implementation. The small scale of the required implementation effort often results in a translator whose correctness can be trivially verified. The size of typical executable specification languages means that the implementor must often take the correctness of the language's implementation on trust.

On the other hand, the system architect contemplating the use of a DSL architecture should also have in mind the following potential shortcomings of this approach:

Tool support limitations. CASE and integrated software development tools offer only limited support for integrating DSLs into the development process. Ad hoc solutions are often required to smoothly integrate DSL code with existing revision control systems, compilers, editors, source browsers, and debuggers.

Training costs. In contrast to established specification languages such as Z (Potter et al., 1991) system implementers and maintainers will by definition have no prior exposure to the DSL being used. This problem is somehow mitigated by the fact that an appropriately chosen DSL will be familiar to other participants of the implementation effort such as those involved in the specification, beta testing, and final use. These participants will be able to perform the DSL code walkthroughs – a task normally reserved for experienced software engineers.

Design experience. DSL-based system architectures are not widely adopted within the software industry. As a result, there is an evident lack of design experience, prescriptive guidelines, mentors, design patterns, and supporting scientific literature. Early adopters will need

to rely more on their own judgement as they adopt the approach in a stepwise fashion.

Software process integration. The use of DSLs is not yet an integral part of established software processes. Therefore, the software process being used has to be modified in order to take into account the design, implementation, integration, debugging, and maintenance of the adopted DSLs.

The implementation of a DSL differs from the implementation of a general purpose language. Compilers for general purpose languages are typically structured as a lexical analyser, a parser, a semantic analyser, an optimiser, and a target code generator. In contrast, the limited scope of a DSL allows and requires different implementation strategies. The lexical, syntactic, and semantic simplicity of DSLs often obviate the need for some elements that would be required by a general purpose language compiler; for example, instead of using a parser front-end, DSL implementations often process the source language using regular expressions. In addition, the often-limited user population of a DSL does not justify a large implementation effort forcing DSL implementers to choose the most economical realisation strategies; as an example, compilation into assembly code of the target machine is rarely a practical proposition. Finally, as DSLs are often part of the development process of a larger system, schedule pressures drive DSL builders towards implementation methods that can rapidly deliver results. The aim of this paper is to provide, in the form of a pattern language, a repertoire of methods often used in the implementation of a DSL.

3. Design patterns

The notion of design patterns has its origins on the seminal work of the architect Christopher Alexander. Alexander outlines how the relationship between recurring problems and their respective solutions establishes patterns as follows:

> "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." – (Alexander et al., 1977)

Twenty years later Gamma et al. (1995) cross-pollinated these ideas into the field of reusable object-oriented software design. Design patterns offer a convenient way to capture, document, organise, and disseminate existing knowledge from a given area in a consistent and accessible format. Patterns differ from algorithms and data structures in that the concepts they describe cannot be coded and used as a subroutine or an object class. Patterns also differ from frameworks as they do not describe the structure of a complete system: interrelated patterns are typically used together to solve a general design problem in a given context.

In this paper, we describe eight recurring patterns that we have identified as being used for DSL design and implementation. The description of these patterns provides the DSL designers with a clear view of the available DSL realisation strategies, the forces that will guide them towards the selection of a specific pattern, the consequences of that decision, examples of similar uses, and the available implementation alternatives. In our description of the patterns, we followed – in free text form – the format and classification used by Gamma et al. (1995). We classify each pattern as *creational* if it involves the creation of a DSL, *structural* if it describes the structure of a system involving a DSL, and *behavioural* if it describes DSL interactions.

4. DSL design patterns

In the following sections, for every pattern we:

- provide the name that will be used to describe it;
- illustrate its structure using a simple UML (Rumbaugh et al., 1999) diagram;
- classify it as creational, behavioural, or structural;
- illustrate the design problem that provides our motivation to use the pattern;
- outline the situations where that pattern can be applied;
- outline the pattern's participants;
- describe how the pattern supports its objectives;
- provide examples and prescriptive guidelines towards the pattern's implementation.

