LANGUAGE-ORIENTED PROGRAMMING VIA DSL STACKING

Bernhard G. Humm
Darmstadt University of Applied Sciences, Department of Computer Science, Darmstadt, Germany

Ralf S. Engelschall
Capgemini sd&m Research, Munich, Germany

Keywords: Language-oriented programming, Extensible programming languages, Domain-specific languages, Meta programming, Semantic web, Business information systems, Lisp.

Abstract: According to the paradigm of Language-Oriented Programming, an application for a problem should be implemented in the most appropriate domain-specific language (DSL). This paper introduces DSL stacking, an efficient method for implementing Language-Oriented Programming where DSLs and general-purpose languages are incrementally developed on top of a base language. This is demonstrated with components of a business information system that are implemented in different DSLs for Semantic Web technology in Lisp.

1 INTRODUCTION

When analyzing large-scale commercial IT projects today, you usually find that five or even more languages are being used – even in a "pure Java project" if you take a closer look. HTML and JavaScript may be used for GUI programing, SQL or EJBQL for database accesses, UML for data modeling, Perl for the UML to Java generator, BPEL for process logic and several XML dialects for configuration. Why would projects put up with the high costs of heterogeneity? This is because *different problem domains require different language constructs*.

This is the main idea of a paradigm called *Language-Oriented Programming* (Ward, 1994; Fowler, 2005; Dmitriev, 2005; Brauer et al., 2008). It can be briefly summarized as follows. *When you have to solve a complex problem, first choose – or, if necessary, develop – a language that is most appropriate for the problem. Then, implement the solution in this language.*

In most publications today, such a language is called a *Domain-Specific Language (DSL)* (van Deursen et al., 2000) – see Sect. 5.2. Earlier publications use the term *Problem-Oriented Language* (Goldfinger, 1961). In this paper we stick to the term DSL although it shall be noted that the problem area does not have to be a business domain and the language may be more or less specific, depending on its purpose.

Like all software engineering paradigms, Language-Oriented Programming ultimately serves one goal: to allow for developing applications of *high-quality* at *reasonable cost*. To this end, which criteria do we expect from a DSL most appropriate for a particular domain and why?

Sufficient Semantic Expressiveness. The DSL must allow to express logic necessary for the respective application domain directly in the language, e.g., rules in a knowledge-based system. This is necessary for implementing the application at all.

Conciseness and Comprehensibility. Statements should be expressed adequately, i.e., as concisely and comprehensibly as possible. Boilerplate code should be avoided. Inadequate representation has several drawbacks. Firstly, developing unnecessarily voluminous code increases the costs for developing the application. Even higher are the follow-up costs for maintenance. Secondly, the quality of the solution decreases since errors cannot be detected as easily. The more adequate the representation – i.e., the closer to the problem domain – the more problem domain experts can be involved in the development.

Integration. It must be possible to integrate the DSL application with other applications, possibly im-

G. Humm B. and S. Engelschall R. (2010).
LANGUAGE-ORIENTED PROGRAMMING VIA DSL STACKING.
In *Proceedings of the 5th International Conference on Software and Data Technologies*, pages 279-287
DOI: 10.5220/0002925402790287

plemented in other DSLs. This includes invocation at run-time and, ideally, integrated development including integrated debugging. Integration is mandatory if the DSL application is a component of a larger application, e.g., a business information system. Integrated development decreases development costs, particularly for testing and debugging.

Performance. Applications implemented in the DSL must meet the respective performance requirements.

Those aspects help the *application developer* (designer and implementer). Language-Oriented Programing adds another role: the *language developer* (designer and implementer). The following aspect helps the language developer.

Efficient DSL Development. The design, implementation including validation logic, testing, and maintenance of DSLs should be as efficient as possible. Efficient DSL development directly affects the total development cost.

In this paper, we present a method for Language-Oriented Programming called *DSL Stacking*. DSL stacking may meet all the criteria described above. We demonstrate this with a sample Lisp application using Semantic Web technology in the insurance business.

This article is structured as follows. Sect. 2 and 3 give an overview of Language-Oriented Programming, Semantic Web technology, the insurance sample and Lisp. In Sect. 4 we present a family of DSLs for applying Semantic Web technology in the insurance business. Sect. 5 explains the concept of DSL stacking. In Sect. 6, we discuss the results. Sect. 7 concludes this article.

