Reusable Architectural Decisions for DSL Design

Foundational Decisions in DSL Development

Uwe Zdun[1] and Mark Strembeck[2]
[1]*Distributed Systems Group, Information Systems Institute*
Vienna University of Technology, Austria
zdun@infosys.tuwien.ac.at
[2]*Institute of Information Systems, New Media Lab*
Vienna University of Economics and Business, Austria
mark.strembeck@wu-wien.ac.at

Domain-specific languages (DSL) receive a constantly growing attention in the area of software development. However, so far the documentation of reusable architectural knowledge for DSL design is rather limited. In this paper, we systematically explore the DSL design space by combining reusable architectural decision modeling and software patterns. In particular, we have explored three reusable architectural decisions in this design space: the decision for the type of DSL development process, the decision for the concrete syntax style, and the decision for developing an external vs. an embedded DSL. These decisions are foundational for each DSL project. Each of these decisions has a number of (candidate) patterns for DSL design. These (candidate) patterns define alternative solutions in the shared context and problem space of the respective reusable architectural decision.

1 Introduction

Domain-specific languages (DSLs) are languages that are specifically tailored for the needs of a particular problem or application domain. They promise that domain experts themselves can understand, validate, modify, test, and sometimes even develop DSL programs. DSLs receive a constantly growing attention in recent years (see, e.g., [11, 20, 30, 48]), and with the DSL toolkits becoming more mature (see, e.g., [11, 24]), DSLs are not only used in niches anymore. For many architects, DSLs are an interesting approach that can potentially be used in projects where domain knowledge is hard to capture. In such a context, DSLs can help as they aim to enable involving domain experts more closely in the software design activities. Unfortunately, making architectural decisions[1] in a particular DSL development project is not always easy because many of the decisions that must be taken by the architect may have important consequences for other parts of the project. In addition, many existing DSL toolkits pre-select a number of decision options. Therefore, choosing a particular toolkit too early in a project, i.e. without a sound understanding of the decision's impact, may have severe consequences on the project as a whole.

Nevertheless, many of the architectural decisions, as well as their forces and consequences, are recurring in different DSL projects. Recurring design decisions are often described in pattern form [18].

[1]In this paper, we define the term "architectural decision" as follows: A decision qualifies as an *architectural decision*, if it affects the architecture of a software system or the role of the architect.

However, so far, only a few patterns have been proposed for DSL development [17, 28, 44]. Moreover, in the context of DSL development, not all options and alternatives of an architectural decision can be well described in the pattern form, as patterns come with certain well-founded assumptions, such as *not* describing novel solutions, having at least three known uses, being non-obvious, and so on (see [5]). In order to cover the whole design space of DSLs, we thus want to explore the use of a *reusable architectural decision model* for capturing design decisions for DSL development.

A consequence of this approach is that by systematically exploring the *design space* of DSL design, we can also identify patterns more systematically. That is, the alternatives identified for an architectural decision can be seen as (candidate) patterns. Each of these (candidate) patterns shares the context and problem of the reusable architectural decision, but offers a different alternative solution to the decision's problem. In particular, we think that focusing on the decision making process, instead of focusing on the individual patterns, enables us to put more emphasis on the links between the patterns and the rules defining how these patterns can be combined (i.e., the "pattern language aspect"). However, this is only a side-aspect of our paper: The focus of the paper is on the architectural decisions in the area of DSL design.

In particular, we start to explore the DSL design space by describing three decisions in a reusable architectural decision model. These decisions are foundational decisions to be made in any DSL project:

- The decision on the **DSL development process**[2] deals with the problem how the DSL development process is organized. This decision directly influences many other decisions during DSL development. The decision has three alternative solutions: LANGUAGE MODEL DRIVEN DSL DEVELOPMENT[3], MOCKUP-LANGUAGE DRIVEN DSL DEVELOPMENT, and EXTRACTING THE DSL FROM AN EXISTING SYSTEM.
- The decision **concrete syntax style** deals with the problem of how to present the concrete syntax (or syntaxes) of a DSL to the DSL user. The decision has four alternative solutions (the last one being a variant of the first one): TEXTUAL CONCRETE SYNTAX, GRAPHICAL CONCRETE SYNTAX, FORM-/TABLE-BASED CONCRETE SYNTAX, and TEXTUAL CONCRETE SYNTAX WITH GENERATION OF VISUALIZATIONS.
- The decision **external vs. embedded DSL** deals with the problem of choosing a style for parsing and interpreting the DSL's concrete syntax in order to map it to an abstract syntax representation. The decision has three alternative solutions (the last one being a variant of the other two): EXTERNAL DSL, EMBEDDED DSL, and HYBRID DSL.

These decisions are reusable in the sense that they can be followed in different projects. For all three decisions, multiple alternatives and variants thereof can be selected. The decision descriptions explain in detail the forces of the decisions.

The target audience of this work are software designers, engineers, and architects who conduct DSL projects or who are interested in DSL design and development; in particular those who are new to DSL development. In addition, we address the patterns community and interested researchers by investigating reusable decision models as a way of systematically exploring a design space for pattern mining.

[2]We use **boldface** font to markup the names of reusable architectural decision.
[3]We use SMALLCAPS font to markup alternative solutions to architectural decisions.

This paper is organized as follows: In Section 2, we provide an overview of the main DSL concepts and artifacts for readers who are not familiar with this topic. Next, Section 3 gives an overview of the reusable architectural decisions described in this paper. Subsequently, Sections 4-6 present the three reusable architectural decisions in detail. In the appendix, we describe how we integrate reusable architectural decision models and patterns in this paper. In addition, the appendix discusses their synergies, our decision template, and how decisions and their alternatives can be interpreted as (candidate) patterns.

2 Background: Domain-Specific Languages

A *domain-specific language* (DSL) is a tailor-made (computer) language for a specific problem domain. In this sense, DSLs differ from general purpose languages, such as C, C#, Java, Perl, Ruby, or Tcl, that can be applied to arbitrary problem domains. In particular, DSLs are often not Turing complete, and they only provide abstractions suitable for one particular problem domain. However, this specialization results in significant gains in expressiveness and ease of use in the DSL's application domain compared to a general purpose language. In recent years, domain-specific languages received a constantly growing attention, especially in the areas of model-driven development (MDD, see, e.g., [16, 41, 42, 45]) and dynamic languages [12]. The basic idea of domain-specific languages, however, already has a long history (see, e.g., [1, 19, 27]).

In general, DSLs can be designed and used on different abstraction layers, ranging from DSLs for technical tasks to DSLs for tasks on the business-level. Thus, DSLs can also be defined for non-technical stakeholders, such as business analysts or biologists, for example. DSLs provide higher-level language abstractions that enable domain experts to either directly configure the behavior of software systems, or to take part in the configuration, without previous knowledge of a general purpose programming language.

Software development with DSLs, also referred to as *language-oriented programming* (see [8, 54]), has additional advantages from a software engineering perspective: Development of and with DSLs results in a clear separation of concerns in different dimensions. First, it clearly separates domain-specific concerns from other facets of the software system, such as the technology platform on which the DSL runs. Second, it separates the DSL's design from platform-specific DSL implementation(s). Third, the concerns of different problem domains can be separated in different DSLs.

By narrowing the development efforts to a clearly defined and relatively small application domain, software engineering projects become more manageable. This is an important factor because the complexity of software engineering projects still causes many projects to substantially exceed their budget and projected time frame, or to fail entirely (see, e.g., [2, 4, 21, 46]).

However, the need to define a DSL and to build the corresponding infrastructure can only be justified if the advantages of DSLs are required in the respective software development project. Often the development of a DSL does not pay off with a single use, but rather with the second or third use of the DSL. Therefore, one needs to weigh the benefits of a DSL against the costs of building the DSL infrastructure (see, e.g., [45]).

Figure 1 depicts the main DSL artifacts and their relations[4]. From a language user's point of view,

[4]Please note that these artifacts are relevant for any DSL. For many existing DSLs, however, not each of these artifacts is necessarily realized as a separate artifact. For instance, the DSL's core language model is often not explicitly specified, but only existing in the developers' minds and/or buried in the code.

the DSL consists of language elements. The definition of these language elements is provided by the language model. The DSL's *language model* specifies elements from the DSL's target domain. Some authors call the language model the *abstract syntax* of the DSL (see, e.g., [16]). The language model is a composite model that consists of three sub-models (see Figure 1).

The *core language model* captures all relevant domain abstractions and specifies the relations between these abstractions. Examples of domain abstractions are account, bond, client, fund, stock, or stock order in the banking domain; pre-condition, post-condition, test case, or test result in the domain of software testing; role, subject, permission, or constraint in the domain of access control. The DSL's language model thus formalizes domain-specific knowledge and must be validated by domain experts. The language model can be defined using any suitable modeling language, such as the UML [35] and UML extensions.

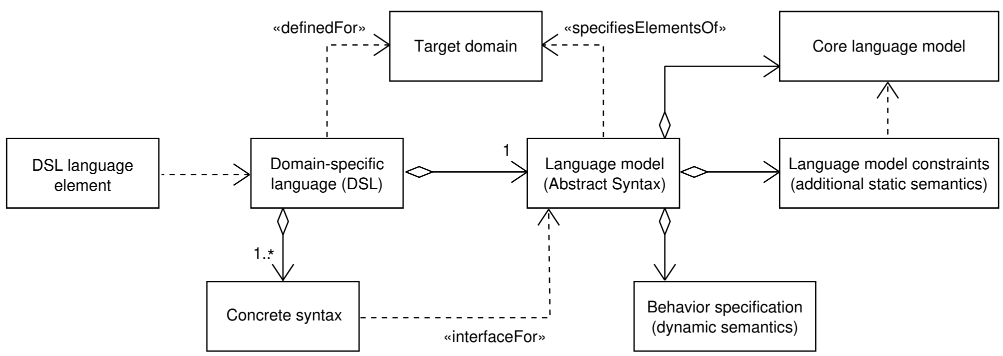


Figure 1: Domain-specific languages: Artifacts

The *language model constraints*, also referred to as (additional) *static semantics*, are essentially a part of the DSL language model (see Figure 1). They express invariants on model elements and/or on relations between those elements and thereby define semantics that cannot be directly expressed in the (graphical) core language model. An example from the access control domain would be an invariant which defines that two mutual exclusive roles must never be assigned to the same user. Usually, constraints are provided in a formal constraint language. If the language model is defined using the UML, for example, such constraints can be specified using the Object Constraint Language (OCL) [34].

The *DSL behavior specification*, sometimes also referred to as *dynamic semantics*, is a part of the language model and defines the (behavioral) effects that result from using a DSL language element. Moreover, it defines how the DSL language elements can interact at runtime. The behavior can be specified in many different ways, ranging from high-level control flow models, over detailed behavioral models, to a precise textual specification.

In addition to the artifacts specifying the abstract syntax of the DSL, each DSL needs a concrete syntax to use the DSL in a certain system environment. A *concrete syntax* represents the abstractions defined through the DSL's abstract syntax, and each DSL can have multiple concrete syntaxes, e.g. a graphical syntax and a textual syntax.

The concrete syntax serves as the DSL's interface. Thus, the definition of the concrete syntax(es) is especially important from the DSL user perspective. In case a DSL is primarily built for non-programmer human users, usability properties of this interface, such as being simple and intuitive, are of central importance. In an ideal case, the concrete syntax is therefore both, convenient for human users and easy to process by software components. If a DSL is primarily built for technical experts and/or machine interaction, however, a more complex concrete syntax may be acceptable or useful.