4.1. Piggyback

The piggyback structural pattern (Fig. 2) uses the capabilities of an existing language as a hosting base for a new DSL. Often a DSL needs standardised support for common linguistic elements such as expression handling, variables, subroutines, or compilation. By designing the DSL on top of an existing language, the needed linguistic

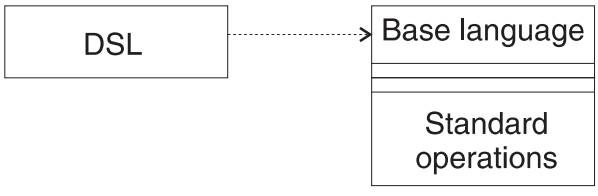

Fig. 2. The piggyback pattern.

support is provided "for free". The piggyback pattern can be used whenever the DSL shares common elements with an existing language. Typically, the DSL language processor passes the linguistic elements that are expressed in the existing language to the language processor of the existing language. Where the DSL is implemented as a compiled language, a typical implementation compiles the DSL code into the base language: DSL code is compiled as needed, while embedded base-language elements are emitted unmodified. Consequently, the resulting output of the compilation consists entirely of the base language. If the DSL is implemented as an interpreter, a similar strategy can be applied if the base language provides a facility for calling its interpreter with suitable arguments from within the DSL interpreter.

Typical examples of this approach are the *yacc* (Johnson, 1975) and *lex* (Lesk, 1975) processors. While the specifications of the input grammar (in the case of *yacc*) and the input strings (in the case of *lex*) are expressed in a DSL, the resulting actions for recognised grammar rules and tokens are specified in C which is also the processors' output language. *Yacc* uses the piggyback approach more aggressively as it introduces special variables (denoted by the $ sign) to the C constructs used for specifying the actions.

The piggyback approach resembles in structure the compiler front-ends that generate an intermediate language. However, the structure we propose uses an existing, human-readable, and typically general-purpose language, as the compilation target rather than a specialised, machine-readable intermediate language. The effort of translating the DSL into an existing human-readable language instead of implementing an alternative compiler front-end is substantially lower. In addition, the process of this translation is relatively straightforward and can be implemented as a simple source-to-source transformation – often merely using lexical processing constructs as described in Section 4.3. In contrast, the implementation of a compiler front-end requires detailed knowledge of the intermediate language, and often intimate knowledge of a specific compiler implementation.

4.2. Pipeline

The *pipeline* behavioural pattern (Fig. 3) solves a problem of DSL composition. Often a system can best be described using a family of DSLs. The prototypical example for such an application is the composition of diverse mark-up languages in text processing systems.

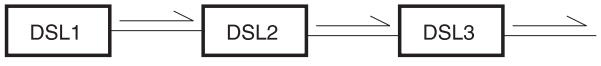

Fig. 3. The pipeline pattern.

Such different languages can be used to specify tables (Lesk, 1979b), mathematical equations (Kernighan and Cherry, 1974), chemical formulas (Bentley et al., 1987), pictures (Kernighan, 1982), graphs (Bentley and Kernighan, 1986), and organic element chemical structures (Bentley et al., 1987). In cases where a number of DSLs are needed to express the intended operations, their composition can be designed and implemented using a pipeline. Typically, all DSLs are organised as a series of communicating elements. Each DSL handles its own language elements and passes the rest down to the others. Sometimes, the output of one DSL can be expressed in terms of the input expected by another DSL further down the pipeline chain (Bentley, 1986). The use of the pipeline pattern encourages the division of responsibility among small specialised DSLs and discourages bloated feature-rich language designs. The DSL-based system can be built in a stepwise fashion, adding components as needed with new components utilising existing ones.

