Chain of Builders: A Pattern of Variable Syntax Processing for Internal DSLs

Stefan Sobernig
WU Vienna
Austria
stefan.sobernig@wu.ac.at

ABSTRACT

CHAIN OF BUILDERS is a language-implementation pattern at the centre of realising variable textual syntaxes for internal domain-specific languages (DSL). An internal DSL is built on top of a general-purpose software language (GPL; Java) and uses the GPL infrastructure for processing and for enacting DSL scripts. A DSL is said to be variable when it allows for deriving a family of DSL variants (a.k.a. a language-product line), varying at the levels of abstract syntax, concrete syntax, semantics, and execution (e.g., interpretation or generation). CHAIN OF BUILDERS combines the CHAIN OF RESPONSIBILITY and EXPRESSION BUILDER patterns. With this, the paper adds to known pattern languages for DSL development (e.g., Fowler's). Its known use has been in development systems for internal DSLs such as DjDSL, but also applies to applications processing mixed syntaxes (XML and JSON).

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; **Software design techniques**.

KEYWORDS

domain-specific language, internal DSL, concrete syntax, variability, language-product line, language family, Java

ACM Reference Format:
Stefan Sobernig. 2019. Chain of Builders: A Pattern of Variable Syntax Processing for Internal DSLs. In *24th European Conference on Pattern Languages of Programs (EuroPLoP '19), July 3–7, 2019, Irsee, Germany*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3361149.3361179

1 INTRODUCTION

A domain-specific software language (DSL) is a software language that is tailored for describing, prescribing, or implementing software systems in a selected domain [10, 11, 21]. As a software language, a DSL comprises an abstract syntax, one or more concrete syntaxes, structural and behavioural context conditions, semantics definitions, and behavioural definitions based on a target platform. The concrete syntax refers to the well-defined representation of a DSL script (model) as text or visual. The abstract syntax means a well-defined representation of a DSL script (model) that abstracts from details specific to the textual or visual representation, to better serve some post-processing task on the DSL script (model). When a DSL is developed as an extension of a general-purpose software language (GPL) and when the DSL uses the GPL infrastructure for processing and for enacting DSL scripts, the DSL is an *internal DSL* and the GPL becomes the DSL's host language [6, 8, 13].

Software patterns play an important role as implementation techniques for internal DSLs, including the implementation of variable DSLs. A variable (incl. extensible) DSL has the ability to vary at the levels of abstract syntax, concrete syntaxes, semantics, and enactment in a systematic and efficient manner. Developing variable DSL shifts emphasis from developing and analysing a single DSL to developing and analysing reusable development artefacts for a family of DSL. Patterns describe proven practises and techniques to design and to implement variability for a DSL. This is mainly because patterns are readily applicable to implementations in (object-oriented) host languages and because they genuinely document ways of implementing runtime variability using proven solutions [4]. The latter is key when implementing internal DSL that can be composed with the host language, other DSLs, or DSL extensions in a disciplined manner. Relevant software patterns and pattern languages describe recurring problems and solutions for architecting a DSL (architectural patterns), for designing the main DSL components and their interactions (language and DSL patterns), and for implementing the internals of the DSL components (design patterns and idioms).

Pattern-based approaches apply to and are equally relevant to developing external DSL [6] and their internals. However, their role for developing internal DSL based on and embedding with a host language, render them more prominent in design-decision making for internal DSL. The following patterns are relevant for architecting, designing, and implementing composable internal DSL for different language-composition scenarios: extension, unification, extension composition, and self-extension [5]. In addition, a pattern-based approach paves the way to developing and to maintaining a hybrid DSL. A hybrid DSL combines the characteristics of an internal and an external DSL, e.g., an internal and external concrete syntax on top of a shared abstract syntax.

The contribution of this paper is to document an additional pattern of advanced syntax processing: CHAIN OF BUILDERS. This pattern is related to existing patterns and design decisions on engineering DSLs. The concerns of syntax processing in DSL relate to patterns from different pattern collections and pattern languages [2]. These include architectural design patterns (e.g., EXPLICIT INTERFACE), software-language design patterns (e.g., EXPRESSION BUILDER [6] and MESSAGE REDIRECTOR [22]), general OO design

EuroPLoP '19, July 3–7, 2019, Irsee, Germany

ACM ISBN 978-1-4503-6206-1/19/07…$15.00
https://doi.org/10.1145/3361149.3361179

patterns (e.g., CHAIN OF RESPONSIBILITY [7]), and implementation-level ones (e.g., abstract-syntax representation patterns [14]).

With this, this paper primarily addresses software developers who want to create a DSL. I imagine the readers of the actual pattern description in Section 3 to be familiar with the basic tenets of developing a DSL [6]. The surrounding text should also be useful to developers wondering what a DSL actually is and why patterns of syntax processing are worth considering beyond their narrow use to implement a DSL. The running examples and code listings are available from a supplemental Web site.[1]

2 BACKGROUND

The concrete syntax refers to the representation of a script, program, or model as text or visual. Syntax processing means processing such a script, program, or model according to a well-defined syntax definition into another representation. This other representation abstracts from representation details as text or visual. This abstract-syntax representation is to better serve a certain processing or tooling step on the script, program, or model. Syntax processing and parsing are very important for applications other than software-language engineering, including application generators, data processing, and application-level network and serialisation protocols.

To accomplish syntax-processing tasks in these fields, it is not necessary to learn and to master methods, techniques, and tools for external-syntax processing (e.g., grammar-based parsing); at least not early in your project. Processing internal syntaxes is about working in your regular, general-purpose language environment (Java) for this purpose.

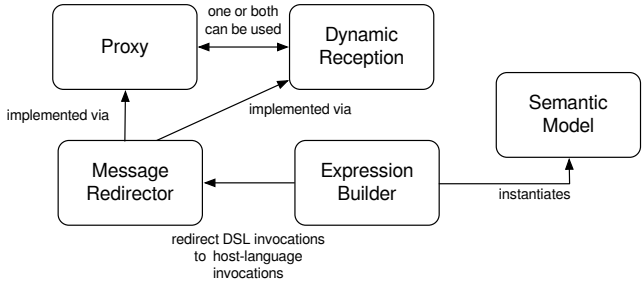

Figure 1: An overview of the pattern relationships specific to the EXPRESSION BUILDER pattern. See Table 1 for brief pattern descriptions.

2.1 Syntax Processing: A Pattern Point of View

Fowler [6] documents established and proven practises of processing DSL scripts, written using an internal syntax, into internal representations (abstract-syntax graphs, model instances) in terms of special-purpose BUILDERS [7, p. 97], e.g., an EXPRESSION BUILDER (see Figure 1). A BUILDER separates the logic to build a complex of objects from the objects themselves. This is particularly needed when different variants of this building logic should be applicable. In DSL syntax processing, different BUILDERS turn out useful: EXPRESSION BUILDER and CONSTRUCTION BUILDER.