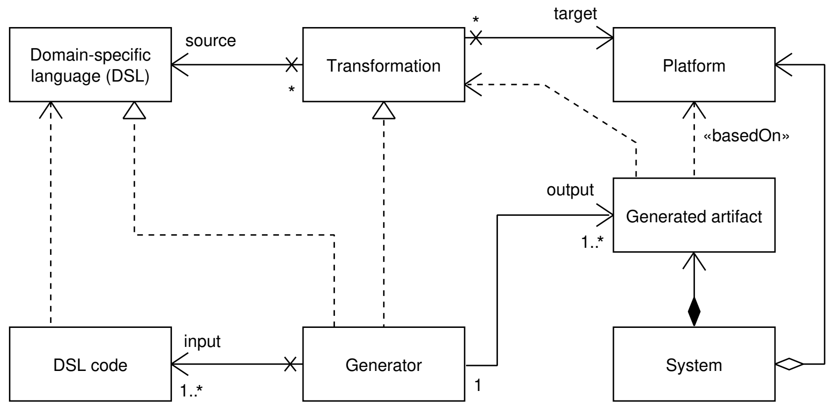


Figure 2: DSL transformations: Concepts (loosely based on [45])

Transformations are defined to transform DSL code written in a concrete syntax to another model representation or to (programming language) code that can be executed on a specific platform (see Figure 2). In general, a *transformation* is a directive that defines how one (model) format is to be transformed to another (model) format (see, e.g., [29, 43]).

A *platform* consists of software building blocks that provide functions to implement the DSL's semantics in a specific system environment. The platform consists of generic platform artifacts, such as programming languages and frameworks (for example based on the Enterprise JavaBeans (EJB) technology or Microsoft .NET), as well as DSL-specific platform artifacts (i.e. parts of the software platform that need to be implemented or integrated into a generic platform just to support the DSL). DSL-to-platform transformations are conducted by a *generator* component (see Figure 2) and result in *generated artifacts* (such as Java, C#, or Ruby code for example) based on the respective platform (see, e.g., [7, 24, 45]).

3 Overview: Reusable Architectural Decisions for DSL Design

In this paper, we present the three architectural decisions shown in Figure 3. The typical sequence of the three decisions is that first we determine the **DSL development process**, second choose the **concrete syntax style**, and third decide whether implementing an **external vs. embedded DSL** best suits the needs of a particular DSL project.

Even though this sequence of the three decisions is a typical way of considering them, virtually any of the three decisions can be the entry point of the design space. Thus, from our experiences, it is not possible to predetermine a particular decision sequence that fits for every DSL project.

For example, for the **concrete syntax style** decision it must be checked whether a new concrete syntax is needed in the DSL project. This is because in some cases developing a new concrete syntax is not necessary, e.g. if a syntax has already been selected. For instance, if we conduct a DSL project that intends to use the BibTex syntax as its input source, we don't need to decide for a syntax style, as it is already pre-determined. In such a case the decision on the **concrete syntax style** can be omitted. Moreover, the decision **external vs. embedded DSL** depends on the **concrete syntax style** decision, as only TEXTUAL CONCRETE SYNTAX and its variants require this follow-on decision. This is because with today's programming languages choosing a GRAPHICAL CONCRETE SYNTAX or a FORM-/TABLE-BASED CONCRETE SYNTAX always results in developing an external DSL.

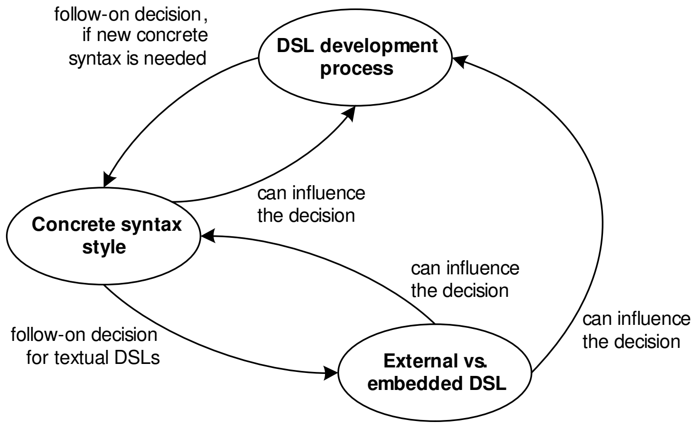


Figure 3: Design space overview

Figure 3 shows the main relations between the decisions. As explained in the appendix, relationships between decisions are mainly mandatory or optional "follow-on decision" relationships (see Figure 12). Most often a follow-on decision is only selected if particular alternatives are selected or applied in specific circumstances. Hence, we use the rather vague "can influence the decision" relationships in Figure 3 (the respective consequences and liabilities will be detailed in the text below).

4 Architectural Decision: DSL Development Process

Context

DSL development is usually embedded in some general purpose software development process, such as the Unified Process or an agile process. In addition, there are many DSL project-specific factors that require special attention, such as end-user or domain expert involvement. These factors lead to the need to choose a development micro-process that best fits a particular DSL project. This micro-process is then used as part of the corresponding general purpose development process (see [48]).

Problem

How do you conduct the DSL development micro-process, considering the influencing factors of a certain DSL project?

Forces

- *Type of DSL project*: In general a DSL project can be conducted to build a *new DSL*, it can be a maintenance project to build a *revision* of a DSL, or it can be a *documentation* project for a pre-existing DSL. A documentation project builds an ex-post documentation for a DSL that resulted from a previous development project but is not yet documented in a sufficient manner. The project type directly affects the corresponding DSL development process.

- *Intended (main) users of the DSL*: The intended user(s) of a DSL may for instance be software developers or end-users[5] from a specific application domain (e.g. bank clerks or physicians). For example, different means are applied to communicate and interact with end-users in contrast to communicating with software developers. In particular, end-users do most often have no or only marginal knowledge of software development techniques or related areas (such as software modeling or programming languages). Moreover, software developers often prefer other DSL syntaxes than end-users. Thus, the intended users of a DSL directly affect the different activities that are conducted when developing a DSL.

- *Availability of domain expert(s)*: As DSLs are tailor-made for a particular target domain, a correct and comprehensive understanding of the target domain is paramount to building a DSL. This is especially true if we build a DSL for a specialized end-user domain (e.g. for the specification of financial products or for the definition of medical procedures). In such a case, DSL developers depend on corresponding domain experts in order to develop the DSL. Here, the frequency and the length of the time periods a domain expert is available to the DSL developers, directly influences the DSL development process.

- *Software-technical qualification of domain expert(s)*: Consider we build a DSL for software developers (e.g. a DSL for software testing or a DSL for software documentation). In this case, DSL developers are either themselves domain experts or they can check back with other software developers to shape the DSL. Therefore, developing DSL's for software experts is comparatively easy and requires no specific adjustments of the process. However, if the domain experts are end-users, who are not familiar with software engineering methods and techniques, the DSL process must consider bridging the gap between domain knowledge and software-technical knowledge. This means either the domain experts need to be educated in software modeling (to a certain degree) and similar software-technical knowledge to take part in DSL development, or the DSL engineers must use different means (e.g. other than standard modeling languages such as the UML) to present end-users the corresponding models in an understandable way and to establish a common understanding of the corresponding domain abstractions.

- *Accessibility of the target platform*: Often DSLs are build to raise the accessibility of a target platform for a particular type of domain expert. In this case, the DSL is build to provide an additional interface to the corresponding platform. For example, if we have a platform that provides a graphical user interface but lacks a well-documented API for software developers, a corresponding TEXTUAL CONCRETE SYNTAX can provide a convenient interface for developers to control system functions. If a platform provides a well-documented API but lacks a graphical interface, building a graphical DSL might provide a corresponding interface for end-users.

- *Type of the used general purpose development process*: Because DSL development is most often conducted in context of other software development activities, and therefore embedded in a larger software project, the development process of the overall software project significantly influences the DSL development micro-process. In case DSL development is embedded in an agile software development project, for example, we typically have a few small teams that especially use source code as their primary development artifact. If, however, DSL development is embedded in a large-scale and more formal development process, for example based on the Unified Process or using a customized V-Model process, it is likely that more people are involved

[5]For the sake of simplicity, we use the term "end-user" throughout the paper for all users who are not software developers. Note that the term "domain expert" is more generic and can be applied to experts from the domain of software engineering as well as to experts from a particular application domain.

in the different development activities and that we have a significantly increased coordination and communication effort. In such large-scale projects source code often is not used as the primary means for communication between the different stakeholders. Instead models and specifications are used as primary development artifact. Therefore, it is important to choose a DSL process variant that best fits with the respective project and the "culture" of the corresponding organization.

Alternative Solutions

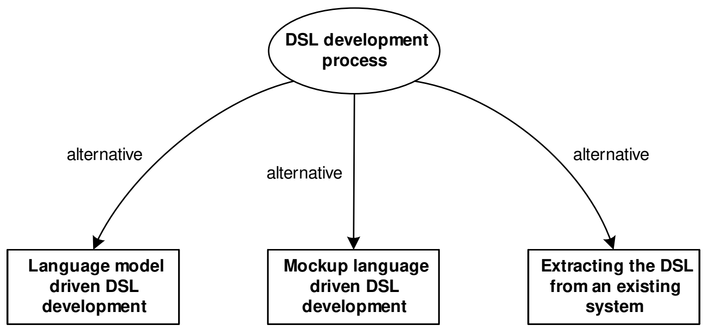


Figure 4: Decision overview: DSL development process

Figure 4 illustrates the alternative solutions for this architectural decision.

4.1 Alternative: Language Model Driven DSL development

Solution

At first, select the target domain of the DSL. Next, define an (initial) core language model for this target domain. The main results of this activity are the DSL core language model and corresponding language model constraints. Subsequently, define the DSL's behavior and at least one concrete syntax. Finally, map the different artifacts to the target platform.

In most cases you should conduct the two activities of defining the DSL's behavior and the concrete syntax in parallel because the behavior and the concrete syntax of a DSL may mutually affect each other. Performing the two activities in parallel is especially useful for EMBEDDED DSLS because syntax and behavior of the host language[6] have a considerable influence on the concrete syntax and behavior of the DSL. Moreover, conducting both activities in parallel is often useful because intermediate results can be exchanged to produce a concrete syntax and corresponding behavior definitions which are consistent, complete, and well integrated with each other (see also [48]).

After defining the DSL's behavior and concrete syntax, we need to map the different artifacts to the target platform. This activity produces transformations and corresponding integration tests for the DSL. Furthermore, if the target platform needs to get extended or adapted in the course of the DSL development, we need to produce the corresponding platform extensions.

[6] Here the term "host language" refers to the programming language that is used to implement a DSL. For example, if an embedded DSL is developed with Ruby, Ruby is the host language of the respective DSL.

The SYNTACTIC ICING pattern [17] describes how to design a language for conceptual cleanness first and to add a convenient syntax in a subsequent step to avoid that syntax dominates the language design. This practice is inherent part of LANGUAGE MODEL DRIVEN DSL DEVELOPMENT, as it starts with modeling the target domain and defines a corresponding syntax later in the process.

Consequences

LANGUAGE MODEL DRIVEN DSL DEVELOPMENT comes with the following benefits:

- For a well-known domain, especially for DSLs where the DSL developers are domain experts themselves, it is often straightforward to define the core language model first. In such cases it is therefore sensible that the language model drives the development process.
- The process is also well-suited for DSL revision projects, e.g. to extend an existing DSL with additional language elements. In this case, we already have a verified version of the corresponding DSL artifacts which are then extended by adding one (or more) additional element(s) to the language model.
- The process is easy to understand and follow in a documentation. Thus, even if the DSL is defined using some other tailored process variant, it makes sense to document or explain the DSL with the language model driven process.