As suggested by its name, the pattern can often be implemented using a pipeline of independent communicating system processes. Many modern operating systems provide facilities for setting up such a pipeline, while the Unix shells also provide a supporting built-in notation. The pipeline approach has been used by the *troff* (Ossanna, 1979) family of text processing tools. Elements of a *troff*-based text processing pipeline can include *eqn* (Kernighan and Cherry, 1974) for processing equations, *tbl* (Lesk, 1979b) for processing tables, *pic* (Kernighan, 1982) for processing pictures, *grap* (Bentley and Kernighan, 1986) for drawing statistical displays, *dag* (Gansner et al., 1988) for typesetting directed graphs, *chem* (Bentley et al., 1987) for typesetting chemical structures, and *refer* (Lesk, 1979a) for processing references. A similar structure has also been used to produce algorithm animations (Bentley and Kernighan, 1991). In addition, if one considers the command line arguments passed to typical Unix commands as a mini-DSL, then typical pipelines of Unix tool invocations can also be considered as an application of this pattern. This mode of use allows the implementation of sophisticated applications such as spell checkers or complicated operations on images and sound using families of tools such as the system's text processing tools, the *pbm* (Poskanzer et al., 1993) portable bitmap collection, and the *sox* sound tools.

4.3. Lexical processing

The *lexical processing* creational pattern (Fig. 4) offers an efficient way to design and implement DSLs. Due to their – by definition – limited field of applicability, DSLs impose severe restrictions to the effort that can be used for their design and implementation. Many DSLs can be designed in a form suitable for processing by techniques of simple lexical substitution; without tree-based syntax

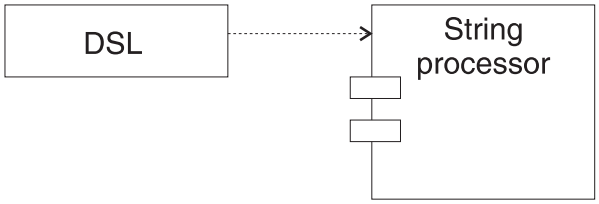

Fig. 4. The lexical processing pattern.

analysis. The design of the DSL is geared towards lexical translation by utilising a notation based on lexical hints such as the specification of language elements (e.g., variables) using special prefix or suffix characters. The form of input for this family of DSLs is often line-oriented, rather than free form and delimited by character tokens. This design pattern can be used together with the *piggyback* pattern in cases where, after some lexical processing, the output of the DSL processor can be passed to the processor of the *base* language.

The utilisation of this pattern lowers the implementation cost for DSLs making them a practical proposition for applications where the cost of a full parser-based translation would not be justified. As translators based on lexical structure are often implemented using interpreted or rapid prototyping languages, the DSL design and implementation can gracefully evolve together in a combined iterative process. Examples of this application include numerous DSLs implemented using tools such as *sed* (McMahon, 1979), *awk* (Aho et al., 1988), *Perl* (Wall and Schwartz, 1990), *Python* (Lutz, 1996), *m4* (Kernighan and Ritchie, 1979), and the C pre-processor. Most of these tools offer a rich set of lexical processing and substitution facilities – often expressed in terms of extended regular expressions – that can be used to implement a complete DSL in tens of lines of code.

4.4. Language extension

The *language extension* creational pattern (Fig. 5) is used to add new features to an existing language. Often an existing language can effectively serve a new need with the addition of a few new features to its core functionality. In this case, a DSL can be designed and implemented as an extension of the base language. The language extension pattern differs from the piggyback pattern by the roles played by the two languages: the piggyback pattern uses an existing language as an implementation vehicle for a DSL, whereas the extension pattern is used when an existing language is extended within its syntactic and semantic framework to form a DSL. The design of a DSL using this pattern involves the addition of new language elements to an existing base language. These elements can include new data types, language block interaction mechanisms, semantic elements, or syntactic sugar. Typically, the DSL inherits all syntax and semantics of the base language used, while adding its own extensions. An object-oriented class hierarchy can thus be formed with a number of DSLs being derived from base languages and forming base languages for other DSLs.

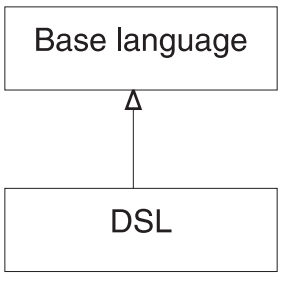

Fig. 5. The language extension pattern.