An EXPRESSION BUILDER separates syntax processing from instantiating a language model (i.e., the primary abstract syntax of a DSL). See Figure 3 for an exemplary language model. Using an EXPRESSION BUILDER does not only separate the two concerns properly, but also provides for providing alternative builders and, hence, alternative syntaxes in front of one abstract syntax (language model). Alternatively, an EXPRESSION BUILDER allows for providing different language-model backends (incl. variants of a language model, or different language-model implementations) for one frontend syntax.

In what follows, the example of Ansible playbooks is adopted (see Figure 3). In Ansible, a playbook represents a deployment and maintenance descriptor (script) for orchestrating Ansible (distributed) system inventories. A playbook consists of one or several *plays*, each linking a list of *tasks* to a group of inventoried systems or system groups called *hosts* ("webservers", "databases" in Listing 4).

Table 1: Thumbnail descriptions of relevant patterns for syntax processing for *internal DSL syntaxes*. See Figure 1 for an overview of their relationships.

Pattern	Problem	Solution
BUILDER [7, p. 97]	Building a structure from different parts (objects) by connecting them is not trivial (e.g., it depends on the data types of the various parts). In addition, different structures (e.g., representation) should be built from the same set of parts.	Separate the building logic of the structure (e.g., an object graph) from the parts under composition, into a dedicated builder entity.
EXPRESSION BUILDER [6, pp. 343]	The behaviour to instantiate a language model and to process a DSL script are implemented by the same (model) classes.	Separate instantiation and DSL syntax processing into separate, but closely linked (builder vs. model) objects.
SEMANTIC MODEL [6, pp. 159], a.k.a. language model, domain model	A primary representation of a DSL script is derived from the concrete syntax (concrete-syntax graph or tree), which is not necessarily suited for processing the DSL script.	Provide an abstracted (in-memory) representation of a DSL script that is very close to the purpose of processing of the DSL script (its application domain).
DYNAMIC RECEPTION [6, pp. 427]	The receiver of an invocation request (e.g., a specific builder) does not provide a method implementation to process the DSL invocation	Trap and handle (DSL) invocations to builder objects without having defined corresponding methods using built-in message interception or redirection techniques.
MESSAGE REDIRECTOR [22]	(DSL) Clients requesting a certain behaviour (e.g., by sending messages) and (host-language) providers offering the behaviour implementation are not (meant to be) known directly to each other (e.g., to realise dynamic method dispatch).	Provide for redirecting (DSL) invocations to host-language objects implementing the invocation behaviour.
PROXY [7, pp. 207]	The receiver of a message (e.g., a specific builder) does not provide a method implementation to process the message; or the actual provider is not available in a given invocation context.	Provide a placeholder object for another one to manage access to it.

[1] https://github.com/mrcalvin/cob-ansible

```
play {
  hosts       webservers
  remote_user admin
  task {
    name    "is webserver running?"
    service {
      name  http
      state started}}}
play {
  hosts databases
  remote_user admin
  task {
    name "is postgresql at the latest
          version?"
    yum  {
      name  postgresql
      state latest}}}
```

```
play(
  hosts("webservers"),
  remote_user("admin"),
  task(
    name("is webserver running?"),
    service(
      name("http"),
      state("started")))),
play(
  hosts("databases"),
  remote_user("admin"),
  task(
    name("is postgresql at the latest
          version?"),
    yum(
      name("postgresql"),
      state("latest")
      )))
```

```
PlaybookBuilder
.Playbook(/*... */)
  .play(/*... */)
    .task("run some command")
      .shell("/usr/bin/some")
    .task("run another command")
      .shell("/usr/bin/another",
            Expect.Shell())
      .response("OK")
        .body("send_user done")
      .response("NOTOK")
        .body("send_user failed;
              abort")
/*... */
```

Listing 2: Exemplary Ansible playbook syntax styles: using a LITERAL LIST on the left, using NESTED FUNCTIONS in the middle, and (Java) METHOD CHAINING on the right.

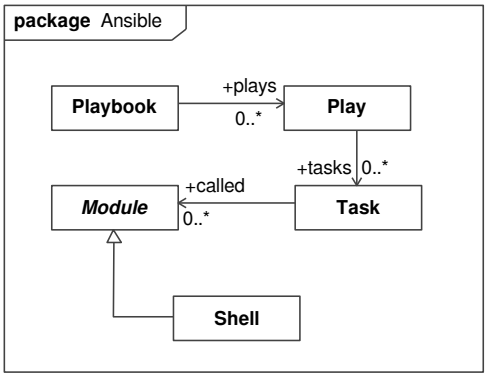

Figure 3: An exemplary language model (a.k.a. SEMANTIC MODEL) for Ansible playbooks.

A task itself defines a call to an Ansible module, such as *service* to check the deployment status of a service application or *yum*, a package manager, to update installations in a single batch on an array of systems. Figure 3 exemplifies a *shell* module, which can be used to manage and to script a shell session on a remote system.

There are two frequently adopted representation choices for DSL scripts [8, pp. 318]:

- DSL scripts are represented as host-language collection data, such as lists and maps. Listing 2 (left) gives the example of an Ansible playbook represented as a list of lists.
- DSL scripts are represented as a host-language program. Listing 2 (middle) shows a representation of the same Ansible playbook as a host-language script using nested function calls to model the relationships between playbook items. Listing 2 (right) shows another common syntax flavour using (Java) METHOD CHAINING.

Whatever the representation style chosen, the EXPRESSION BUILDER itself must map DSL syntax elements or invocations to the host-language invocations that instantiate the corresponding elements from the underlying language model. For invocations, mappings between different invocation abstractions may be applicable. In

```
public class PlaybookBuilder {
  private Playbook book;

  /* context variable */
  private Play currentPlay;

  public static PlaybookBuilder Playbook() {
    return new PlaybookBuilder();
  }

  public PlaybookBuilder() {
    /* eager construction */
    book = new Playbook();
  }

  public PlaybookBuilder play() {
    /* eager construction */
    currentPlay = new Play();
    book.add(currentPlay);
    return this;
  }

  public PlaybookBuilder task(String name) {
    if (currentPlay != null) {
      currentPlay.add(new Task(name));
    }
    return this;
  }

  public PlaybookBuilder task() {
    return task("");
  }

  public Playbook get() {
    return book;
  }
}
```

Listing 4: An exemplary (Java) implementation using a single EXPRESSION BUILDER `PlaybookBuilder` and METHOD CHAINING. The instantiation of the language model (from Figure 3) is obtained eagerly (i.e., upon instantiation of the builder or calling any chained method).

case of the internal DSL being realised based on messages (method calls) exchanged between objects (around an object-oriented or "fluent" API), a mapping is established between DSL messages and

host-language messages as well as method implementations. Different invocation abstractions can also be supported depending on the available host-language concepts. For example, intermediaries such as commands in a COMMAND LANGUAGE [22] are well suited as an abstraction.