LANGUAGE MODEL DRIVEN DSL DEVELOPMENT has the following liabilities:

- In complex domains or in projects where the domain experts are only temporarily available, building a language model upfront is a complicated task. In such a context, it is often not sensible to let the language model drive the process.
- If domain experts should be involved in the process, they must have some knowledge of software modeling. However, in software engineering projects teaching modeling abstractions to domain experts is often out of scope, as it can be time-consuming or is not accepted by domain experts (e.g. resulting from other work priorities).

Known uses

- Within the context of a larger research project, we developed a DSL for *BPMN process modeling*. Two people were involved in architecting the DSL. The development was mainly carried out by one developer, but access to the DSL users was very limited (only a few meetings). An agile process and test-driven development were chosen for the overall project (the architect played the proxy for the DSL users when they were not available). The main focus of the project was on the evolution of the models – the TEXTUAL CONCRETE SYNTAX was mainly used as a convenient way to populate the models. A highly iterative version of the LANGUAGE MODEL DRIVEN DSL DEVELOPMENT micro-process was chosen, as it was the most efficient way to facilitate the communication of architects and developers between the project's regular releases.
- In another research project, we developed a family of DSLs for *SOA architectural views*. This project was a research project and development was carried out by 5 PhD students. Domain

knowledge was provided by regular architecting sessions, patterns on the domain, and infrequent discussions and reviews with industrial DSL users. Because the different developers worked on separate areas and DSLs with some overlaps, we decided to follow the LANGUAGE MODEL DRIVEN DSL DEVELOPMENT micro-process, as it turns out to be efficient for the discussions. This was feasible because most of the domain experts were familiar with modeling abstractions.

4.2 Alternative: Mockup-language Driven DSL Development

Solution

Start DSL development with the concrete syntax design and then distill the language model from this concrete syntax. This means, the concrete syntax is developed together with a domain expert by utilizing the domain expert's domain knowledge. Next, develop an initial prototype implementation of the resulting mockup-language. After a version of the prototype is completed, document the corresponding core language model and behavior definition that result from this version of the prototype. Subsequently, the process continues with the next iteration to further enhance the draft version of the DSL. This co-evolution of the different DSL artifacts continues until the domain experts and software engineers define the DSL as complete.

In this process, the initial DSL prototype serves as a means to perform acceptance testing activities, to further refine and tailor the concrete syntax to the domain expert's needs. This is done in multiple iterations to get closer to the domain abstractions in a stepwise manner.

MOCKUP-LANGUAGE DRIVEN DSL DEVELOPMENT is used in cases where a domain is very complex and cannot be easily understood by the software engineers (e.g. air-traffic control, definition of financial products, or power-plant control) and/or DSL engineering requires a very close cooperation of domain experts and DSL engineers. In such a project, it is sensible to drive the development process via the definition of a *mockup language*.

That means, the subprocess for definition of the concrete syntax is guiding the whole DSL development process. In particular, definition of the concrete syntax can either be performed iteratively (per language element) or a version of the concrete syntax can be completely defined (including all perceived language elements). For instance, in case a second concrete syntax is added to a DSL, it sometimes makes sense to specify the whole syntax at once. In our experience, a more iterative approach makes sense in case the language is used for incrementally illustrating and sketching domain concepts while the DSL is developed. This, of course, is dependent on the availability of the domain experts. Often MOCKUP-LANGUAGE DRIVEN DSL DEVELOPMENT is well-suited for projects that follow an agile development approach which advocate onsite domain experts (customers).

Consequences

MOCKUP-LANGUAGE DRIVEN DSL DEVELOPMENT comes with the following benefits:

- Because the concrete syntax provides the user interface of a DSL, it is a good means to raise the participation of domain experts in the development process and to get constant feedback on new DSL features. Thus, it is often very productive to let the development of the concrete syntax drive the process. This is also true for complex domains that the DSL developers have no detailed knowledge of.

- The mockup-language driven development process is well-suited to build an additional concrete syntax for an existing DSL. In this case, the language model already exist, and most development activities can concentrate on the development of the new concrete syntax and corresponding platform artifacts.

MOCKUP-LANGUAGE DRIVEN DSL DEVELOPMENT has the following liabilities:

- There is the danger to focus on syntax aspects only and to forget the language model design. It is more challenging to distill a "good" language model from a given syntax than to design it from scratch. Hence, the architect must make sure the design of the language model is adequately considered during mockup-language driven design.
- This process variant highly depends on the availability of domain experts. If domain experts are only available occasionally and/or at irregular intervals, the DSL development significantly slows down.
- The mockup-language driven development process only poorly supports the development of DSL revisions. This is because an additional language element does not significantly impact the concrete syntax. However, due to multifaceted dependencies between different DSL artifacts, a new language element often affects other DSL artifacts (i.e. dependencies and artifacts defined via the language model).
- Mockup-language driven language development is not suited for DSL documentation projects.

Known uses

- In a research project, we developed a DSL for *DSL editor specifications*. Two people were involved in architecting the DSL. The development was mainly carried out by one developer. In this project, the syntax of the language played an important role, as user acceptance was regarded as a crucial success factor for the project. Hence, the MOCKUP-LANGUAGE DRIVEN DSL DEVELOPMENT micro-process was chosen and executed in a highly iterative fashion. The mockup-language design was frequently discussed with potential DSL users to get user feedback early on.
- A DSL project for *software testing* was part of a larger research project where we investigated the testing of dynamically changing runtime structures and dynamically changing runtime behavior of software programs. Our target platform was the dynamic object-oriented programming language XOTcl, and the intended users (i.e. domain experts) are XOTcl developers. The goal of this project was to build an embedded Testing DSL in XOTcl. At the beginning of the project we only had basic knowledge of the testing domain. Therefore, we decided to choose the MOCKUP-LANGUAGE DRIVEN DSL DEVELOPMENT micro-process and started with the definition of a concrete syntax. Two other XOTcl developers were consulted as domain experts to provide feedback on the EMBEDDED DSL. Both domain experts were available on short notice. Based on the discussions, the interim versions of the resulting mockup-language were iteratively refined.
- A project for *data backup* was conducted as a case study to develop a DSL for the specification of backup policies. The intention was to build an EMBEDDED DSL for XOTcl developers. Thus, a major requirement was the specification of a syntax that can easily be used by XOTcl developers.

For this reason, we chose an agile development approach based on the MOCKUP-LANGUAGE DRIVEN DSL DEVELOPMENT micro-process. In particular, we used our own XOTcl expertise and collected feedback from two other domain experts (XOTcl developers) to iteratively adapt the syntax. The backup DSL was then implemented as an EMBEDDED DSL that can be used in other XOTcl programs.

4.3 Alternative: Extracting the DSL From an Existing System

Solution

Start by extracting the DSL from an existing system if the goal of the DSL development project is to define a DSL as an (additional) user interface for an existing software system. At first, identify (potential) language elements in the existing system/platform. Examples for such elements are system components or interfaces that can be mapped to DSL core language model elements. That is, the domain abstractions for the DSL can directly be derived from the existing system. After that, define a concrete syntax and the integration with the platform. Integration with the platform here primarily means the definition of transformations from models (that conform to the language model) to code that is executable on the target platform.

If the architecture of the corresponding software system is documented (e.g., using a graphical modeling language such as the UML), this architecture description can be a valuable source for the elicitation of domain abstractions and DSL behavior. Here, domain experts either take part in the elicitation process, or they review the resulting language model. The process can be further tailored depending on the availability of the domain experts and on how iterative the development process is.

Consequences

EXTRACTING THE DSL FROM AN EXISTING SYSTEM comes with the following benefits:

- The domain abstractions can be directly derived from the existing system/platform. Hence, less design effort for domain abstraction is needed. If the existing system model describes the domain abstractions correctly, it is likely that the domain abstractions are also correctly captured by the DSL model derived from the existing system.
- The existing system clearly defines the scope of the DSL and predetermines the behavior and semantics of DSL elements.

EXTRACTING THE DSL FROM AN EXISTING SYSTEM has the following liabilities:

- Sticking to the existing domain abstractions can hinder innovative ideas that often emerge during a DSL project.
- The process variant is not well-suited if the DSL should significantly deviate from the concepts and abstractions of the existing system or platform. This can for instance be the case, if the DSL project is launched to refactor the existing system (and domain abstractions should change during refactoring).

Known uses

- In a re-engineering project for a *document archiving* system, we applied the EXTRACTING THE DSL FROM AN EXISTING SYSTEM approach. This project was an industry project with the goal to re-engineer a large scale industry system. Little languages were used for various tasks in this system. The goal of the project was to architecturally improve the system. The existing little languages of the system were improved using a unified DSL concept. The domain concepts as implemented by the existing system were not in question. The main users of the DSL were the system developers and teams configuring the system for customers. For these reasons and because no changes to the domain abstractions were needed, it made sense to follow the EXTRACTING THE DSL AN FROM EXISTING SYSTEM micro-process – and then gradually improve the design in frequent meetings.

- In a project for a *role-based access control* DSL, we aimed to build an EMBEDDED DSL for the xoRBAC platform [33, 47]. In this project, changing or extending the xoRBAC platform was not intended. Moreover, an API documentation of xoRBAC was available. Therefore, we chose to follow the EXTRACTING THE DSL FROM AN EXISTING SYSTEM micro-process. Because of our experiences from a previous RBAC DSL project, and because of our detailed knowledge of the xoRBAC platform, we were able to conduct the project in only four weeks. No additional domain experts were involved in the project.

4.4 Recommendation

Software engineers often like to start exploring a new system design or discussing a design iteration using models – e.g. such models are drawn by the development team in a whiteboard design session. This technique lends toward a LANGUAGE MODEL DRIVEN DSL DEVELOPMENT micro-process for DSL design, and is applicable both for building a DSL from scratch and building a revision. However, often this technique is difficult to apply in a DSL project. For instance, it can be hard to discuss models with end-users, if they are not familiar with software modeling languages. If participation of external domain experts is not possible, the software developers and designers must have a good knowledge of the target domain in order to apply the LANGUAGE MODEL DRIVEN DSL DEVELOPMENT micro-process. However, if domain experts participate in a DSL development project, they must have knowledge about the modeling abstractions used in the language model design (which, again, might require additional training effort for the domain experts).

If a new DSL is built from scratch, and one of the above mentioned conditions does not apply, it is often sensible to organize the development process following a MOCKUP-LANGUAGE DRIVEN DSL DEVELOPMENT process. Here, the concrete syntax of the corresponding DSL is in focus of the development process. The other artifacts, such as the core language model and the behavior specification, are derived from the concrete syntax in subsequent development activities. A benefit of this alternative is that participating domain experts do not need knowledge about software modeling languages.

In case we like to build a new revision of a DSL and the software developers understand the domain well enough, we apply a LANGUAGE MODEL DRIVEN DSL DEVELOPMENT process. Because we already have an existing version of the DSL (and the corresponding DSL artifacts), it is often a straightforward task to extend an existing DSL language model with additional language abstractions and/or additional behavior. Likewise, we use LANGUAGE MODEL DRIVEN DSL DEVELOPMENT if we conduct a documentation project of an already existing DSL. In this case, the DSL is already implemented and the respective language model can be derived directly from the DSL's implementation.

Therefore, the LANGUAGE MODEL DRIVEN DSL DEVELOPMENT micro-process is also well-suited to be applied in DSL documentation projects.

If we like to build a DSL as an (additional) interface to an existing system, we apply EXTRACTING THE DSL FROM AN EXISTING SYSTEM. In this case, language abstractions for the DSL can be derived from the corresponding target system and/or the target system's documentation.