The use of the language extension pattern frees the DSL designer from the burden of designing a full-featured language. In addition, where the pattern is used to design a non-trivial DSL hierarchy, the pattern offers a clear way of organising the language relationships and interactions (Spinellis et al., 1995). Compiled-language implementations of the extension pattern are often structured in the form of a pre-processor which transforms the DSL into the base language. Alternatively, source-to-source transformations (Cordy et al., 1991), code composition (Stichnoth and Gross, 1997), or intentional programming (Simonyi, 1995) techniques can be used to augment the language using high level operators. One of the earliest examples of this pattern is the "rational FORTRAN" (*Ratfor*) compiler (Kernighan, 1975) which provided a structured version of FORTRAN. The implementation of the original C++ compiler (*cfront*) also used this technique (Stroustrup, 1984). A current effort using the extension pattern involves the addition of generic types to the Java programming language (Bracha et al., 1998). Extensions of interpreted languages can also benefit from this design pattern by implementing the language extension using a meta-interpreter (Sterling and Shapiro, 1986, pp. 303–330) or a meta-circular evaluator (Abelson et al., 1990, pp. 293–382). Examples include the examination of abstract syntax trees using an interpreter of Prolog extended with an ambient current object (Crew, 1997) and the extension of ML for graph drawing (Kamin and Hyatt, 1997).

4.5. Language specialisation

Language specialisation (Fig. 6) is a creational pattern that removes features of a base language to form a DSL. In some cases, the full power of an existing language may prevent its adoption for a specialised purpose. A representative case arises when requirements related to the safety or security aspects of a system can be satisfied only by removing some "unsafe" aspects (such as

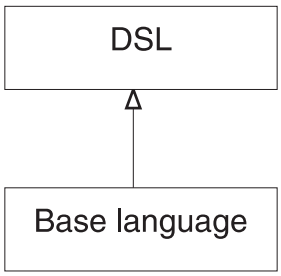

Fig. 6. The language specialisation pattern.

dynamic memory allocation, unbounded pointers, or threads) from a language (Motor Industry Research Association, 1994). In such cases, a DSL may be designed and implemented as a subset of an existing language. Whenever some specific features of an existing language render it unsuitable for a given application, the design of a DSL following the specialisation pattern can result in a mature language that satisfies the given requirements. The design of the DSL involves the removal from the base language of the unwanted syntactic or semantic features. Since the DSL is effectively a subset of the base language, the removal can be guaranteed by a language processor that checks the DSL conformance. In a limited number of cases, additional run-time checks may be required. Examples of DSLs designed following the specialisation pattern are Java*light* (Nipkow and von Oheimb, 1998) which is a type-safe subset of Java, the educational subsets of Pascal used for a stepwise introduction to the language (Savitch, 1995), the HTML (Berners-Lee and Connolly, 1995) application of SGML (ISO8879, 1986), and the automotive "safer-subset" of C (Edwards and Rivett, 1997).

4.6. Source-to-source transformation

The *source-to-source transformation* creational pattern (see Fig. 7) allows the efficient implementation of DSL translators. As outlined in Section 2, the resources available for implementing a DSL are often severely constrained. Source-to-source transformation can be used to ease the burden of implementation. When the DSL cannot be designed as a language extension, specialisation, or using the piggyback pattern, it is often possible to leverage the facilities provided by existing language tools using a source-to-source transformation technique. The DSL source code is transformed via a suitable shallow or deep translation process into the source code of an existing language. The tools available for the existing language are then used to host – compile or interpret – the code generated by the transformation process.

When using this pattern, one capitalises on the existing language processor infrastructure. This can include optimising compilers, linkers, and native code instruction schedulers. In addition, in some circumstances, even tools that rely on mappings between the source code and the machine code (such as profilers, execution tracers, and symbolic debuggers) can be used. In particular, some candidate host languages such as C offer a mechanism for specifying the file and source line of the DSL code that generated a particular sequence of host code instructions. The use of this pattern makes it also relatively easy to troubleshoot the DSL compilation process, because the resulting code will often be easy to read and reason about. Another possibility involves the translation of the DSL code into the intermediate language used by existing language compilers. The pattern can be implemented using a traditional lexical analysis, parsing, and host code generation process. In addition, a number of tools such as TXL (Cordy et al., 1991) and KHEPERA (Faith et al., 1997) can be used to speed-up the implementation process.