A syntax definition can be embodied by a single BUILDER in front of a single language model. For example, `PlaybookBuilder` in Listing 4 implements all syntax-handling methods (`play`, `task`) responsible for constructing all entities from the language model. However, the responsibility of syntax processing for a single syntax definition, in front of a given language model, can also be distributed across and implemented by multiple orchestrated BUILDERS. This serves for realising separation of concerns in syntax processing, but also to provide for planned syntax extensions. Not every language-model element, such as `Module` in Figure 3, is necessarily visible at the concrete syntax-level. Therefore, there might not be a corresponding BUILDER.

Separation and composition of syntax processing. When different BUILDERS are provided to transform an input stream into different parts of the language model (e.g., sub-trees or branches of a syntax tree), these child BUILDERS form a COMPOSITE. For example, the responsibilities of the single `PlaybookBuilder` in Listing 4 could be distributed among three distinct ones, each responsible for one language-model entity: `PlaybookBuilder` for `Playbook`, `PlayBuilder` for `Play`,`TaskBuilder` for `Task`, and so forth. Each BUILDER then implements the syntax handlers (methods in METHOD CHAINING) for its child BUILDERS.

The benefit is that the dependency between the BUILDERS are aligned with the syntax dependencies [6, Sections 32.1 and 32.4]. More importantly, having separate builders allows for diverging from the construction procedure as otherwise dictated by the input stream (e.g., a sequence of DSL invocations). This can take the form of re-arranging the order of retrieving the BUILDERS' results; or, by postponing construction (lazy vs. eager acquisition). In the latter case, BUILDERS act as PLACEHOLDERS [17]. A PLACEHOLDER is a kind of PROXY that indirects and defers messages sent to objects under construction, e.g., to control for cyclic dependencies during instantiation of cyclic object graphs.

The advantages of such a decomposition into distinct builders are countered by the complexity of managing the relationships (part-of, and call dependencies) between the builders. For example, in METHOD CHAINING certain methods (`play`) must be implemented for the scope of more than one BUILDER. This is because the sequence of method calls (and the respective BUILDER objects returned from each call) does not correspond to the intended hierarchy of the syntax (e.g., a play *consisting of* tasks). A COMPOSITE can help manage the part-of relationships, and help propagate invocations between the elements of a composite.

Preplanned and bound syntax extensions. In a such a WHOLE-PART [16] structure of BUILDERS, a parent BUILDER can be made to support alternative child BUILDERS in terms of a STRATEGY [7, pp. 315]. This way, depending on evaluating conditions in the client or the parent BUILDER, at runtime, different alternative BUILDER implementations become effectuated. Consider the playbook example in Listing 2, on the right. The `shell` invocation can be made to select from different shell implementations (e.g., Ansible built-in, an Expect shell) to execute the submitted commands. This choice is made based on an extra argument to `shell` (`Expect.Shell()`), or by its bare absence. This way, known syntax elements (e.g., chaining methods) can be bound to different processor implementations. Each alternative implementation can then interpret the syntax stream, e.g., by instantiating the language model differently. This way, *preplanned* extension points can be implanted into syntax processing.

The downside is that these extension points (e.g., the overall STRATEGY interface to be implemented by alternative child BUILDERS) must be defined early in a DSL project. That is, in the playbook example in Listing 2 (right), when the `shell` method is provided (and the corresponding `ShellBuilder`), the available shell-specific syntax (`response`, `body`) must be decided on. On the same page, the one interface shared between builders prevents each alternative from implementing a deviating (i.e., additional or fewer) syntax fragments. This is prominently the case in host languages with typing restrictions. As a consequence, any syntax extension using STRATEGY is *bound* by a predefined interface.

2.2 DSL composition

Erdweg et al. [5] identify and describe four basic types of unanticipated software-language composition, i.e., composition *without preplanning*: language extension, language unification, extension composition, and self-extension.

- *Extension*: Define and apply units of extensions to a base DSL in an incremental and modular manner.
- *Unification*: Form a unified DSL out of two or more composed DSL; in particular when the composed ones have *not* been developed with composition in mind (no preplanning).
- *Extension composition*: The ability of a DSL development system to, first, compose two or more DSL extensions before enabling them for the base DSL.
- *Self-extension*: Extend a DSL by defining and by applying extensions from within a DSL program or model itself, in an embedded manner.

Patterns for designing and supporting variable syntax processing are equally relevant for all four types of composition. Each composition type, however, presents unique challenges to syntax processing.

3 PATTERN DESCRIPTION: CHAIN OF BUILDERS

You want your DSL to remain open for extending its basic syntax, in an unforeseeable manner, along with an extensible language model and an extensible language runtime (behind the scenes). Or, your DSL implements an initial, minimal but viable language as a first step towards a more complete language definition and implementation to be continued over multiple iterations. That is, you develop your DSL-based software system in a *reactive* (rather than in a proactive) manner. You consider a design using multiple EXPRESSION BUILDERS realising the internal DSL syntax. The separated language model (SEMANTIC MODEL) is open for extension.

You considered using *dynamic parametrisation* [4] for your BUILDER objects to encode different syntax variants. In dynamic parametrisation, per-object flags mark syntax increments to become enabled

or to become disabled at construction time of the object (e.g., by passing them as arguments to the constructor calls). When syntax processing is implemented using METHOD CHAINING, the flags are used at the sub-method level to implement a conditional control flow. Each resulting control-flow walk realises one syntax variant. Modifying an existing syntax variant, or adding a new one, requires you to revisit and to modify the existing syntax-handling method implementations (e.g., by adding a new branch to the control flow). The changing (growing) space of syntax variants results in more and more complex control-flow condition expressions, that are difficult to maintain. This is because the control-flow implementation is scattered across multiple method implementations and across multiple (partial) EXPRESSION BUILDERS.

> **The EXPRESSION BUILDER must be open to extension, in a stepwise manner and without the need for preplanning. Client code to the DSL subsystem, as well as the base EXPRESSION BUILDER must not be affected by the unplanned syntax extensions.**

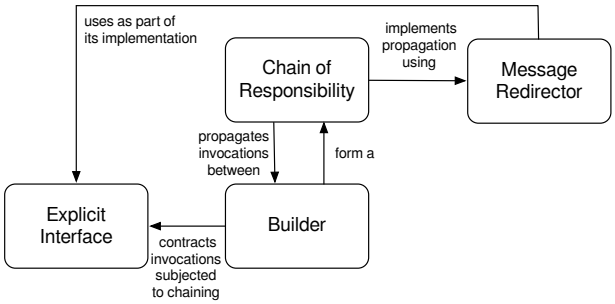

Figure 5: Processing of an internal syntax can be rendered extensible by a structure of BUILDERS that operate as a CHAIN OF RESPONSIBILITY. Each BUILDER is *responsible* for building up the different portions of the (now combined) language model (SEMANTIC MODEL). Refer to Table 2 for brief pattern descriptions.