2 LANGUAGE-ORIENTED PROGRAMMING

The term *Language-Oriented Programming* (Ward, 1994; Fowler, 2005; Dmitriev, 2005; Brauer et al., 2008) has already been used for a number of years. The idea, however, is even much older and has been around since the invention of the first high-level programming languages. There have been a number of *extensible language initiatives* in the 1960 and 1970s, e.g, translator writing systems like "The Syntax Directed Compiler" (Irons, 1961) or extensible programming languages like Lisp (McCarthy, 1960), IMP (Irons, 1970), and Smalltalk-72 (Kay, 1993). Extensible languages facilitate the development of DSLs. The Unix little languages like AWK (Aho et al., 1988) or make (Feldman, 1979) are examples of DSLs.

Recently, there have evolved a number of Language-Oriented Programming initiatives. Apart from OMG's MDA initiative these are the STEPS project (Kay et al., 2008), Microsoft's "Software Factories" (Greenfield, 2004), "Intentional Programming" (Simonyi et al., 2006) and the "Meta-Programming System" (Dmitriev, 2005). In those publications, the term "language workbench" is used as an integrated environment to design and implement DSLs and to program applications in those DSLs.

Additionally, a number of new extensible programming languages like F#, Ruby, Groovy, and Scala have evolved in the last decade. They explicitly provide features for language-oriented programming like Groovy's Meta-Object Protocol (MOP).

Two approaches to Language-Oriented Programming can be distinguished: *external DSLs* and *internal DSLs* (Fowler, 2005).

Internal DSLs are idiomatic ways of writing higher-level language code in the host language of the application (*"base language"* – see Sect. 5.2) directly. Example: AllegroProlog, a Prolog (Clocksin and Mellish, 1987) implementation by Franz Inc. fully integrated in Lisp that allows Prolog programming in Lisp notation. The extensible programming language initiatives all fall into this category.

External DSLs are different from the base language. DSL programs are scanned, parsed and transformed into the base language using some form of compiler or interpreter. Example: a BPEL (OASIS, 2007) engine parses XML code and transforms it into data structures of a programming language like Java. The Unix little languages and MDA fall into this category. External DSLs may be embedded in a host language, e.g., as strings, or may be programmed in separate files.

In this paper, we focus on internal DSLs.

3 SAMPLE DOMAIN: SEMANTIC WEB TECHNOLOGY IN THE INSURANCE BUSINESS

3.1 Semantic Web Standards

The World Wide Web Consortium (W3C) has standardized a number of languages under the term *"Semantic Web"* (Berners-Lee et al., 2001). The lan-

guages are based on the technologies XML and URI. The main Semantic Web standards are as follows (World Wide Web Consortium (W3C), 2010).

RDF (Resource Description Framework) is the basic Semantic Web modeling language. The fundamental data structure is a triple consisting of subject, predicate, and object.

RDF-S (RDF Schema) builds on top of RDF adding the concepts of classes and properties.

OWL (Web Ontology Language) builds on top of RDF-S and adds expressive set-based logic.

SPARQL (SPARQL Protocol and RDF Query Language) allows for querying RDF-S and OWL statements.

3.2 RDF Triples

The *triple* is the basic yet powerful data structure of RDF. Every triple consists of *subject, predicate,* and *object*. See Figure 1. It allows for basic statements like "Peter has an income of $ 65,000". It furthermore allows statements about class / instance relationships as in "Peter is a Person" and "Person is a Class". It also allows statements about properties like "has-income is a Property".

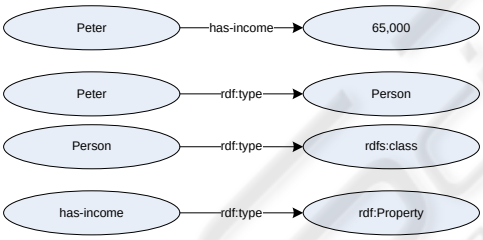

Figure 1: RDF triples.

3.3 Serialization Syntaxes for RDF

Different serialization syntaxes for RDF have been standardized: RDF/XML, N3, Turtle, and N-Triples. The W3C recommends RDF/XML for exchanging RDF triples.

The following code example implements an RDF triple stating that "Peter has the disease heart-stroke".

```
<rdf:RDF
  xmlns:rdf="http://-
    www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:ins="http://
    www.fbi.h-da.de/insurance-example#">
  <rdf:Description rdf:about="http://
    www.fbi.h-da.de/insurance-example#Peter">
    <ins:has-disease>
      <rdf:Description
        rdf:about="http://www.fbi.h-da.de/
          insurance-example#Heart-Stroke">
      </rdf:Description>
    </ins:has-disease>
  </rdf:Description>
</rdf:RDF>
```

Ten lines of XML code are necessary for expressing a simple triple. With this verbose code, RDF/XML is certainly no adequate representation for Semantic Web programming. Other representations more or less alleviate shortcomings of RDF/XML to provide a more concise and human-comprehensible form – one of the main concerns of Language-Oriented Programming.