4.7. Data structure representation

The *data structure representation* creational pattern (Fig. 8) allows the declarative and domain-specific specification of complex data. Data-driven code (Kernighan and Plauger, 1978, p. 67) relies on initialised data structures whose complexity can often make them difficult to write and maintain. Complicated structures are better expressed using a language rather than their underlying representation (e.g., a graph adjacency list may be easily expressed as a list of path connections). Designing a DSL to represent the data offers an attractive solution to the problem. The pattern is of use whenever a non-trivial data structure (anything other than rectangular arrays) needs to be initialised with data. It is particularly applicable to the initialisation of data structures whose elements are interrelated such as trees, graphs, arrays of pointers to statically initialised structure elements, arrays of pointers to functions, and multilingual text elements.

The DSL typically defines a user-friendly, alternative but isomorphic, representation of the underlying data structure elements. The DSL compiler can then parse the alternative data representation and transform the data elements to the structure needed for the internal

Fig. 7. The source-to-source transformation pattern.

Fig. 8. The data structure representation pattern.

representation. The adoption of this pattern minimises the chances of initialising data structures with wrong or inconsistent data, as the DSL compiler can perform such checks when compiling the data into the internal format. In addition, the data can be generated in the most efficient internal representation using tools such as the perfect hash function generator *gperf* (Schmidt, 1990). The pattern is most often implemented as a DSL compiler from the external to the internal representation. Where runtime efficiency is not a major constraint, the DSL can be directly coded within the system's hosting language source code utilising the user-friendly alternative data structure and suitably interpreted at runtime. Such strategies are often employed in systems written in interpreted declarative languages such as Prolog or Lisp. A representative example of a DSL based on this pattern is FIDO (Klarlund and Schwarzbach, 1997) which is designed to concisely express regular sets of strings or trees. Other cases of DSL-based data specifications are the table initialisations generated by *yacc* (Johnson, 1975) and *lex* (Lesk, 1975) for the table-driven parsing and lexical analysis automata they create.

4.8. System front-end

The configuration and adaptation of a system can often be relegated to a DSL front-end (Fig. 9). Complicated software systems offer hundreds of configuration options, while their users require ever-increasing adaptation possibilities. Adding more features and configuration options can enlarge and complicate the system with diminishing returns on real functionality and user-friendliness. Making the system programmable by means of the DSL front-end structural pattern provides its users with a declarative, maintainable, organised, and open-ended mechanism for configuring and adapting it. Systems with more than a few configuration options, and systems whose operation cannot be adequately specified by means of some arguments or a graphical user interface typically benefit from the addition of a DSL front-end.

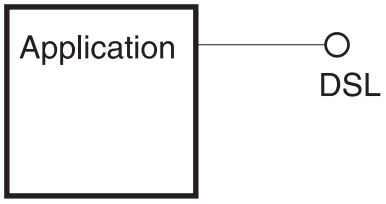

Fig. 9. The system front-end pattern.

Using this strategy, the system's configuration parameters and internal functionality are exposed as elements of the DSL – e.g., as variables and functions, respectively. At this point, it is often advantageous to remove from the system all elements that can be specified by means of the DSL, and code them in terms of it, thus simplifying its structure.

Often the addition of a DSL to a system can reveal synergistic effects by enabling its communication with other systems, allowing for the automatic generation of DSL programs with increased functionality, establishing a common language among its user base, providing a mechanism for optimising or checking the system's configuration, and opening a market for third-party add-on applications. The pattern is most often implemented as an interpreted language embedded within the target system. A number of existing interpretative languages have been used or targeted explicitly for this purpose. Lisp-like languages have been used by systems such as the Emacs (Stallman, 1984) editor and the AutoCAD package (Rawls and Hagen, 1998), while languages such as Tcl (Ousterhout, 1994), Perl (Wall and Schwartz, 1990), and Microsoft's Application Basic