You face the challenge to add, to modify, or to remove syntax-processing capabilities from an EXPRESSION BUILDER without requiring modifications to interfaces or to implementations, both of client code submitting DSL expressions (scripts) or of the existing EXPRESSION BUILDER itself.

A solution must be applicable to different ways of representing invocations (clauses) in the internal DSL syntax in terms of the EXPRESSION BUILDER implementation. METHOD CHAINING directly maps DSL invocations to method calls in the host language. NESTED FUNCTIONS [6, pp. 357] assumes a procedure or function abstraction in the host language; or methods. Collections like lists and maps are another implementation device for internal DSL syntaxes to be supported (LITERAL LIST, LITERAL MAP; [6, pp. 417]). This list of implementation techniques can grow (closures, initialisers).

Besides, the internal syntax subject to extension can be implemented using multiple, orchestrated EXPRESSION BUILDERS rather than a single one. This is to separate different syntax-processing concerns into different builders. Such a distributed syntax implementation forms a WHOLE-PART structure [3]. A top-level EXPRESSION BUILDER aggregates a number of lower-level EXPRESSION BUILDERS, responsible for different syntax sections. Direct access to these constituent builders is not possible. An unplanned syntax extension then translates into extending such a WHOLE-PART structure of several builders, without having full access to part builders, or their internals.

To support different language-composition techniques (e.g., extension, unification, extension composition), a solution must be able to activate (to deactivate) two or more syntax extensions at a time. Each syntax extension must be given a chance to handle a DSL invocation, or to pass it along, or to drop it, without being aware of any other extensions being present or absent. This conflicts with the requirement to establish basic guarantees that DSL invocations actually become processed (by at least one syntax-processing extensions); or that there is no unintended passing-along or dropping.

Table 2: Thumbnail descriptions of relevant patterns for internal DSL extension. See also Figure 5.

Pattern	Problem	Solution
CHAIN OF RESPONSIBILITY [7, pp. 223]	The client sending a message does not know the actual provider (e.g., the responsible builder) of the requested behaviour.	Give more than one (builder) object the chance to act as a provider of syntax-processing behaviour.
EXPRESSION BUILDER [6, pp. 343]	The behaviour to instantiate a language model and to process a DSL script are implemented by the same (model) classes.	Separate instantiation and DSL syntax processing into separate, but closely linked (builder vs. model) objects; but also into separate builders for each syntax variant.
MESSAGE REDIRECTOR [22]	How to represent DSL invocations and how to implement their propagation between chained BUILDERS?	Provide for redirecting (DSL) invocations from one builder object to another by a built-in or a custom redirection mechanism (e.g., DYNAMIC RECEPTION or a PROXY).
EXPLICIT INTERFACE [1]	Allowing direct and full access to your BUILDER implementation makes the client components dependent on one implementation and side effects internal to this BUILDER.	To avoid coupling to one BUILDER and its internals, provide a distinct BUILDER interface effectively shielding client components from the BUILDER implementation. The BUILDER implementation realising the interface can be changed (at runtime), also as part of a MESSAGE REDIRECTOR.

Therefore:

> **Provide for more than one EXPRESSION BUILDER to process a DSL syntax element or DSL invocation. Render each builder responsible for handling a syntax extension. Form a chain of builders and pass the syntax-processing requests along this chain. Each builder, in its turn, may decide *independently* from other builders (a) to handle the request, (b) to forward the request (as-is or in a modified manner), or (c) to drop it entirely.**

In such a CHAIN OF BUILDERS, more than one BUILDER may handle a (DSL) processing request sent by a client. To the client, the BUILDER ultimately responsible is not known beforehand. Each refining BUILDER acts as a potential MESSAGE REDIRECTOR (see Figure 5). When the chain can be modified during runtime, the location of processing is not even known to the chained BUILDERS themselves.

The class diagram in Figure 6 shows one possible structural overview of a CHAIN OF RESPONSIBILITY between BUILDERS (on the right). An abstracted interaction protocol between chained BUILDER objects is depicted on the left. The challenge is to design and to implement a chaining protocol that matches the requirements (orthogonality between builders etc.). The key idea is to have a refining builder receive and, conditionally, forward DSL invocations to the base builder, or any other refining builder in-between, for that matter (see Figure 5).

From a bird's eye perspective, a BUILDER can maintain a forward reference to a successor BUILDER. These forward references can be realised as explicit and named references between BUILDER objects, or by encoding the references as more generic (language-level) relationships between BUILDERS (e.g., mixin relationship, or the composition of Java interfaces). Therefore, over several forward references, BUILDERS form a chain of succeeding BUILDERS that ends with a terminal BUILDER. The terminal BUILDER acts as the base EXPRESSION BUILDER (see Figure 6, on the left). Maintaining backward references to predecessors might also be necessary (e.g., to enable a kind of call forwarding with changing self-references).

Once references between BUILDERS have been established, DSL invocations must be propagated between predecessor and successor `builders`, if needed. The propagation logic establishes whether to process a DSL script (or fragments of it) locally, with the currently responsible `ConcreteBuilder`, or to forward the script to the successor, if any. This generic decision condition is represented by the `opt` fragment's guard of Figure 6 (on the left). This propagation logic can be implemented explicitly, by one BUILDER method forwarding a call to the respective successor builder, or implicitly, by reusing language-level mechanisms such as calls along a super-reference or an *unknown* mechanism.

It is recommended practise to provide a common anchor for all BUILDER implementations, e.g., an ABSTRACT SUPERCLASS [] (see also Figure 5). Such a common anchor provides both for unifying and for contracting a common method signature to DSL clients the signature interface of all BUILDERS towards DSL clients. For one, it may offer a `get` method to obtain the instantiated language model corresponding to a DSL script. In addition, a common anchor can also help reuse implementation details between BUILDERS (e.g., accessors and mutators to language-model elements during the construction procedure).

Applying a CHAIN OF BUILDERS, from a DSL client's perspective, does no guarantee ultimate or complete processing of the input stream. A valid syntax element may fall off the end of the chain, if propagation is not handled correctly between two chained BUILDERS. Also, assigning blame to the responsible BUILDER for a syntax fragment becomes more difficult. Propagation of DSL invocations in a CHAIN OF BUILDERS, on the one hand, and between BUILDERS in a WHOLE-PART structure must also be accommodated. The former allows for (reactive) syntax extensions while the latter helps organise more complex syntax extensions internally. One may also consider using a PROPERTY LIST [15] at the level of the language model to facilitate extension by a CHAIN OF BUILDERS.