3.4 Business Domain: Insurance

We have developed a family of DSLs and have implemented an insurance example application using those DSLs. The example is about issuing disability insurance. The customer interested in a disability insurance must provide, apart from personal data, information about employment, financial situation, health, and lifestyle. Based on this situation, the application will perform a customer rating. For example, severe diseases present in the family history may lead to the decision of not offering a policy to the applicant. Based on the rating, the application automatically accepts or declines the application or involves human intervention.

3.5 Lisp as Base Language

Lisp (McCarthy, 1960) is one of the earliest high-level programming languages. To date, numerous dialects and versions have been specified and implemented. Two aspects of Lisp make it ideal for developing internal DSLs:

Code is Data. Lisp data are represented by *symbolic expressions*, so-called *S-expressions* (McCarthy, 1960). S-expressions are atoms like numbers or characters and arbitrary lists of S-expressions. Lisp programs are also expressed as S-expressions: function definitions, control structures, class declarations etc. So, the syntax tree of Lisp code is directly expressed as Lisp data and, hence, can itself be transformed by Lisp programs.

Macro Processor. With the built-in macro processor, language extensions to Lisp can be implemented efficiently and with limited effort.

For those reasons, we have chosen Lisp for the implementation of the sample application. However, it shall be noted that Language-Oriented Programming is not at all restricted to Lisp.

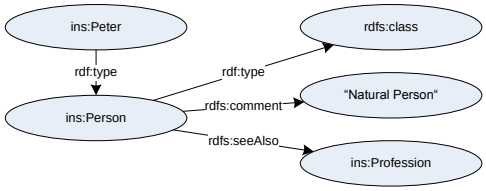

Figure 2: Definition of a class and an instance.

We used the Allegro CL Express Edition by Franz Inc. as integrated development environment for programming Allegro Common Lisp, a professional implementation of the ANSI Common Lisp standard (Steele, 1984). Allegro CL was chosen due to its extensive libraries, particularly AllegroGraph for Semantic Web programming.

4 DSLS FOR THE SAMPLE DOMAIN

4.1 Parts

The code example in Sect. 3.3 has shown that RDF/XML does not fulfill the requirement of being concise and comprehensive: ten lines of code for expressing one triple. We, therefore, introduce more adequate language constructs. In the first step, we abbreviate URI namespaces as in XML qualified names. As a language extension, we use the ! macro of AllegroGraph.

Instead of writing

```
http://www.fbi.h-da.de/insurance-example#
Heart-Stroke
```

we can simply write

```
!ins:Heart-Stroke
```

– assuming that we have registered the name space:

```
(register-namespace "ins"
  "http://www.fbi.h-da.de/insurance-example#")
```

4.2 Triples

Next, we use the `add-triple` macro of AllegroGraph as a concise syntax for specifying triples. The following code example is equivalent to the verbose RDF/XML triple definition from Sect. 3.3.

```
(add-triple !ins:Peter !ins:has-disease
            !ins:Heart-Stroke)
```

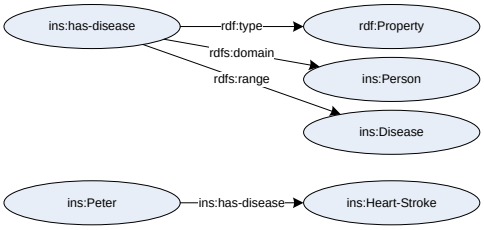

Figure 3: Definition of a property.

4.3 Classes, Subclasses and Instances

So far, we have used concise Lisp-based notations for expressing parts and triples, the basic RDF data structures. Now, we extend the language by higher-order constructs for classes, subclasses, and instances. We define the macros `add-class`, `add-subclass`, and `add-instance`. Figure 2 shows an example in RDF-S. The equivalent statements are expressed in two lines of DSL code as follows.

```
(add-class !ins:Person :comment "Natural Person"
  :see-also !ins:Profession)
(add-instance !ins:Peter !ins:Person)
```

`:comment`, `:see-also`, and `:label` are optional keyword parameters to provide further information about the class. We have provided them for all higher-order triple-generating macros. They can all be omitted as, in this example, `:label`.

The macro definition of `add-class` is straight forward.

```
(defmacro add-class
  (class &key comment label see-also)
   `(progn
      (add-triple ,class !rdf:type !rdfs:class)
      ,(if comment `(add-triple ,class
        !rdfs:comment (literal ,comment)))
      ,(if label `(add-triple ,class
        !rdfs:label (literal ,label)))
      ,(if see-also `(add-triple ,class
        !rdfs:seeAlso ,see-also))))
```

The macro always generates an `rdf:type rdfs:class` triple and, optionally, up to three triples for comment, label, and see-also. It allows specifying several statements in one expression. Additionally, it frees the modeler from unnecessary and possibly confusing details, e.g., that the predicate `type` is defined in namespace `RDF` whereas the class `class` is specified in namespace `RDFS`.