In each of the process variants, we typically conduct several iterations of the different development activities to evolutionary enhance the DSL. In our experiences, an iterative and incremental development approach allows for a co-evolution of the different DSL artifacts and thereby directly contributes to enhancing the overall quality of the DSL (see, e.g., [48]).

The EVOLVING LANGUAGE pattern [17] describes an iterative process when creating a programming language: to start with a small/minimal feature set and then grow the language. This practice has been recommended for all alternatives provided in this architectural decision. In different projects, however, the size of the increments (ranging from a single element of the DSL language model to multiple, related elements) may vary significantly. In [48], we focus the detailed description of a systematic approach for DSL development. In particular, [48] provides a complementary view to the discussion in this paper.

5 Architectural Decision: Concrete Syntax Style

Context

Development of the concrete syntax for a DSL, including the supporting frontend components such as parsers or graphical editors.

Problem

In which style should the concrete syntax or syntaxes of a DSL be presented to the DSL user?

Forces

- *Expressiveness of DSL:* An important criterion for DSL syntax design is which kinds of models can be expressed in the syntax and how difficult it is to express a model in the DSL syntax.
- *Communication of models to DSL stakeholders:* Models expressed in the DSL should be easy to communicate to all stakeholders, including domain experts and technical experts. This includes that the DSL's concrete syntax is tailored to the level of understanding that the respective DSL user needs for his work with the DSL.
- *Support for understanding models expressed in the DSL:* Depending on the use cases of the DSL, the DSL must support at least one of the following scenarios. a) the DSL should enable DSL users to get a quick overview of the models expressed in the DSL. b) the DSL should also allow DSL users to delve into all details necessary for their task. In some cases it is necessary to support both scenarios. This is especially the case if different kinds of DSL users exist such as technical experts and end-users.

- *Effort for building tools such as editors:* For some syntaxes, it is mandatory to build an editor; i.e., without an editor the syntax cannot be used adequately (think of a GRAPHICAL CONCRETE SYNTAX for example). In other cases, an editor with features, such as error highlighting, help texts, syntax highlighting, and so on, is a nice-to-have. For different kinds of syntax, different efforts are required to build a full-featured editor (see also [52]).

- *Integrability with existing tools:* Users of a DSL usually have an existing set of tools they work with. This includes modeling and analysis tools, but also developer tooling such as grep, diff, svn, cvs, merge, and so on. Depending on the use cases for the DSL, it is either a requirement or simply useful to be able to integrate with the preferred tools of DSL users (cf. [52]).

- *Model evolution and change:* Models expressed in a DSL may change frequently. Different DSLs have different properties regarding the efforts needed to change a model expressed in the DSL. For instance, performing automatic changes (e.g. via a change script) is usually difficult for graphical syntaxes. Of course, changes can also be performed on the abstract syntax, for instance using a model-to-model transformation. However, in case we have to conduct many "small" changes, it is often not feasible to develop a model-to-model transformation for each of these changes. In such a case, the change must be performed "manually" by using the concrete syntax (see, e.g., [52]).

- *Acceptance by developers:* Developers usually accept languages for different reasons. Examples are that the language gets the technical job done, it integrates well with the existing tooling and other languages/frameworks, it is similar to the languages the developers are used to, it supports a working style similar to what they are used to, and so on. As software developers usually perform most of the major development tasks, and end-user programming by domain experts is still rather the exception, it is often important that the DSL does not get in the way of the developers and is well accepted by them. This force is also discussed in [52].

- *Selling the DSL:* The decision for and the evaluation of the success of a DSL project is often done by non-technical stakeholders such as managers, rather than by domain experts and DSL developers. As the concrete syntax and its tooling is the visible part of the DSL presented to those stakeholders, the choice of a particular concrete syntax can be very important for selling the DSL. Similarly, the technical solution envisioned by the DSL developers needs to be sold to the domain experts.

Alternative Solutions

Figure 5 illustrates the alternative solutions for this architectural decision.

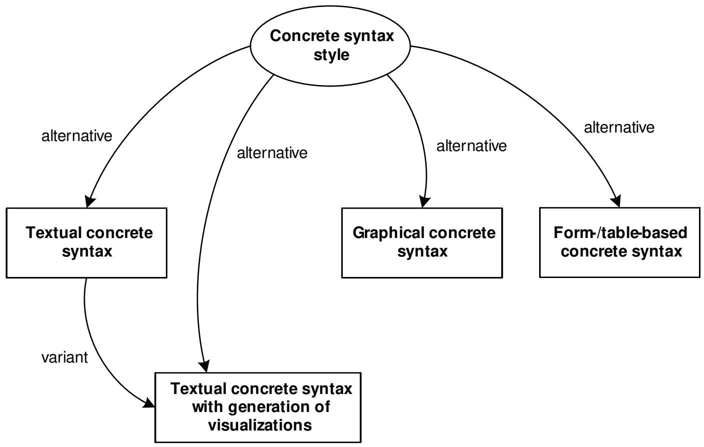


Figure 5: Decision overview: Concrete syntax style

5.1 Alternative: Textual Concrete Syntax

Solution

Use a text-based input to describe models[7] in the DSL. The text-based input can be edited with ordinary text editors or special IDEs or IDE plugins for the DSL. Parse or interpret the text-based input in order to map it to the abstract syntax of the DSL.

A TEXTUAL CONCRETE SYNTAX should be used if users of the DSLs primarily use text-based input to develop models in the DSL. The TEXTUAL CONCRETE SYNTAX is processed using a parser. The parser can be developed from scratch using a parser generator, such as Antlr or COCO/R. Alternatively, the interpretation infrastructure of an existing (programming) language can be customized to enable DSL parsing. Often scripting languages and dynamic languages, such as Ruby, Tcl, Python, or Perl are used in this context, as they are easy to extend with new language elements.

In both cases, a mapping to the abstract syntax of the DSL is needed. That is, if DSL text is parsed, the output of the parsing step must provide a model conforming to the DSL's abstract syntax.

In any case, a TEXTUAL CONCRETE SYNTAX can be edited with an ordinary text editor. However, often it makes sense to provide custom editors for the DSL including features such as error highlighting, syntax highlighting, help texts, and so on. In case a custom parser has been written, this step is mandatory if more support than simple text editing is needed. If an interpreted (programming) language is customized (i.e. extended with DSL-specific language abstractions), editors for the corresponding host language can be used to phrase DSL expressions – even though such editors most often do not provide native support for the DSL elements. If the customization is close to the original syntax of the host language, it is often possible to modify the existing editors. Otherwise a custom text editor is needed as well.

[7] Remember that DSLs reflect the abstraction level of the corresponding problem domain. Therefore, we speak of "models" that are developed using a DSL or that are defined via DSL expressions – even if the corresponding "models" are defined using a TEXTUAL CONCRETE SYNTAX and thus are more similar to source code written in a programming language at first glance. Here the textual syntax is just one way to (conveniently) phrase domain-specific expressions. Moreover, it is always possible to generate an additional graphical representation of textual DSL models, if needed (see also Sections 2 and 5.2).

Consequences

There are the following benefits of the TEXTUAL CONCRETE SYNTAX alternative:

- A TEXTUAL CONCRETE SYNTAX can express any formal language.
- For TEXTUAL CONCRETE SYNTAXES, it is not mandatory to build an editor – in principle, any text editor can be used to define DSL expressions. Nevertheless, full-featured editors are relatively easy to build, and many tools and frameworks exist to ease the tasks of building textual editors (cf. [52]).
- A TEXTUAL CONCRETE SYNTAX helps in understanding all technical details of DSL models.
- TEXTUAL CONCRETE SYNTAXES can be easily integrated with existing developer tooling such as diff or merge tools [52]. In addition, it is easy to use textual artifacts together with version management systems such as SVN or CVS.
- Advanced model evolution techniques working on the abstract syntax, such as model-to-model transformations, can be chosen to support model evolution. In addition, a TEXTUAL CONCRETE SYNTAX offers the fallback to use widespread text-based tools, such as search/replace or grep (see also [52]).
- TEXTUAL CONCRETE SYNTAXES are often more appreciated by developers (see, e.g., [52]).

There are the following liabilities of the TEXTUAL CONCRETE SYNTAX alternative:

- Especially, for high-level, domain-oriented concerns, TEXTUAL CONCRETE SYNTAXES require education of the domain experts in order to make them accept and understand a (formal) textual language.
- A TEXTUAL CONCRETE SYNTAX is often not the best way to get an overview or the "big picture" of a complex model because the primary model elements are often scattered in the textual code, which may be overloaded with details.
- Even though end-user development research shows that end-users who work in a domain are able and willing to learn a (formal) textual language if it suits their work goals (see, e.g., [32]), a textual language is often not highly appealing to non-technical DSL users. Hence, when selling a DSL or the DSL idea to non-technical domain experts, offering only a textual DSL can be an obstacle.

Known uses

- Typical examples of model-driven textual DSLs are DSLs developed using openArchitectureWare's xText framework [36] or Microsofts' OSLO framework [31]. Figure 6 shows an xText Grammar example (taken from Sven Efftinge's blog) for an introductory example originally described by Martin Fowler. From this grammar, xText can generate a language model (abstract syntax) and an Eclipse editor for the concrete textual syntax.
- Parr [37] presents a number of textual DSLs based on the ANTLR parser generator.

```
Statemachine :
  'events'
     (events+=Event)*
  'end'
  'commands'
     (commands+=Command)*
  'end'
  (states+=State)*;

Event :
  (resetting?='resetting')? name=ID code=ID;

Command :
  name=ID code=ID;

State :
  'state' name=ID
     ('actions' '{' (actions+=[Command])+ '}')?
     (transitions+=Transition)*
  'end';

Transition :
  event=[Event] '=>' state=[State];
```

Figure 6: Example: xText Grammar

- Frag [55] is a dynamic language that provide a framework called FMF for defining DSL language models. Figure 7 illustrates how an UML model can be mapped to an FMF language model. In the same syntax style the language model is defined, you can also define instances of that model. In addition, Frag allows you to define custom parsers for different textual syntaxes.

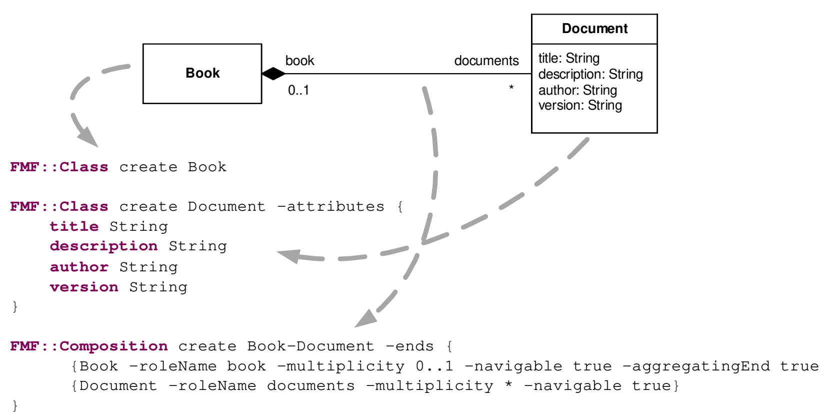


Figure 7: Example: Frag FMF Language Model Example

5.2 Alternative Variant: Textual Concrete Syntax with Generation of Visualizations

Variant of

TEXTUAL CONCRETE SYNTAX

Solution

Visualizing DSL programs or models from a corresponding textual syntax is often not too much effort. If a full-fledged graphical concrete syntax editor is not necessary, one can simply render the DSL models either for printing or presentation in a GUI, e.g. via tools like Graphviz [15] or Prefuse [38].