Sample Code. Consider a single BUILDER for Ansible playbooks in Listing 7: `MyPlaybookBuilder`. This BUILDER implements a variable DSL syntax using METHOD CHAINING. It is variable because one can request a BUILDER variant supporting Ansible's *when* clauses, another variant without them. A *when* clause an Ansible marks a certain step (task) as conditionally executable. This is an example of DSL extension. Figure 8 documents one possible design and implementation of the variable internal syntax for this two-member DSL family. In Java, a CHAIN OF BUILDERS can be implemented using interfaces with *default methods* [9, Chapter 9]. A default method provides a default implementation of a method for any class that implements the interface, without overriding the method. A concrete BUILDER (out of two possible ones) is implemented by a composition class *MyPlaybookBuilder*; this materialises one product out of two possible ones for this DSL family. As a mere unit of composition, the class pulls together the different assets. Namely, it implements one or more Java interfaces (`IChainableBase`, `IChainableConditions`) that act as the assets of the small DSL family. Each asset interface implements one or more chainable methods for METHOD CHAINING. The interface `IChainableBase` provides for the methods common to all BUILDER variants: `play()` and `task()`. The extension interface `IChainableConditions` adds `when()`. By implementing either of these interfaces, the composition class *MyPlaybookBuilder* enables either the base set of methods, or the extended one.

Listing 9 depicts the compact composition class (on the top, in its entirety). The sole purpose of this class is (a) to implement the interface providing the chainable methods (line 3, `IChainableConditions`) and (b) to implement a detail required by the chainable methods, internally (lines 6–9): The `getBuilder` method implementation is required from within the chainable methods to be able to return the given BUILDER instance to continue the METHOD CHAINING. The latter is a mere consequence of Java's default methods which do not provide for the typed self-context (`this` cannot be used from therein).

To implement the BUILDER behaviour (e.g., testing context variables such as the most current play or task), the chainable interfaces require a minimal getter and setter interface around a given BUILDER instance (`setCurrent()`/ `getCurrent()`, `set()`/ `get()`; see Figure 8). To share the implementation of these required methods between, and so to minimise, the composition classes, they are provided by the ABSTRACT SUPERCLASS `PlaybookBuilder`. This completes the idea of a BUILDER composition class, that combines the chainable methods from a set of interfaces and that pulls the methods required by chainable methods from the abstract superclass.

Listing 9 also showcases a Java default method implementing `when()` (at the bottom, lines 7–18). Its header section checks for the valid execution context (there must be a play, and at least one task) and then proceeds to extract the most recently added task to set the execution constrain (not shown). Line 17 shows the required `return` statement, to have METHOD CHAINING continue with another chained method. `getBuilder` is a required interface of this interface, in essence compensating for the absence of `this` in a default

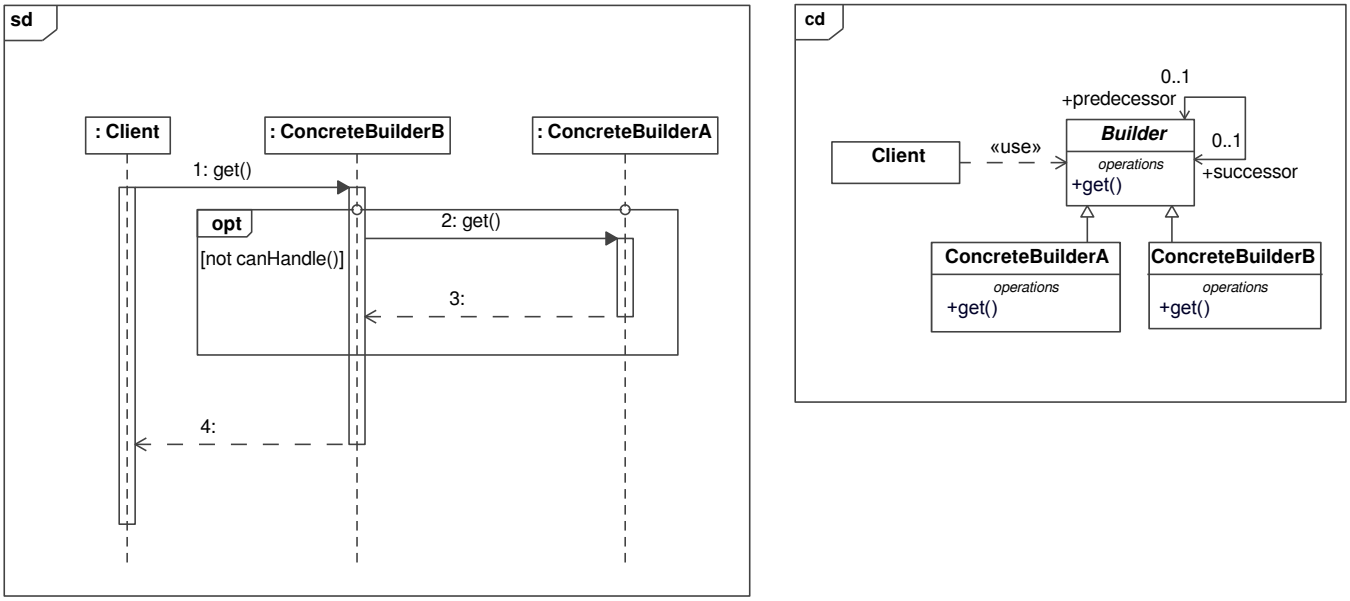

Figure 6: Structural and behavioural overview of a CHAIN OF RESPONSIBILITY between BUILDERS. Adapted from [7, pp. 223].

```
MyPlaybookBuilder.Playbook()
.play()
  .task("run some command")
  .task("run another command");
```

```
MyPlaybookBuilder.Playbook()
.play()
  .task("run some command")
    .when("statusCode is failed")
  .task("run another command")
    .when("statusCode is succeeded");
```

Listing 7: An Ansible playbook DSL family, one variant w/o *when* clauses (top) and one variant w/ *when* clauses (bottom).

method. The composition class, when declaring the implementation of this required interface, also provides for an argument of the type parameter `A`). This way, METHOD CHAINING is guaranteed to operate on the most concrete type level from the perspective of the interface composition (`MyPlaybookBuilder` itself).

To sum up, in this Java implementation variant, the CHAIN OF BUILDERS manifests in terms of an interface composition owned by a BUILDER class (i.e., the product derived from a DSL family), or alternatively by establishing generalisation/ specialisation relationships between special-purpose interfaces then implemented by this class. The chainable methods in METHOD CHAINING are implemented as default methods by these interfaces (i.e., the assets of the DSL family). In this example, a `MyPlaybookBuilder` implementing `IChainableBase` will only support `play` and `task` calls (see Listing 7, on the top). A `MyPlaybookBuilder` implementing `IChainableBase` will additionally allow for `when` calls for a given task (see Listing 7, at the bottom).