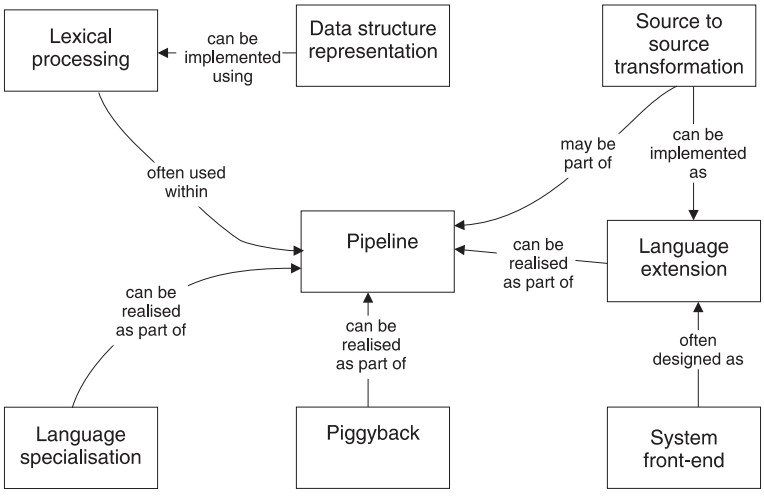

Fig. 10. Relationships within the DSL pattern language.

(Boctor, 1999) provide explicit support for system embedding.

5. Conclusions

We have described a DSL pattern language consisting of eight DSL design patterns. Our pattern language does not include the design of a DSL using traditional programming language design and implementation techniques (lexical analysis, parsing, code generation), as the aspects of those are extensively covered in the existing literature. The relationships of the patterns we described are depicted in Fig. 10. The interrelationships between the patterns are both interesting, and typical of a pattern language focused on a specific domain. Throughout our literature research for drafting this work we were impressed by the multitude of DSL designs, implementation strategies, and resulting systems, and the scarcity of supporting design frameworks and methodologies. We hope that the pattern language we have presented can be used as a building block for a systematic view of the software development process involving DSLs.

Acknowledgements

We would like to thank the anonymous referees for their insightful comments on the previous version of this paper.

References

Abelson, H., Sussman, G.J., Sussman, J., 1990. Structure and Interpretation of Computer Programs. MIT Press, Cambridge, MA.

Aho, A.V., Kernighan, B.W., Weinberger, P.J., 1979. Awk – a pattern scanning and processing language. Software-Pract. Exper. 9 (4), 267–280.

Aho, A.V., Sethi, R., Ullman, J.D., 1985. Compilers, Principles, Techniques, and Tools. Addison-Wesley, Reading, MA.

Aho, A.V., Kernighan, B.W., Weinberger, P.J., 1988. The AWK Programming Language. Addison-Wesley, Reading, MA.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., Angel, S., 1977. A Pattern Language. Oxford University Press, Oxford.

Bell, J., Bellegarde, F., Hook, J., Kieburtz, R.B., Kotov, A., Lewis, J., McKinney, L., Oliva, D.P., Sheard, T., Tong, L., Walton, L., Zhou, T., 1994. Software design for reliability and reuse: a proof-of-concept demonstration. In: Proceedings of the Conference on TRI-Ada '94, ACM, ACM Press, New York, pp. 396–404.

Bentley, J.L., 1986. Programming pearls: little languages. Commun. ACM 29 (8), 711–721.

Bentley, J.L., Kernighan, B.W., 1986. GRAP – a language for typesetting graphs. Commun. ACM 29 (8), 782–792.

Bentley, J.L., Kernighan, B.W., 1991. A system for algorithm animation. Comput. Syst. 4 (1), 5–30.

Bentley, J.L., Jelinski, L.W., Kernighan, B.W., 1987. CHEM – a program for phototypesetting chemical structure diagrams. Comput. Chem. 11 (4), 281–297.

Berners-Lee, T., Connolly, D., 1995. RFC 1866: Hypertext Markup Language – 2.0 (November). Status: PROPOSED STANDARD.

Boctor, D., 1999. Microsoft Office 2000 Visual Basic Fundamentals, Microsoft Press.