Consequences

There are the following benefits of this variant:

- As this variant is based on a TEXTUAL CONCRETE SYNTAX, any formal language can be expressed.
- As this variant is based on a TEXTUAL CONCRETE SYNTAX, existing textual editors can be reused, and it is easy to build a corresponding editor.
- A TEXTUAL CONCRETE SYNTAX helps in understanding all technical details of models. A proper visualization can help to understand the big picture and to quickly acquire an overview.
- A proper visualization can help to communicate DSL models to various stakeholders.
- A proper visualization can help to sell the DSL project as it is producing artifacts understandable for others stakeholders than developers, even though no dedicated graphical editor suite has to be developed.
- As this variant is based on a TEXTUAL CONCRETE SYNTAX, model evolution using simple text-based tools is possible.
- As this variant is based on a TEXTUAL CONCRETE SYNTAX, the syntax is usually accepted by developers.

There are the following liabilities of this variant:

- Additional effort for creating the visualization is needed. However, this effort is most often limited and not comparable to developing a full-fledged graphical editor.
- Even though a visualization helps in selling the DSL, it still is inferior to a full-fledged graphical tooling.

Known uses

- Voelter presents DSLs for architecture description with a TEXTUAL CONCRETE SYNTAX [52] and uses Graphviz [15] for visualization of the architecture diagrams. He generates the Graphviz files from the DSL language models using openArchitectureWare [36].
- Prefuse [38] can be used to create interactive visualizations of the content in a DSL language model.

```
Alternative create TextualConcreteSyntax -def {
      name {
            Textual concrete syntax
      }
      description {
            Using a textual syntax means ...
      }
      decision {
            ConcreteSyntaxStyle
      }
      pros {
            TextualSyntaxExpressiveness
            TextualSyntaxEffort
            TextualSyntaxUnderstandingDSLModelsDetails
            TextualSyntaxIntegrability
            TextualSyntaxModelEvolution TextualSyntaxAcceptanceByDevelopers
      }
      cons {
            TextualSyntaxCommunicationToStakeholders
            TextualSyntaxUnderstandingDSLModelsBigPicture
            TextualSyntaxSellingTheDSL
      }
      figureWidth 0.7
}
Consequence create GraphicalSyntaxExpressiveness -def {
      force ExpressivenessOfDSL
      description {
            A graphical syntax ...
      }
}

Consequence create GraphicalSyntaxUnderstandingDSLModelsBigPicture -def {
      force UnderstandingDSLModels
      description {
            A suitable graphical model ...
      }
}
```

Figure 8: Example: Frag textual syntax for an AD template (excerpt)

- In a Frag project [55], we developed a small DSL for describing architectural decisions using the same template that we apply in this paper. From the corresponding DSL models/programs we generated visualizations using Graphviz [15]. The Graphviz files are generated from DSL models/programs using Frag's transformation templates (see Figure 8).
- In [48], we described the engineering of an RBAC DSL. For this DSL, we developed a number of different syntaxes, including the option to visualize textual DSL models via a graphical tool (see Figure 9)

5.3 Alternative: Graphical Concrete Syntax

Solution

Design DSL models (or programs) by arranging symbols in a graphical editor. Use an interchange format or a textual syntax of the DSL to export the DSL models to other DSL components.

Often the resulting DSL models/programs are saved or exported by the editor tool in an interchange format, such as XMI. The interchange format is then parsed by another tool which maps the model to the abstract syntax of the DSL. In integrated tool suites, containing both a graphical editor and a generator in one tool, another option is to directly transform the internal memory representation of the model used inside the graphical editor to the abstract syntax of the DSL – i.e. without going through an interchange format.

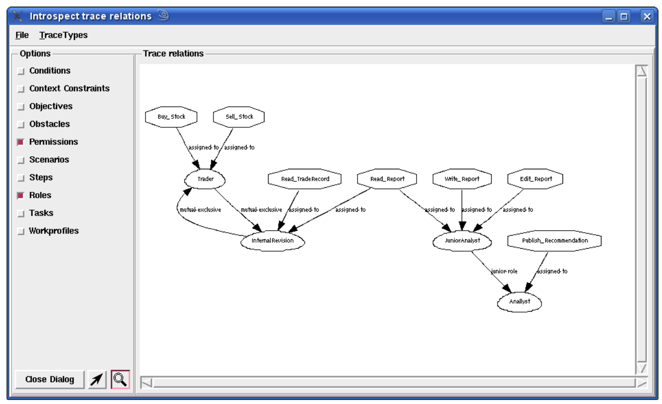
That is, the development of a graphical editor and a mapping from the internal representation used by this editor to the abstract syntax of the DSL is needed. In addition, depending on the use cases of the DSL (resp. of DSL models/programs), optional export and import components for an interchange format such as XMI are required.

```
<policySpec>
 <subject name="Sarah">
   <role>Analyst</role>
   <role>Trader</role>
 </subject>

 <subject name="Sophie">
   <role>InternalRevision</role>
 </subject>

 <role name="Analyst">
   <permission>Publish_Recommendation</permission>
   <junior>JuniorAnalyst</junior>
 </role>

 <role name="JuniorAnalyst">
   <permission>Read_Report</permission>
   <permission>Edit_Report</permission>
 </role>

 <role name="Trader">
   <permission>Buy_Stock</permission>
   <permission>Sell_Stock</permission>
   <ssd>InternalRevision</ssd>
 </role>

 <role name="InternalRevision">
   <permission>Read_TradeRecord</permission>
   <permission>Read_Report</permission>
   <ssd>Trader</ssd>
 </role>

 <permission name="Buy_Stock">
   <operation>buy</operation>
   <object>stock</object>
 </permission>

 <permission name="Read_TradeRecord">
   <operation>read</operation>
   <object>traderecord</object>
 </permission>

 ...

 <permission name="Write_Report">
   <operation>write</operation>
   <object>report</object>
 </permission>
</policySpec>
```

XML-based concrete syntax (external RBAC DSL)

```
Role create Analyst
Role create JuniorAnalyst
Analyst addJuniorRole JuniorAnalyst
Role create InternalRevision
Role create Trader
...
Permission create Read_Report -operation read -object report
Permission create Write_Report -operation write -object report
Permission create Edit_Report -operation edit -object report
...
JuniorAnalyst assignPermission Read_Report
JuniorAnalyst assignPermission Edit_Report
...
Subject create Sarah
Sarah assignRole Analyst
Sarah assignRole Trader

Subject create Sophie
Sophie assignRole InternalRevision
...
```

Textual concrete syntax for developers (embedded RBAC DSL)



Graphical representation of DSL models via platform specific tools

Figure 9: Different concrete syntaxes of a DSL for role-based access control [48]

Consequences

There are the following benefits of the GRAPHICAL CONCRETE SYNTAX alternative:

- In principle, a GRAPHICAL CONCRETE SYNTAX can express any formal language. However, in some cases it is challenging to design a GRAPHICAL CONCRETE SYNTAX in a usable and understandable fashion (e.g., if many details or many connections between modeling elements are needed).
- The biggest advantage of GRAPHICAL CONCRETE SYNTAXES is that they appear to be intuitive and easy to understand (provided that the visual/graphical syntax is designed to facilitate a good/intuitive understandability, of course). Hence, they are useful for discussions and presentations. Moreover, they are very useful for end-users who are not familiar with technical details of the DSL.
- A suitable graphical model can be a great advantage in conveying the big picture: “A picture says more than a thousand words.”

- Within the same graphical tool suite, it is often rather easy to add additional GRAPHICAL CONCRETE SYNTAXES or customize existing ones (if suitable syntax extension mechanisms are provided, and only within the boundaries and limitations given by the tool's syntax extension mechanisms, of course).
- An integrated graphical tool suite looks like a professional environment which is easy to understand and use, even for non-technical stakeholders. Therefore, a corresponding tool suite can help to "sell" the DSL (or the DSL idea) to end-users and/or managers.

There are the following liabilities of the GRAPHICAL CONCRETE SYNTAX alternative:

- If all technical details are packed into a graphical model or if the model is not properly partitioned, a graphical representation of a complex issue can get cluttered and/or huge, and hence not easy to understand. While the designer of a GRAPHICAL CONCRETE SYNTAX can provide means to partition the models, DSL users can always use the modeling elements in other ways than intended. Especially models used for generating code tend to be overloaded with details, as the details are needed for the generation process. Such models, however, are often hard to understand and, as a consequence, hard to communicate.
- Building custom graphical editors requires substantial efforts, even if supporting frameworks such as GMF are used (see, e.g., [52]).
- A GRAPHICAL CONCRETE SYNTAX is difficult to use with widespread text-based tools such as diff or merge (see, e.g., [52]). Thus, for graphical syntaxes, often only export formats (such as XMI) can be used easily with versioning tools such as SVN or CVS. However, textual export formats tend to be hard to understand for users of graphical tools. Hence, tasks like comparing versions requires either import into the graphical tool or the respective user is forced to learn the textual export format syntax. Moreover, some tools only support custom or binary formats (this is rather seldom and can hence be neglected in most cases). In general, different graphical tools tend to be very hard to integrate. Usually, if your organization is not the tool vendor, the only reasonable option is to integrate different graphical tools at the level of textual export formats.
- Changing a graphical model usually implies manual efforts. In some cases, such efforts can be substantial. In the ideal case, a model-to-model transformation on the abstract syntax can be used to perform the changes. But still the graphical layout of the modeling elements needs to be adjusted. Sometimes, however, a model-to-model transformation is not feasible (e.g., because of the efforts required to implement the transformation) or even impossible. In such cases, all affected models need to be changed manually. As some graphical models (and corresponding layouts) are harder to change than others, the frequency of changes should be considered as an input for the design of a GRAPHICAL CONCRETE SYNTAX.
- It is reported that developers tend to prefer TEXTUAL CONCRETE SYNTAXES over GRAPHICAL CONCRETE SYNTAXES. This relates to sentiments, such as "real developers don't draw pictures" [52], but has also many technical reasons such as the difficulties in integrating and changing GRAPHICAL CONCRETE SYNTAXES or integrating with existing developer tooling.

Known uses

- Many examples of graphical DSLs developed using the MetaCase tool are given in [24].

- Most BPM tool suites provide a GRAPHICAL CONCRETE SYNTAX for visualizing and modifying a process description. In addition, they usually provide a textual syntax for exchange of the models, often in XML. For instance, BPEL extensions for tools like Eclipse or Netbeans BPEL modelers provide a GRAPHICAL CONCRETE SYNTAX for BPEL, and in addition they provide an export of BPEL in XML syntax.
- Voelter et al. present a graphical DSL for state charts developed using openArchitectureWare [53].

5.4 Alternative: Form-/Table-based Concrete Syntax

Solution

If the models defined via a DSL have the structure of forms or tables, use a form- or table-based concrete syntax for the DSL. Using a form-based syntax means that the DSL user is presented a list of labels and corresponding input fields. Using a table-based syntax means that the DSL user enters information in rows and columns format.

Usually FORM-/TABLE-BASED CONCRETE SYNTAXES come with a respective form or table editor. In some cases only a purely textual input syntax is offered to DSL users (e.g., something akin to a comma-separated values (CSV) format). The form or table is then saved in this format, and later parsed and mapped to the abstract syntax. Sometimes such textual syntaxes are also used as interchange formats.

That is, if one chooses this type of concrete syntax, in most cases the development of a tailored form-/table editor is needed (a mapping from the internal, i.e. in-memory, representation for working with the form/table data to the abstract syntax is always needed). In addition, depending on the use cases of the DSL optional, export and import components for an interchange format are required.