Known Uses.

- Fowler [6] points at composing internal DSLs. The motivation of (internal) DSL composition is glimpsed at in [6, Section 6.9]. To avoid bloating an internal DSL up (in terms of additional expressiveness sought), composing two or more smaller, and possibly independent SEMANTIC MODELS (i.e., DSL unification) is hinted at. At the syntax-level, mixing (distinct) builders is only contrasted to composing an internal DSL with the host language: "You can also use the host language's abstraction features to help make the composition [of internal DSLs] work" [p. 111, 6]. CHAIN OF BUILDERS puts the spotlight on the forces, as well as the different design and implementation options regarding *abstraction features* for composable BUILDERS, ranging from subclassing, mixins (decorators and default methods), to meta-programming.
- DjDSL [18] is a DSL development system for developing families of domain-specific languages (DSLs). DjDSL allows for designing and for implementing a DSL family's abstract syntax in a variable manner, along with context conditions and multiple concrete syntaxes. There is support for developing mixed DSLs, that is, DSL having two or more internal or external syntaxes (in the spirit of Frag [23], but based on parsing grammars) and support for the various DSL composition types (extension, unification, extension composition, and even self-extension). As part of DjDSL, *variable* internal DSL syntaxes can be defined and maintained using CHAIN OF BUILDERS implemented as *decorator mixins* [24].
- Mixed-syntax processing: When BUILDERS form part of an XML-to-object mapper [12] or a JSON-to-object mapper [19], building CHAINS OF BUILDERS allows for processing mixed content, e.g., XML documents with CDATA sections holding JSON documents. At the head of the chain, an XML BUILDER is responsible for the XML processing, forwarding to a JSON

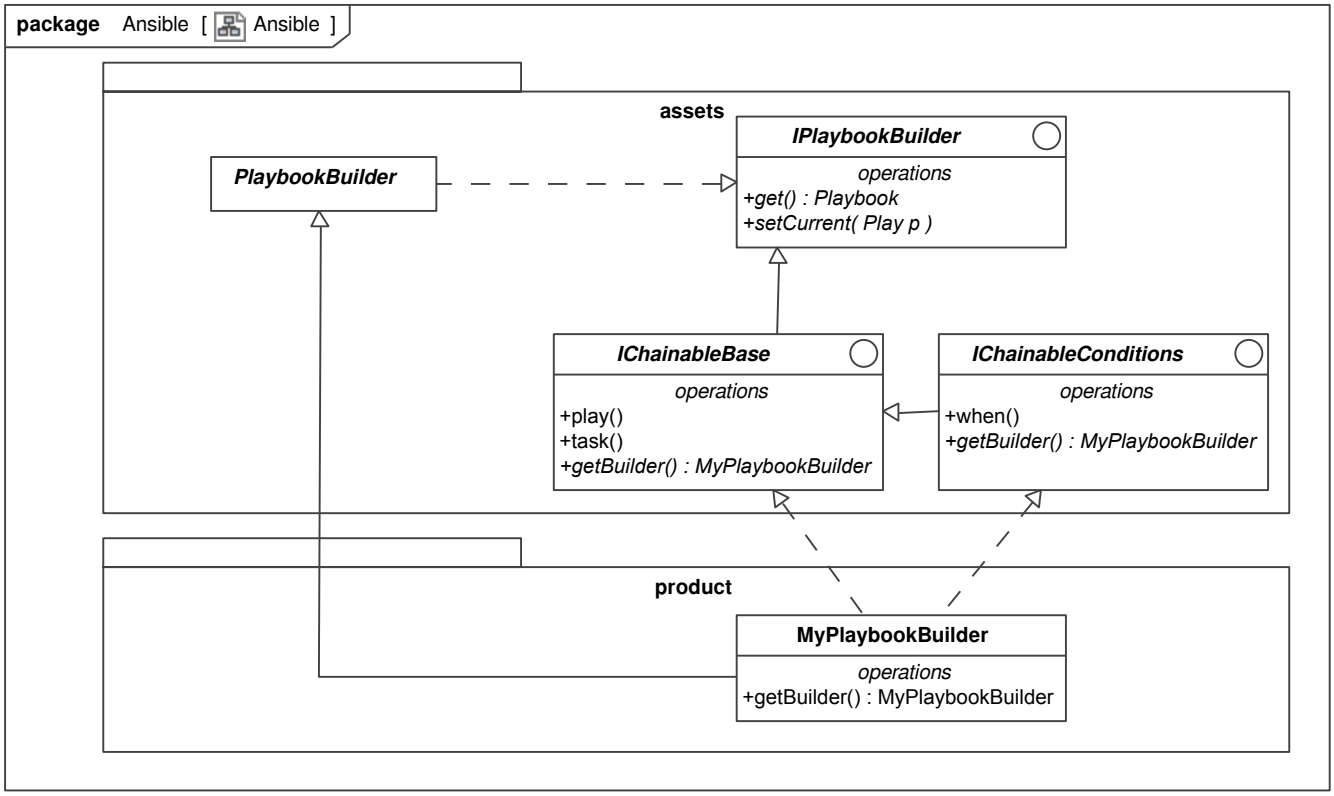

Figure 8: DSL extension: An overview of implementing a variable internal Java syntax using default methods, to provide for the Ansible playbooks in Listing 7.

BUILDER at the tail for the protected (CDATA) content. More generally, or in the same instance, BUILDERS each responsible for processing different XML fragments (e.g., corresponding to different XML schema fragments) or different JSON fragments (e.g., corresponding to different JSON schema fragments) can be combined into CHAINS, to reuse partial builders for different variants of input documents.

4 APPLYING CHAINS OF BUILDERS: SOME STORIES

The pattern description in Section 3 stresses examples of DSL extension. In DSL extension, a DSL developer composes a base language (Ansible playbooks with plays and tasks) with a language extension (when clauses for plays). Generally, a language extension is an incomplete language fragment which depends directly on the base language for completion (in terms of the concrete syntax, the abstract syntax, and the behaviours; [5]).

In the following, the focus is shifted towards additional composition types for DSLs (unification, extension composition); and towards how CHAINS OF BUILDERS help support them at the syntax level.

4.1 DSL restriction

DSL restriction has been considered a variant of DSL extension [5]; not undisputedly, though. At the level of concrete syntax, equivalently, restricting an internal DSL can be realised by prepending special-purpose BUILDERS at the chain's head. Preceding BUILDERS can consume then restricted DSL invocations. The consuming BUILDER can either report the disallowed syntax element or discard it silently, depending on the type of restriction sought. This way, no corresponding abstract-syntax elements end up in the language-model instantiations, to begin with. This form of syntax-level DSL restriction may apply to either the base DSL, or any DSL composition.