Bracha, G., Odersky, M., Stoutamire, D., Wadler, P., 1998. Making the future safe for the past: adding genericity to the Java programming language. In: Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98), ACM SIGPLAN Not. 33 (10), pp. 183–200.

Coplien, J.O., Schmidt, D.C., 1995. Pattern Languages of Program Design. Addison-Wesley, Reading, MA.

Cordy, J.R., Halpern-Hamu, C.D., Promislow, E., 1991. TXL: a rapid prototyping system for programming language dialects. Comput. Languages 16 (1), 97–107.

Crew, R.F., 1997. ASTLOG: A Language for Examining Abstract Syntax Trees. In: Ramming (1997).

Edwards, P.D., Rivett, R.S., 1997. In: Daniel, P. (Ed.), Towards an Automative 'Safer Subset' of C. Proceedings of the 16th International Conference on Computer Safety, Reliability and Security: SAFECOMP '97, European Workshop on Industrial Computer Systems: TC-7. Springer, New York, pp. 185–195.

Faith, R.E., Nyland, L.S., Prins, J.F., 1997. KHEPERA: A System for Rapid Implementation of Domain Specific Languages. In: Ramming (1997).

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA.

Gansner, E.R., North, S.C., Vo, K.P., 1988. DAG – a program that draws directed graphs. Software-Pract. Exper. 18 (11), 1047–1062.

ISO8879, 1986. Information processing – text and office systems – standard generalized markup language (SGML). International Organization for Standardization, Geneva, Switzerland, ISO 8879.

Jackson, M., 1999. Specializing in software engineering. IEEE Software 16 (6), 119–121.

Johnson, S.C., 1975. Yacc – Yet another compiler–compiler, Computer Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, USA, July.

Johnson, S.C., Lesk, M.E., 1987. Language development tools. Bell Syst. Tech. J. 56 (6), 2155–2176.

Kamin, S.N., Hyatt, D., 1997. A Special-Purpose Language for Picture-Drawing. In: Ramming (1997).

Kernighan, B.W., 1975. Ratfor – a preprocessor for a rational Fortran. Software-Pract. Exper. 5 (4), 395–406.

Kernighan, B.W., 1982. PIC – a language for typesetting graphics. Software-Pract. Exper. 12, 1–21.

Kernighan, B.W., Cherry, L.L., 1974. A system for typesetting mathematics, Computer Science Technical Report 17, Bell Laboratories, Murray Hill, NJ, USA, May.

Kernighan, B.W., Plauger, P.J., 1978. The Elements of Programming Style, second ed. McGraw-Hill, New York.

Kernighan, B.W., Ritchie, D.M., 1979. The M4 Macro Processor. In: Unix Programmer's Manual (1979).

Klarlund, N., Schwarzbach, M.I., 1997. A Domain-Specific Language for Regular Sets of Strings and Trees. In: Ramming (1997).

Landin, P.J., 1966. The next 700 programming languages. Commun. ACM 9 (3), 157–166.

Lesk, M.E., 1975. Lex – a lexical analyzer generator, Computer Science Technical Report 39, Bell Laboratories, Murray Hill, NJ, USA, October.

Lesk, M., 1979a. Some Applications of Inverted Indexes on the Unix System. In: Unix Programmer's Manual (1979).

Lesk, M.E., 1979b. TBL – A Program to Format Tables. In: Unix Programmer's Manual (1979).

Lutz, M., 1996. Programming Python, O'Reilly and Associates.

McMahon, L.E., 1979. SED – A Non-interactive Text Editor. In: Unix Programmer's Manual (1979).

Motor Industry Research Association, 1994. Development Guidelines for Vehicle Based Software, November.

Nipkow, T., von Oheimb, D., 1998. $Java_{light}$ is type-safe–definitely. In: Proceedings of the Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 161–170.

Ossanna, J.F., 1979. NROFF/TROFF User's Manual. In: Unix Programmer's Manual (1979).

Ousterhout, J.K., 1994. Tcl and the Tk Toolkit. Addison-Wesley, Reading, MA.

Paryavi, M.N., Hankley, W.J., 1995. OOSPEC: an executable object-oriented specification language. In: Proceedings of the of ACM 23rd Annual Computer Science Conference CSC '95, ACM, ACM Press, New York, pp. 169–177.