Consequences

There are the following benefits of the FORM-/TABLE-BASED CONCRETE SYNTAX alternative:

- If a FORM-/TABLE-BASED CONCRETE SYNTAX is expressive enough to define DSL models, both, forms and tables, are usually easy to understand.
- Implementing a form or table editor is easy and straightforward. Most GUI frameworks support form or table widgets.
- Textual input formats for forms or tables are often easy to integrate with existing developer tooling. Moreover, form or table edit functionality is usually easy to integrate in graphical editor tools. If both kinds of integrability are needed, it is usually possible to define a simple export format from the editor tool. If properly designed, the mental mapping between a GUI form/table and a textual form/table in an export format is easy.
- Forms and tables are usually easy to communicate to different stakeholders.
- If used inside an integrated tool suite, forms and tables are usually easy to sell. Forms and tables can also serve as input formats from tools end-users are familiar with, such as Microsoft Excel or OpenOffice Calc.

- If it is possible to work on a textual export or interchange format and re-import changes into form-/table-editors, it is often straightforward to implement model evolution or model changes via scripts or simple tools such as grep or search/replace.

- If it is possible to edit the information in the forms/tables via textual export or interchange formats, FORM-/TABLE-BASED CONCRETE SYNTAXES are usually well accepted by developers.

There are the following liabilities of the FORM-/TABLE-BASED CONCRETE SYNTAX alternative:

- FORM-/TABLE-BASED CONCRETE SYNTAXES are limited to problem domains that are suited to be expressed in a form/table format, of course.

- Integrability of form-/table-based solutions is usually good, but can suffer when proprietary interchange formats are used. For instance, the file formats of Excel are not easy to parse and use with tools like diff/merge. However, often simple representations, such as the comma-separated value (CSV) export/import of Excel, can be used to avoid this problem. Another example is the limited integrability, if developers are forced to embed the forms/tables into a predefined interchange format, such as XMI, which contains a lot of other information.

- For the same reasons integrability with tools can suffer, if predefined/external (non-DSL) formats (such as Excel's file format or XMI) are used. In such a case model evolution can suffer, because scripts or tools like grep or search/replace don't work well with these formats.

Known uses

- JetBrains' MPS system [8, 23] is a model-driven DSL toolkit akin to openArchitectureWare [36]. A main difference is that the concrete syntax is based on forms that are filled with the MPS tool. DSLs in MPS are themselves specified using this form-based editor.

- Most BPM tool suites combine a GRAPHICAL CONCRETE SYNTAX for the process description with a FORM-/TABLE-BASED CONCRETE SYNTAX for properties and details of the processes. This design can also be found in many other graphical editors for DSLs.

- JBoss Rules/Drools [39] is a JBoss enterprise framework which provides a platform for the processing of business rules. JBoss business rules are defined using a DSL called Drools rule language (DRL). Different concrete syntaxes for DRL exist, including a table-based syntax for the definition of decision tables.

5.5 Recommendation

DSLs that should, in first place, be presented to end-users or domain experts who occasionally work with the DSL should have a GRAPHICAL CONCRETE SYNTAX. If a GRAPHICAL CONCRETE SYNTAX is required, it is mandatory to build a respective graphical editor.

For DSL containing many technical details, usually a TEXTUAL CONCRETE SYNTAX is more suitable.

For DSLs that should express form-/table-based information, a FORM-/TABLE-BASED CONCRETE SYNTAX should be considered, as this style has many benefits. However, often forms and tables can be excluded because of their limited expressiveness. Nevertheless, for some cases it makes sense to combine the FORM-/TABLE-BASED CONCRETE SYNTAX with other syntax styles, such as GRAPHICAL CONCRETE SYNTAX to the mutual benefit. An example is: To use the GRAPHICAL CONCRETE SYNTAX for presenting the big picture and applying forms for setting detailed properties of individual elements in a DSL program/model.

If a GRAPHICAL CONCRETE SYNTAX as well as a TEXTUAL CONCRETE SYNTAX is required, the variant to generate a visualization of the textual models should be considered first. Only if this is not enough, an additional graphical editor should be developed.

If end-users should define DSL programs, it makes sense to consider offering them a TEXTUAL CONCRETE SYNTAX instead of offering only a graphical editor. In some domains (e.g. if frequent change to DSL models would be needed), this can ease their work significantly, as in many areas the distinction between software users and developers is beginning to disappear [10]. End-user programming research has shown that end-users are willing to learn even complex programming languages, if they see a benefit for daily work (see, e.g., [32]). However, selling a graphical end-user tool is often easier.

6 Architectural Decision: External vs. Embedded DSL

Context

Realizing a concrete syntax and mapping it to an abstract syntax.

Problem

Which style of parsing and interpretation (aka mapping to the abstract syntax) of the DSL's concrete syntax should be chosen?

Forces

- *Influence on concrete syntax:* A concrete syntax can either be built from scratch or on top of a given host language. If a host language is used, syntactic features of the host language can be (re-)used. However, this means that the DSL is dependent on and perhaps limited by the host language's syntax.
- *Relation to host language and/or platforms:* If a specific host language is chosen for interpretation of the DSL, the host language features can be (re-)used in the DSL. However, this also entails dependence on the host language: Usually, if a concrete syntax is built on top of a specific host language, the corresponding language infrastructure (such as interpreter, libraries, etc.) are chosen as the DSL's target platform. This, again, means platform-dependence. For example, if we build an EMBEDDED DSL using a scripting language such as Python, Perl, Ruby, or Tcl, our DSL can directly use the parser and interpreter of the respective host language. The alternative is to define the DSL independent of a host language and use generative techniques (such as model-to-model transformations) to map DSL models/programs to an arbitrary target language and/or platform.
- *Tool support:* Different styles of parsing and interpretation go along with different language engineering tools that can be used or reused to build the DSL's parsing and interpretation functions. Examples of such tools are parser generators and interpreters. However, even if we build

an EMBEDDED DSL and reuse the tools and infrastructure of a particular host language, it is often necessary to specifically adapt different tools as they are not aware of DSL commands (e.g. language editors).

- *Knowledge required by the DSL user:* In the ideal case, the DSL users should only require their domain knowledge to use and understand the DSL. In some cases, however, it is sensible to provide programming language features in the DSL that require a greater learning effort (e.g. loops or conditional branches).
- *Integration of different DSLs:* In some cases, various DSLs developed in one organization must be integrated, for instance in a tool suite. If the various DSLs are developed with different concepts and tools, this task is significantly more complex, of course, than integrating DSLs that were build using a unifying concept and tool set during DSL design.

Alternative Solutions

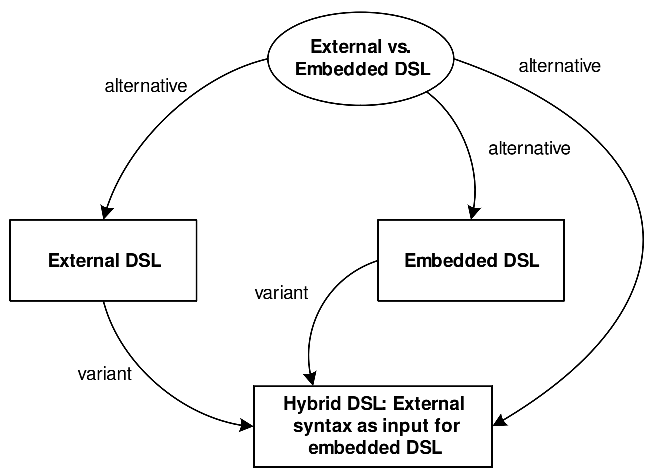


Figure 10: Decision overview: External vs. Embedded DSL

Figure 10 illustrate the alternative solutions for this architectural decision.

6.1 Alternative: External DSL

Solution

Define the DSL models in a syntax that is different and independent to the target language's (or target platform's) syntax (i.e. the environment in which DSL expressions should be executed). Use a special-purpose DSL parser to parse this syntax and map the parsing results to the target language/platform.

An EXTERNAL DSL is defined in a format that is different from the native format of the intended target language(s)/platform(s). Therefore, an EXTERNAL DSL can use all kinds of syntactical elements (independent of any other language) (see also [11, 20, 30]).

Fowler [28] presents a number of techniques for implementing EXTERNAL DSLS in pattern form, such as TREE CONSTRUCTION for creating a syntax tree from source code using a parser, or EMBEDDED TRANSLATION for embedding interpretation code into the grammar.

Consequences

There are the following benefits of the EXTERNAL DSL alternative:

- Any concrete syntax can be chosen. Often multiple syntaxes can be defined rather easily.
- An EXTERNAL DSL is not dependent on a particular host language, but can be mapped to any host language via transformations.
- The DSL is not dependent on a particular platform, but can be mapped to any platform via transformations.
- An EXTERNAL DSL provides a certain level of safety of use because only DSL language elements are accessible in the DSL. Hence, no accidental access to host language elements is possible. This is also a benefit in terms of security: There are no host language features accessible that might entail unknown security threads.
- A great amount of tools exists to support parsing, building graphical editors, transformations, generators, etc.
- The DSL user does not have to be familiar with any host language or platform details.

There are the following liabilities of the EXTERNAL DSL alternative:

- No reuse of syntax knowledge, design, or implementation given by a host language is possible.
- No reuse of host language features in the DSL is possible: Basic functions, such as loops and variables, must be built from scratch. However, basic language feature design is often not as trivial as it seems and can easily be done wrong.
- Even though many tools that support development of EXTERNAL DSLS exist, often substantial additional development efforts are required for developing a parser, a DSL editor, transformations, a generator, and so on. In addition, the existing tools for language design and implementation are not always well integrated.
- An EXTERNAL DSL allows developers to design the DSL's syntax and basic language features in any possible way. Language integration gets difficult, however, if different language concepts or different syntaxes are chosen in different DSLs. That is, in EXTERNAL DSLS it requires discipline and additional effort to develop a number of DSLs in a way that they are easy to integrate – both in a syntactic and semantic sense.

Known uses

- Typical examples of textual EXTERNAL DSLS are DSLs developed using openArchitectureWare's xText framework [36] or Microsofts' OSLO framework [31]. We have already shown an example of XText grammar in Figure 6.
- Many examples of graphical EXTERNAL DSLS developed with the MetaCase tool are given in [24].
- Grammarware [25] is used as a term for the approach that is focusing on grammars and all software artifacts that directly depend on grammars. Many EXTERNAL DSLS are built using grammarware. For instance, Parr presents many DSLs based on the ANTLR parser generator [37].
- A simple version of realizing this alternative has been described in pattern form: The LEXICAL PROCESSING pattern [44] describes how to use lexical translation to realize a DSL. Examples of such EXTERNAL DSLS include numerous DSLs implemented using tools such as sed, awk, Perl, Python, Tcl, and pre-processors such as the C pre-processor. These tools offer rich lexical processing and substitution facilities, such as regular expressions or string manipulation functions. Often simple DSLs can be realized this way in a few lines of code.

6.2 Alternative: Embedded DSL

Solution

Define the DSL models (or programs) in the syntax of a particular host language (i.e. the DSL is defined using the same programming language/platform which is used to execute DSL expressions). Mapping to the abstract syntax of the DSL implicitly happens by interpreting or executing the DSL models (or programs).

An EMBEDDED DSL (also called *internal DSL*) is defined as an extension to an existing programming language and uses the syntactic elements of the underlying (host) language (see also [11, 20, 30]).

Two variants of this architectural alternative have been described before in pattern form:

- The PIGGYBACK pattern [44] describes how to use the capabilities of an existing language as a hosting base for a DSL.
- The LANGUAGE EXTENSION pattern [44] describes how to add new DSL features to an existing host language as a language extension.