Let us return to the running example of Ansible playbooks, once more. A restricted variant of the playbook DSL supporting conditionals (see Listing 9) may impose additional constraints onto the use of *when* clauses; or a restricted variant may prohibit the use of *when* clauses entirely. Pruning an entire syntax clause is more likely a requirement towards a complete base syntax, rather than on the result of a DSL extension. Nevertheless, for the sake of continuity, I stick with this example for the time being. See Listing 10.

The composition class `MyPlaybookBuilder` can provide a method implementation for `when` that overrides the default method provided

```
public final class MyPlaybookBuilder
  extends    PlaybookBuilder
  implements IChainableConditions<MyPlaybookBuilder>
        {

  @Override
  public MyPlaybookBuilder getBuilder() {
    return this;
  }

  public static MyPlaybookBuilder Playbook() {
    return new MyPlaybookBuilder();
  }
}
```

```
public interface IChainableConditions<A>
extends IChainableBase<A> {
  /* required interface */
  A getBuilder();

  default A when(String expression) {
    if (getCurrent() == null ||
        getCurrent().getTasks().size() == 0) {
      throw new UnsupportedOperationException();
    }
    /* get current Task */
    getCurrent().
    getTasks().
    get(getCurrent().getTasks().size()-1);
    /* ... */
    return getBuilder();
  }
}
```

Listing 9: The composition class `MyPlaybookBuilder` (top) and the extension interface `IChainableConditions` (bottom).

```
public final class MyPlaybookBuilder
  extends PlaybookBuilder
  implements IChainableConditions<MyPlaybookBuilder> {

  @Override
  public MyPlaybookBuilder when(String expression) {
    /* throw new UnsupportedOperationException(); */
    return getBuilder();
  }
}
```

Listing 10: Exemplary implementation of a restricted variant of Ansible playbooks, based on Java default methods and overrides.

by the implemented interface `IChainableConditionals`. The overriding method does not forward the call to the default method, therefore, effectively disabling this syntax element. By either returning the BUILDER object or by throwing an exception (both, either unconditionally or conditionally), the DSL restriction can be realised in different ways. With this override in place, the DSL script from Listing 7 (bottom) will either execute (with *when* clauses silently ignored) or an exception will be signalled.

4.2 Extension composition

In extension composition, two or more extensions are composed with one another, or become combined into one amalgam extension, before entering a composition with a base DSL. Picture the

```
    MyPlaybookBuilder.Playbook()
    .play()
      .taskWhen("run some command",
                "statusCode is failed")
      .taskWhen("run another command",
                "statusCode is succeeded");
```

```
public interface IChainable2in1<A>
  extends IChainableBase<A>,
          IChainableConditions<A> {

  /* required interface */
  A getBuilder();

  default A taskWhen(
      String name,
      String expression) {
    task(name);
    when(expression);
    return getBuilder();
  }
}
```

Listing 11: An Ansible playbook variant with a combined `taskWhen` (top), reusing the implementations of the separate `task` and `when` (bottom).

requirement of providing a variant of Ansible playbooks which offer a combined `taskWhen` syntax element, rather than a separated conditional `when` on `task`. At the same time, this variant should not reimplement (duplicate) any of the syntax-processing implementations of the two separated calls.

In the example, the CHAIN OF BUILDERS for composing two (or more) syntax extensions can be implemented by extending the two readily available interfaces `IChainableBase` and `IChainableConditions` using a third composition interface: `IChainable2in1` (see Listing 11, at the bottom). This interface defines a corresponding default method `taskWhen`. The method implementation calls the existing methods `task` and `when`, available from the two extended interfaces. A `MyPlaybookBuilder` class can now implement this single interface to expose the combined syntax of all three interfaces (rather than implementing all three of them).

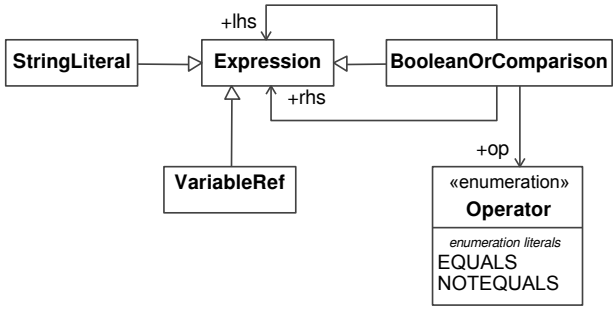

Figure 12: An exemplary language model (a.k.a. SEMANTIC MODEL) for a minimal expression language.

4.3 DSL unification

In DSL unification, two or more, otherwise free-standing and independent DSLs are composed, at all levels: language models, concrete

```
    Playbook()
      .play()
        .task("run some command")
        .when(Test()
            .variable("statusCode")
            .eq("failed"))
        .task("run another command")
        .when(Test()
            .variable("statusCode")
            .eq("succeeded"));
```

```
public interface IChainableConditionsWithExpr<A>
extends IChainableBase<A> {
  default A when(TestExprBuilder expression) {
    /* ... */
    return getBuilder();
  }
}
```

Listing 13: An Ansible playbook variant with `when` conditions that accept test expressions formulated using a second DSL (at the top), reusing the implementations of the corresponding EXPRESSION BUILDER: `TestExprBuilder` (at the bottom).

syntaxes, and behaviour implementations. This may be motivated by integrating (reusing) a generic *kernel language* (e.g., a language to capture expressions [20]) into a more specific language for a given application (i.e., the Ansible playbook language). Let us revisit the `task` and `taskWhen` facilities introduced earlier (see Listings 7 and 11). They represent test or trigger conditions, and these conditions are opaque strings from the perspective of the Ansible playbook DSL (e.g., "statusCode is failed"). Assumingly, these strings are valid and have meaning according to some external syntax for an (non-Java) language. This is an example of FOREIGN CODE [6, pp. 309]. As an alternative, the Ansible playbook DSL could be extended to capture these test conditions as an integral part of playbooks. However, this may complicate the playbook DSL and bloat the language model (given that the conditions are optional parts). In addition, one will find that these test expressions resemble expressions typically found in software-testing languages; and general-purpose functional languages [20]. So, why re-invent this wheel?

If available, an existing *internal* language for expressions can be composed with the Ansible playbook to form a unified DSL, as a middle ground. Consider an exemplary model for expressions in Figure 12. A corresponding EXPRESSION BUILDER (`TestExprBuilder`, implementation not shown) could realise an internal syntax based on METHOD CHAINING on top of this language:

```
    TestExprBuilder.Test()
     .variable("varName").eq("foo")
```

Once processed, this syntax fragment yields an instantiation of the model in Figure 12. This instance contains a `BooleanAndComparison` expression, with an `EQUALS` operation between a left-hand `VariableRef` resolving to a variable `statusCode` and a right-hand `StringLiteral` of value `failed`.