Poskanzer, J. et al., 1993. NETPBM: Extended Portable Bitmap Toolkit, Available online ftp://ftp.x.org/contrib/utilities/, December, Release 7.

Potter, B., Sinclair, J., Till, D., 1991. An Introduction to Formal Specification and Z. Prentice-Hall, Englewood Cliffs, NJ.

Ramming, J.C. (Ed.), 1997. In: Proceedings of the USENIX Conference on Domain-Specific Languages, USENIX, Santa Monica, CA, USA, October.

Rawls, R.R., Hagen, M.A., 1998. Autolisp Programming: Principles and Techniques. Goodheart-Willcox.

Rumbaugh, J., Jacobson, I., Booch, G., 1999. The Unified Modeling Language Reference Manual. Addison-Wesley, Reading, MA.

Savitch, W., 1995. Pascal – An Introduction to the Art and Science of Programming, fourth ed. Benjamin/Cummings, Menlo Park, CA.

Schmidt, D.C., 1990. Gperf: a perfect Hash function generator. In: Proceedings of the USENIX C++ Conference, Usenix Association, San Francisco, CA, USA, pp. 87–100.

Simonyi, C., 1995. The Death of Computer Languages and the Birth of Intentional Programming, Technical Report MSR-TR-95-52, Microsoft Corporation, Redmond, WA, USA, September. Available online at ftp://ftp.research.microsoft.com/pub/tr/tr-95-52.ps.

Sommerville, I., 1989. Software Engineering, third ed. Addison-Wesley, Reading, MA.

Spinellis, D., Guruprasad, V., 1997. Lightweight Languages as Software Engineering Tools. In: Ramming (1997).

Spinellis, D., Drossopoulou, S., Eisenbach, S., 1995. Object-oriented technology in multiparadigm language implementation. J. Object-Oriented Programming 8 (1), 33–38.

Stallman, R.M., 1984. In: Barstow, D. R., Shrobe, H. E., Sandwell, E. (Eds.), EMACS: the extensible, customizable, self-documenting display editor, Interactive Programming Environments. McGraw-Hill, New York, pp. 300–325.

Sterling, L., Shapiro, E., 1986. The Art of Prolog. MIT Press, Cambridge, MA.

Stichnoth, J.M., Gross, T., 1997. Code Composition as an Implementation Language for Compilers. In: Ramming (1997).

Stroustrup, B., 1984. Data abstraction in C. Bell Syst. Tech. J. 63 (8), 1701–1732.

Turski, W.M., Maibaum, T.S.E., 1987. The Specification of Computer Programs. Addison-Wesley, Reading, MA.

Tyszberowicz, S., Yehudai, A., 1992. OBSERV – a prototyping language and environment. ACM Trans. Software Eng. Methodol. 1 (3), 269–309.

Unix Programmer's Manual, 1979. UNIX Programmer's Manual. Volume 2 – Supplementary Documents, seventh ed., Bell Telephone Laboratories, Murray Hill, NJ, USA (also available online http://plan9.bell-labs.com/7thEdMan/).

Wall, L., Schwartz, R.L., 1990. Programming Perl, O'Reilly and Associates, Sebastopol, CA, USA.

Wirth, N., 1974. In: Rosenfeld, J.L. (Ed.), On the design of programming languages. In: Information Processing 74: Proceedings of IFIP Congress 74, International Federation for Information Processing. North-Holland, Stockholm, pp. 386–393.

Diomidis Spinellis holds an MEng in Software Engineering and a Ph.D. in Computer Science both from Imperial College (University of London, UK). He is an assistant professor at the Department of Information and Communication Systems, University of the Aegean, Greece. He has contributed software to the 4.4BSD Unix distribution, the X-Windows system, and is the author of a number of open source software packages, libraries, and tools. His research interests include Software Engineering, Programming Languages, and Information Security. Dr. Spinellis is a member of the ACM, the IEEE, and a founder member of the Greek Internet User's Society. He is a co-recipient of the Usenix Association 1993 Lifetime Achievement Award.