Fowler [28] presents a number of techniques for implementing EMBEDDED DSLS (he uses the term "internal DSL") in pattern form. For example, he describes NESTED CLOSURE which is used to express statement sub-elements of a function call by putting them into a closure in an argument, or LITERAL EXTENSION which is used to add methods to program literals.

Consequences

There are the following benefits of the EMBEDDED DSL alternative:

- The proven syntax of the host language can be used.
- For many dynamic languages and scripting languages it is relatively easy to adapt the host language syntax (to a certain extent).
- Code written in the DSL can potentially access and reuse all features of the host language. Reuse of host language features potentially reduces the development effort to create the DSL: The language designer can concentrate on domain-specific language elements instead of realizing basic language features, such as loops or variable substitution. Host languages that are used for building EMBEDDED DSLS are often dynamic languages or scripting languages – because they are flexible and easy to adapt.
- The host language interpreter is often used as the target platform: this reduces the development efforts.
- Tools of the host language, such as editors or debuggers, can be used for the DSL, if the host language syntax is chosen. The creation of a generator or transformations is not required – we only need interpretation code that maps the DSL's concrete syntax to the respective abstract syntax. This interpretation code can be quite trivial to write. All these aspects reduce the development efforts necessary to implement an EMBEDDED DSL.
- Language integration in EMBEDDED DSLS is usually easy because the same syntactic style, same host language, and/or same platform components are used in all DSLs.

There are the following liabilities of the EMBEDDED DSL alternative:

- To a certain extend, the concrete syntax is determined by the respective host language. Moreover, the host language syntax is usually designed for programming, not for the domain of the DSL. However, many host languages, such as scripting and dynamic languages, allow for flexible syntax adaption.
- If host language features are accessible in the DSL, the DSL users must be familiar with both the DSL and the host language – at least to avoid mistakes. The alternative is to tailor the host language in a way that only the desired subset of host language features is accessible in the DSL. That is, any language features that are unsafe or may cause security threats in the DSL are removed from the EMBEDDED DSL. This could for instance be done by customizing the interpreter (e.g. by using a so-called sandbox interpreter) or running a pre-processor on DSL code.
- If the host language interpreter is used as the DSL platform, the solution maybe platform-dependent (depending on the corresponding interpreter). However, interpreters of scripting and dynamic languages are usually implemented for many different operating systems. If the interpreter is not used as the DSL platform, the solution may be platform-independent. However, in this case a dedicated generator and transformations must be developed. In turn, this also means that we cannot benefit from a reduction of corresponding development efforts that is achieved if we use the host language interpreter as our DSL platform.
- In general, host language tools are not aware of tailor-made DSL extensions. For instance, an editor for the host language will not syntax-highlight keywords of the DSL.

Known uses

- Freeze [12] discusses a typical way how to develop EMBEDDED DSLS in Ruby. Like many other DSLs developed in dynamic languages or scripting languages, the host language constructs are used creatively in order to present the DSL code in a way that looks like an external syntax but actually is legal Ruby code. For example, the concrete syntax for a Recipe DSL suggested in [3] (see Figure 11) is parsable by Ruby with a few syntactic tweaks.

- The above mentioned approach is pretty comparable to developing EMBEDDED DSLS in object-oriented, functional languages such as Scala [9].

- Cuadrado and Molina [6] present an approach to support model-driven development using several DSLs implemented in Ruby. In particular, their approach includes four embedded DSLs for Ruby that support basic tasks in model-driven development: a DSL to create language models, a DSL to specify restrictions on models, a model-to-model transformation DSL, and a model-to-code language.

- Frag is a dynamic language that supports embedded use and definition of DSLs. We have already shown a language model in Frag in Figure 7. The same syntactic style that is used to define the language model can also be used to define instances of that model.

```
recipe "PBJ Sandwich"

ingredients "two slices of bread",
            "one heaping tablespoon of peanut butter",
            "one teaspoon of jam"

instructions "spread peanut butter...",
             "spread jam...",
             "place other slice..."

servings 1
prep_time "2 minutes"
```

Figure 11: Example: Ruby concrete syntax for Recipe DSL (from [3])

6.3 Alternative Variant: Hybrid DSL – External Syntax as Input for Embedded DSL

Variant of

- EMBEDDED DSL

- EXTERNAL DSL

Solution

Provide a frontend with an external syntax for an embedded DSL. Examples are providing an external parser for a textual concrete syntax of an embedded DSL, or adding a graphical concrete syntax as an interface for an embedded DSL.

Consequences

There are the following benefits of this variant:

- As in EXTERNAL DSLS, any concrete syntax can be chosen.
- As in EXTERNAL DSLS, accidental access of unsafe or malicious host language features can be prevented.
- Tool support for EXTERNAL DSLS can be used.
- As in EXTERNAL DSLS, the DSL user does not have to be familiar with any host language or platform details.
- As in EMBEDDED DSLS, language integration is easy with regard to the language model and platforms, as they use the same platform for differeent DSLs.

There are the following liabilities of this variant:

- As in EMBEDDED DSLS, we have a strong platform-dependence to the host language platform.
- Even though many tools exist that support developing EXTERNAL DSLS, development of an additional concrete syntax may cause substantial efforts.
- As in EXTERNAL DSLS, language integration at the concrete syntax level can be difficult, as any possible concrete syntax can be chosen which must then be mapped to corresponding platform features/functions.

Known uses

- In Frag [55], DSL development is based on an explicit language model. This language model is implemented using Frag's syntax, and hence it can be changed and instantiated using Frag's syntax as an EMBEDDED DSL. In addition, if an additional external syntax is needed, the DSL specification can be extended with a rule-based parser specification for the external syntax. This way, an additional EXTERNAL DSL can be provided. Hence, Frag supports the creation of HYBRID DSLS.
- Converge [50] provides (like Frag) an approach to combine external and embedded DSL concepts. Converge's approach is to provide DSL blocks that contain DSL code. The DSL code is written in a different syntax than the target language. The DSL syntax is specified using a declarative grammar language.
- Ghosh demonstrates that Scala's parser combinators can be used to realized similar solution for external DSL grammars [14].

6.4 Recommendation

The forces of this architectural decision can be balanced in many possible ways. However, in the literature two main alternatives are proposed [11, 20, 30]: EXTERNAL DSLS and EMBEDDED DSLS.

EXTERNAL DSLS use a parser and interpretation code that is independent of the target language and platform. This means that an arbitrary syntax for the DSL can be chosen, as well as any interpretation concepts, and any type of mappings/transformations to the platform. Hence, this alternative should be chosen, if flexibility and freedom in the design of syntax, interpretation, and transformation is needed.

EMBEDDED DSLS, in contrast, use a given host language as the foundation for implementing the concrete syntax. This host language is often a scripting language or a dynamic language which offers an adaptable syntax. The host language (respectively the corresponding infrastructure components such as an interpreter) is also used to interpret and execute the DSL code. That is, this alternative is platform-dependent on the host language "platform." The main advantage of EMBEDDED DSLS is that the DSL can directly access all features of the host language including syntactic features, libraries, frameworks, or other platform-specific components. Moreover, the tools (e.g. editor, debugger, compiler, or interpreter) available for the host language (or platform) can directly be used when working with the DSL – even if they do not provide direct DSL support such as syntax-highlighting of DSL commands. This alternative should be chosen, if rapid DSL development and evolution is required, or if the host language should serve as a unifying concept and tool set for a number of DSLs to support DSL integration.

A number of hybrid variants of these alternatives exist. One example is a DSL that uses external parsing and embedded interpretation. Choose this variant, if host language features should not be accessible to the DSL user, but the benefits of EMBEDDED DSLS are required for the interpretation of the DSL.

7 Conclusion

The contribution of this paper is twofold: Firstly, we presented three reusable architectural decisions for DSL design that are foundational decisions in DSL projects. Many of the alternatives and variants of these decisions seem to be rather obvious solutions at first glance. However, so far the detailed design trade-offs and forces governing the decision making process in DSL development have not been documented in an exhaustive and comprehensive work. In our experiences, providing this information is an important basic work for building tools that support architectural decision making in the DSL design space. Secondly, as the alternative solutions together with the decisions can be seen as (candidate) patterns, this work also contributes to exploring the overlaps in the (complementary) concepts of architectural decision modeling and software patterns. Patterns can be used to easier populate and maintain a reusable architectural decision model. On the other hand, reusable decision models are useful to explore a design space and mine patterns.

Acknowledgments

We would like to thank our EuroPLoP 2009 shepherd Paris Avgeriou for his constructive and insightful feedback on this paper. This work was partly supported by the European Union FP7 project COMPAS, grant no. 215175. http://www.compas-ict.eu/.

Appendix: On Reusable Architectural Decision Models and Patterns

Software designers and architects have to make many decisions when designing a software system, and, while making these decisions, many competing forces must be balanced. In the software engineering discipline, a number of concepts have been proposed to deal with the problem of how to manage and reuse the knowledge from previous experiences, and how to document and balance the forces of the decisions.

Two important concepts to preserve and teach software engineering knowledge are patterns and architectural decision modeling. *Software patterns* capture reusable design knowledge and expertise that provides proven solutions to recurring software design problems that arise in particular contexts and domains [40]. To systematically explain how to apply a number of patterns in combination, many pattern authors document patterns as part of *pattern languages*. A pattern language defines a collection of patterns in a domain and describes how these patterns can be combined.

More recently, *architectural decision models* [22, 26, 51] have been proposed as a means to document architectural decisions. Decision models capture selected design options with their strengths and weaknesses, as well as justifications for the respective options. Software architecture can be seen as the set of principal design decisions governing a system design [22, 49]. In this view, software architecture engages the complete range of design activities and includes the complete range of participants in the design process [49].

In the design of a software system, the designers and architects encounter numerous decisions, including decisions that deal with organizational matters or business issues, broad design decisions affecting the system as a whole, decision for detailed design issues, and decisions for technological options [57]. A decision qualifies as an *architectural decision*, if it affects the architecture of a system or the role of the architect. That is, the architects of the system regard those decisions as *principal* decisions. The main argument for architectural decision modeling is that such principal decisions capture important knowledge that should not get lost and must be preserved.

Unfortunately, neither pattern languages nor architectural decision models solve all design and documentation problems for reusable design knowledge. For example, most practitioners only know a few patterns, such as the GoF design patterns [13], although the pattern community has documented patterns for many other domains. Hence, extensive upfront education is required to maximize the benefits that can result from using a number of patterns and/or pattern languages. In addition, decision modeling is most often done retrospectively. It is often seen as a "painful" additional responsibility without many gains [18, 51]. Different techniques, text templates, and tool support for decision modeling have been proposed, but failed to become broadly adopted in practice so far [57].

For these reasons, Zimmermann et al. [56, 57] propose reusable architectural decision models. In particular, Zimmermann et al. present a reusable decision model for recurring decisions in SOAs that is based on SOA patterns. A *reusable architectural decision model* is a decision model that is used to guide the architectural decision making activities [57]. Patterns and architectural decision models have many overlaps (for details see [18]). For instance, the approach of Zimmermann et al. uses decision models for pattern selection. The advantage of this approach is that a decision model which is based on patterns does not have to copy the pattern text and hence is easier to create than a self-contained decision model.