Using a CHAIN OF BUILDERS, this internal DSL can be embedded with the Ansible playbook DSL to render `when` and `taskWhen` conditions first-class (see Listing 13, top). The integration vehicle is a composable interface `IChainableConditionsWithExpr` that provides a chainable method `when` accepting an instance of the EXPRESSION BUILDER instance for test expressions (see Listing 13, bottom). Internal to the default-method's implementation, this EXPRESSION BUILDER can then be used to obtain the corresponding language-model instance of expressions and to link the expression to some evaluation context (e.g., sourced from the runtime occurrence of some Ansible task or module), for conducting the variable tests, eventually. A unifying EXPRESSION BUILDER then offers means to create method chains, both for defining playbooks (`MyPlaybookBuilder.Playbook()`) and for defining test expressions (`MyPlaybookBuilder.Test()`) in one composite expression (see Listing 13, bottom). The CHAIN OF BUILDERS to obtain this DSL unification is as follows (in order of method resolution for METHOD CHAINING)

(1) `MyPlaybookBuilder`
(2) `IChainableConditionsWithExpr` (via `implements`)
(3) `IChainableBase` (via `extends`)

5 CONCLUDING REMARKS

This paper documents known and proven practises to implement variable internal syntaxes of domain-specific languages (DSL); or object-oriented APIs for variable syntax processing. This documentation is centred around the description of CHAIN OF BUILDERS pattern. This pattern corresponds to the combined application of two established patterns: EXPRESSION BUILDER for separating concerns in syntax processing and CHAIN OF RESPONSIBILITY for extending syntax processors in a stepwise manner. The objective is to minimise preplanning (i.e., avoid anticipating explicit extension points) while keeping syntax extension independent from each other (i.e., not requiring invasive changes to existing assets). Relationships to other architecture and language-design patterns, e.g., by Fowler [6], are discussed. Applying CHAIN OF RESPONSIBILITY to implement DSL extensions, DSL extension compositions, and DSL unification are presented. The running examples provide Java implementation examples of internal DSLs for deployment descriptors (Ansible playbooks) and expressions, respectively.

ACKNOWLEDGMENTS

Special thanks are due to Michael Krisper as my shepherd, for his motivating feedback and pointers to known uses beyond DSLs. Thanks also go to all EuroPLoP 2019 workshop participants, who contributed to improving our paper with their comments. This work was partly funded by the Austrian research funding association (FFG) under the scope of the HybriDLUX project[2] within the funding programme ICT of the Future (6th call 2017) of the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT), contract 867535.

REFERENCES

[1] Frank Buschmann and Kevlin Henney. 2003. Explicit Interface. In *Proceedings of EuroPLoP 2003.* Irsee, Germany.
[2] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. 2007. *Pattern-oriented Software Architecture – On Patterns and Pattern Languages.* John Wiley & Sons.

[2] https://hybridlux.wu.ac.at/

[3] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal (Eds.). 2000. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons Ltd., Chichester, England.
[4] Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative Programming — Methods, Tools, and Applications* (6th ed.). Addison-Wesley.
[5] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. 2012. Language Composition Untangled. In *Proc. Twelfth Workshop on Language Descriptions, Tools, and Applications (LDTA'12)*. ACM, 7:1–7:8. https://doi.org/10.1145/2427048.2427055
[6] Martin Fowler. 2010. *Domain Specific Languages* (1st ed.). Addison-Wesley.
[7] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. 1995. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley.
[8] Debasish Ghosh. 2010. *DSLs in Action* (1st ed.). Manning Publications Co.
[9] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. 2015. The Java® Language Specification. Available online; last accessed: 05.04.2019. https://docs.oracle.com/javase/specs/jls/se8/html/index.html
[10] Bernhard Hoisl, Stefan Sobernig, and Mark Strembeck. 2017. Reusable and generic design decisions for developing UML-based domain-specific languages. *Information and Software Technology* 92 (July 2017), 49–74. https://doi.org/10.1016/j.infsof.2017.07.008
[11] Tomaž Kosar, Sudev Bohra, and Marjan Mernik. 2016. Domain-Specific Languages: A Systematic Mapping Study. *Information and Software Technology* 71 (2016), 77–91. https://doi.org/10.1016/j.infsof.2015.11.001
[12] R. Lämmel and E. Meijer. 2007. Revealing the X/O impedance mismatch (Changing lead into gold). In *Datatype-Generic Programming (Lecture Notes in Computer Science)*, Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring (Eds.). Springer.
[13] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and how to develop domain-specific languages. *Comput. Surveys* 37, 4 (2005), 316–344. https://doi.org/10.1145/1118890.1118892
[14] Terence Parr. 2009. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages* (1st ed.). Pragmatic Bookshelf.
[15] Dirk Riehle, Michel Tilman, and Ralph Johnson. 2005. Dynamic Object Model. In *Pattern Languages of Program Design 5*. Addison-Wesley, 3–24.
[16] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. 2000. *Pattern-Oriented Software Architecture*. John Wiley & Sons Ltd.Wiley, Chichester, England, Chapter Whole-Part, 225–242.
[17] Marco Servetto, Julian Mackay, Alex Potanin, and James Noble. 2013. The Billion-Dollar Fix. In *Proc. 27th Europ. Conference Object-Oriented Programming (ECOOP'13) (LNCS)*, Vol. 7920. Springer, 205–229.
[18] Stefan Sobernig. 2018. DjDSL. Available at: https://github.com/mrcalvin/djdsl/.
[19] Uta Störl, Thomas Hauf, Meike Klettke, and Stefanie Scherzinger. 2015. Schemaless NoSQL data stores - object-nosql mappers to the rescue?. In *Datenbanksysteme für Business, Technologie und Web (BTW'15)*. Gesellschaft für Informatik e.V., 579–599.
[20] Markus Völter. 2018. The Design, Evolution, and Use of KernelF. In *Proc. 11th International Conference on Model Transformation (ICMT'18) (LNCS)*, Vol. 10888. Springer, 3–55.
[21] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. 2013. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. dslbook.org. http://www.dslbook.org
[22] Uwe Zdun. 2006. Patterns of Component and Language Integration. In *Pattern Languages of Program Design 5*, D. Manolescu, M. Völter, and J. Noble (Eds.). Addison-Wesley, Chapter 14, 357–400.
[23] Uwe Zdun. 2010. A DSL toolkit for deferring architectural decisions in DSL-based software design. *Information and Software Technology* 52, 7 (2010), 733–748. https://doi.org/10.1016/j.infsof.2010.03.004
[24] Uwe Zdun, Mark Strembeck, and Gustaf Neumann. 2007. Object-based and class-based composition of transitive mixins. *Information and Software Technology* 49, 8 (2007), 871–891. https://doi.org/10.1016/j.infsof.2006.10.001