Figure 12 shows our concepts for describing architectural decisions in more detail. Corresponding to the elements depicted in Figure 12, we use the following template for describing *reusable architectural decisions*:

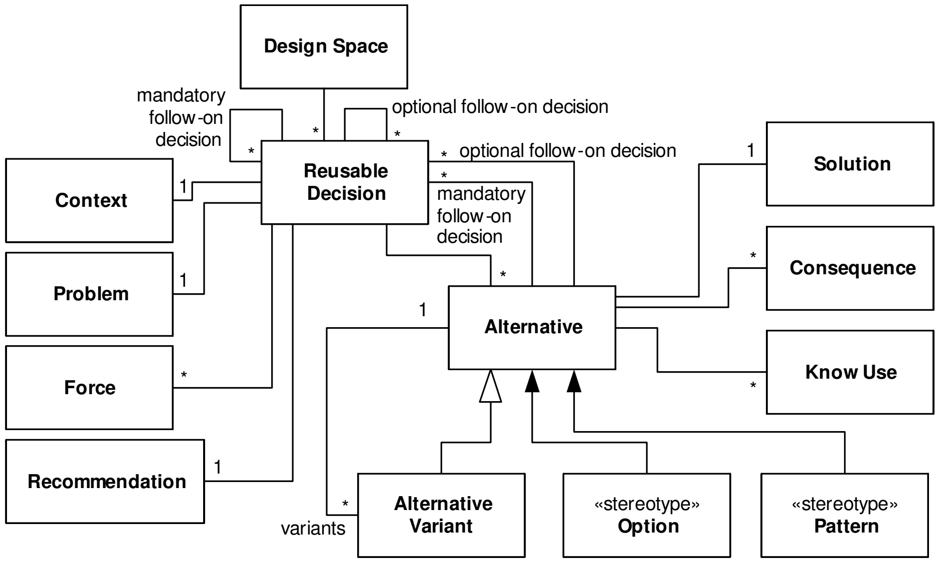


Figure 12: Conceptual Model of Reusable Architectural Decision Model Elements

- *Context:* briefly specifies the context in which the respective decision has to be made.
- *Problem:* outlines the problem that is addressed through a particular decision.
- *Forces:* describes the forces that affect the decision for one of the corresponding alternatives.
- *Alternative Solutions:* provides an overview of the alternative solutions and variants thereof. Following a corresponding overview figure (which shows the different alternatives and their relations), the alternatives are presented in detail.
- *Recommendation:* gives a summary of the different alternatives and provides a recommendation on how to choose which alternative.

For each decision we provide several *alternatives* and/or *alternative variants*. Here, an alternative is a decision option that is disjunct from other alternatives that are available for the same decision. An alternative variant is a variation of a particular decision option (or multiple options). As shown in Figure 12, an alternative can be an *option*. An option is an alternative that is optional. That means, it can either be chosen, or nothing is chosen. As it would be tedious to explicitly model choosing the optional alternative or not choosing it for every optional decision, the *option* concept can be used for marking alternatives that have the implicit alternative of being not chosen. The decisions in this paper have only alternative and alternative variants, but no options. Alternatives and alternative variants are described using the following template:

- *Solution:* provides a detailed description of the respective alternative's solution.
- *Consequences:* describes the benefits and liabilities that are associated with a particular alternative.
- *Known Uses:* illustrates known uses from existing technologies or actual projects as examples for this alternative's solution.

As can be seen, the reusable decision template resembles typical pattern templates. In contrast to typical pattern descriptions, the alternatives of a decision share the same context, problem, and forces. Hence, these decision parts including one of the corresponding alternatives can each be interpreted as one (candidate) pattern. For this reason, following the model in Figure 12, any alternative can be stereotyped as describing a *pattern*.

Compared to other architectural decision templates proposed in the literature (see, e.g., [22, 26, 51, 56, 57]), our decision template is very concise and closer to pattern templates. We have chosen this template because the pattern form is well established for recording reusable design knowledge. Most existing architectural decision templates have their origins rather in recording architectural decisions; of course, then many aspects of the actual design situation must be recorded, too, such as the outcome of a decision or the considerations why a particular alternative has been chosen. We only focus on the "reusable" aspects of the decisions. A second reason is the minor goal of this paper to describe an approach that can serve for pattern mining: Our template can very easily be mapped to a pattern form.

References

[1] J. Bentley. Programming Pearls – Little Languages. *Communications of the ACM*, 29(8), August 1986.

[2] F. Brooks. *The Mythical Man-Month: Essays in Software Engineering – 20th Anniversary Edition*. Addison Wesley Longman, 1995.

[3] J. Buck. Writing domain specific languages. http://weblog.jamisbuck.org/2006/4/20/writing-domain-specific-languages/, 2006.

[4] R. Charette. Why Software Fails. *IEEE Spectrum*, 42(9), September 2005.

[5] J. O. Coplien. *Software Patterns*. SIGS Books, New York, New York, 1996.

[6] J. S. Cuadrado and J. G. Molina. Building Domain-Specific Languages for Model-Driven Development. *IEEE Software*, 24(5), September/October 2007.

[7] K. Czarnecki and U. Eisenecker. *Generative Programming – Methods, Tools, and Applications*. Addison-Wesley, 2000.

[8] S. Dmitriev. Language Oriented Programming: The Next Programming Paradigm. http://www.onboard.jetbrains.com/is1/articles/04/10/lop/, November 2004.

[9] G. Dubochet. On embedding domain-specific languages with user-friendly syntax. In *Proceedings of the 1st ECOOP Workshop on Domain-Specific Program Development*, pages 19–22, Nantes, France, July 2006.

[10] G. Fischer, K. Nakakoji, and Y. Ye. Metadesign: Guidelines for supporting domain experts in software development. *IEEE Software*, 26(5):37–44, 2009.

[11] M. Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? http://martinfowler.com/articles/languageWorkbench.html, June 2005.

[12] J. Freeze. Creating DSLs with Ruby. *Ruby Code & Style*, March 2006.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[14] D. Ghosh. External DSLs made easy with Scala Parser Combinators. http://debasishg.blogspot.com/2008/04/external-dsls-made-easy-with-scala.html, 2008.

[15] Graphviz. http://www.graphviz.org/, 2009.

[16] J. Greenfield and K. Short. *Software Factories: Assembling Applications with Patterns, Frameworks, Models & Tools*. J. Wiley and Sons Ltd., 2004.

[17] A. Haase. Patterns for the definition of programming languages. In *Proceedings of 12th European Conference on Pattern Languages of Programs (EuroPLoP 2007)*, Irsee, Germany, July 2007.

[18] N. Harrison, P. Avgeriou, and U. Zdun. Using patterns to capture architectural decisions. *IEEE Software*, pages 38–45, July/Aug. 2007.

[19] R. Herndon and V. Berzins. The Realizable Benefits of a Language Prototyping Language. *IEEE Transactions on Software Engineering (TSE)*, 14(6), June 1988.

[20] P. Hudak. Building Domain-Specific Embedded Languages. *ACM Computing Surveys*, 28, December 1996.

[21] C. Iacovou and A. Dexter. Surviving IT Project Cancellations. *Communications of the ACM*, 48(4), April 2005.

[22] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *Proceedings of the 5th Working IEE/IFP Conference on Software Architecture, WICSA*, 2005.

[23] JetBrains. Meta programming system - tutorial. http://www.jetbrains.com/mps/docs/tutorial.html, 2009.

[24] S. Kelly and J. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, 2008.

[25] P. Klint, R. Lämmel, and C. Verhoef. Toward an Engineering Discipline for Grammarware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(3), July 2005.

[26] P. Kruchten, P. Lago, and H. Vliet. Building up and reasoning about architectural knowledge. In C. Hofmeister, editor, *QoSA 2006 (Vol. LNCS 4214)*, pages 43–58, 2006.

[27] P. Landin. The Next 700 Programming Languages. *Communications of the ACM (CACM)*, 9(3), March 1966.

[28] DSL Book – Work in Progress. http://martinfowler.com/dslwip/, 2009.

[29] T. Mens and P. V. Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[30] M. Mernik, J. Heering, and A. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4), December 2005.

[31] Microsoft. Microsoft Modeling Platform (code named Oslo). http://msdn.microsoft.com/en-us/library/cc709420.aspx, 2008.

[32] B. A. Nardi. *A small matter of programming: perspectives on end user computing*. MIT Press, 1993.

[33] G. Neumann and M. Strembeck. Design and Implementation of a Flexible RBAC-Service in an Object-Oriented Scripting Language. In *Proc. of the 8th ACM Conference on Computer and Communications Security (CCS)*, November 2001.

[34] OCL 2.0 Specification. available at: http://www.omg.org/technology/documents/formal/ocl.htm, May 2006. Version 2.0, formal/06-05-01, The Object Management Group.

[35] OMG Unified Modeling Language (OMG UML): Superstructure. available at: http://www.omg.org/technology/documents/formal/uml.htm, November 2007. Version 2.1.2, formal/2007-11-02, The Object Management Group.

[36] Open Architecture Ware. openArchitectureWare. http://www.openarchitectureware.org/, 2008.

[37] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, Raleigh, 2007.

[38] Prefuse. http://www.prefuse.org/, 2009.

[39] M. Proctor, M. Neale, M. Frandsen, S. Griffith, E. Tirelli, F. Meyer, and K. Verlaenen. JBoss Rules User Guide – Drools Documentation. Version 4.0.7, available at: http://jboss.org/drools/documentation.html, May 2008.

[40] D. Schmidt and F. Buschmann. Patterns, frameworks, and middleware: Their synergistic relationships. In *25th International Conference on Software Engineering*, pages 694–704, May 2003.

[41] D. C. Schmidt. Model-Driven Engineering – Guest Editor's Introduction. *Computer*, 39(2), February 2006.

[42] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5), 2003.

[43] S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5), 2003.

[44] D. Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1), February 2001.

[45] T. Stahl and M. Völter. *Model-Driven Software Development*. John Wiley & Sons, 2006.

[46] G. Stepanek. *Software Project Secrets: Why Software Projects Fail*. Apress, 2005.

[47] M. Strembeck and G. Neumann. An Integrated Approach to Engineer and Enforce Context Constraints in RBAC Environments. *ACM Transactions on Information and System Security (TISSEC)*, 7(3), August 2004.

[48] M. Strembeck and U. Zdun. An Approach for the Systematic Development of Domain-Specific Languages. *Software: Practice and Experience (SP&E)*, 39(15), October 2009.

[49] R. N. Taylor and A. van der Hoek. Software design and architecture: The once and future focus of software engineering. *Future of Software Engineering (FOSE '07)*, pages 226–243, 2007.

[50] L. Tratt. Domain specific language implementation via compile-time meta-programming. *TOPLAS*, 30(6):1–40, 2008.

[51] J. Tyree and A. Ackerman. Architecture decisions: Demystifying architecture. *IEEE Software*, 22(19–27), 2005.

[52] M. Voelter. A family of languages for architecture description. In *Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling*, Nashville, TN, USA, October 2008.

[53] M. Voelter, B. Kolb, S. Efftinge, and A. Haase. From Front End To Code - MDSD in Practice. http://www.eclipse.org/articles/Article-FromFrontendToCode-MDSDInPractice/article.html, 2006.

[54] M. P. Ward. Language-Oriented Programming. *Software - Concepts and Tools*, 15(4), 1994.

[55] U. Zdun. Frag. http://frag.sourceforge.net/, 2009.

[56] O. Zimmermann, T. Gschwind, J. Kuester, F. Leymann, and N. Schuster. Reusable architectural decision models for enterprise application development. In S. Overhage and C. Szyperski, editors, *Quality of Software Architecture (QoSA) 2007*, Lecture Notes in Computer Science, Boston, USA, July 2007. Springer-Verlag Berlin Heidelberg.

[57] O. Zimmermann, U. Zdun, T. Gschwind, and F. Leymann. Combining pattern languages and reusable architectural decision models into a comprehensive and comprehensible design method. In *WICSA '08: Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, pages 157–166, Washington, DC, USA, 2008.