4.4 Properties

Properties in RDF-S are defined via three triples, specifying the property itself, its domain and range. Figure 3 shows an example. We have defined the macro `add-property` that allows programmers to express those three triples in a single, concise expression:

```
(add-property !ins:Person
  !ins:has-disease !ins:Disease)
(add-triple !ins:Peter
  !ins:has-disease !ins:Heart-Stroke)
```

The RDF triple as a binary predicate is a simple yet powerful building block. However, in semantic modeling often n-ary predicates are needed, with $n \geq 1$. For this reason, we have extended the add-property macro to cope with different arities. See the following examples.

```
(add-property !ins:Disease !ins:is-inheritable)
(add-statement !ins:Heart-Stroke
  !ins:is-inheritable)

(add-property !ins:Person !ins:takes-drug
  !ins:Drug !ins:Frequency)
(add-statement !ins:Peter !ins:takes-drug
  !ins:Alcohol !ins:rarely)
```

The first example shows a unary predicate expressing whether a disease is inheritable – here: heart stroke. The likelihood of suffering from a heart stroke is increased for people whose direct relatives have had a stroke. The predicate is implemented with range `xs:Boolean`.

The second example shows a tertiary predicate expressing the drug consumption behavior of a person – here that Peter drinks alcohol rarely. The predicate is implemented using RDF blank nodes.

So far, our DSLs contain general-purpose Semantic Web extensions: parts, triples, classes, and predicates. In the following section, we add business-domain specific extensions.

4.5 Business Domain Specific Extensions

The following example

```
(add-insurance-statements !ins:Peter
  :daily-duty :sales
  :income 80000
  :height 175
  :weight 98
  :drugs '(:Tobacco :regularly)
  :disease-of-relative :Heart-Stroke)
```

is expanded to:

```
(add-instance !ins:Peter !ins:Person)
(add-triple !ins:Peter
  !ins:has-daily-duty !ins:Sales)
(add-triple !ins:Peter !ins:has-income
  (value->upi  80000 :int))
(add-triple !ins:Peter !ins:has-height
  (value->upi 175 :int))
(add-triple !ins:Peter !ins:has-weight
  (value->upi 98 :int))
(add-statement !ins:Peter !ins:takes-drug
  !ins:Tobacco !ins:regularly)
(add-triple !ins:Peter
  !ins:relative-has-disease !ins:Heart-Stroke)
```

Again, the notation has become more concise and more comprehensible.

5 DSL STACKING

5.1 Bootstrapping

The German author R. E. Raspe (1736-1794) has written the wonderful tale of the impostor Baron Münchhausen. Münchhausen told his credulous audience that he once got into a swamp and he started to sink in. Quick-wittedly, he pulled himself out of the swamp by his hair or – in the less painful variation of the story – on his bootstraps.

Bootstrapping has since then become a metaphor for self-sustaining processes of that kind. With our Semantic Web DSL, we have done one kind of bootstrapping. As the example in Sect 3.3 (ten lines of code for a simple triple) shows, RDF/XML is like a swamp for application developers. Instead of expressing complex matters in the language of their business domain, they sink into a mass of unnecessary, confusing boilerplate code. By defining the DSLs in Sect. 4, we have – step by step – pulled ourselves out of this swamp:

1. Name spaces instead of full URIs
2. Concise triple notation
3. Classes, subclasses and instances
4. Predicates of different arities
5. Business-domain specific extensions

At the end of the bootstrapping process, the application developer is able to express business logic in a concise, comprehensible form.

Analyzing the bootstrapping process shows that we have applied the principle of stacking to our DSLs. We call this method *DSL Stacking*.

Before specifying the method, we need to define a few concepts.

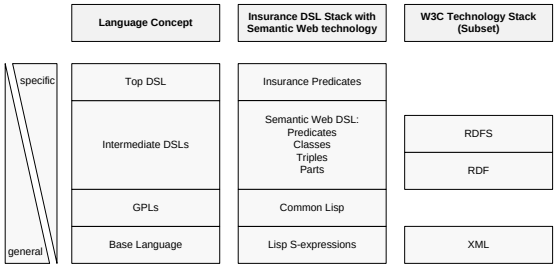

Figure 4: DSL stacks.

5.2 Concepts

In DSL stacking, we distinguish different kinds of languages, all of which are specified by syntax and semantics.

- A *base language* is a language that allows the specification and / or implementation of other languages. Minimally, it must contain *primitives* like numbers and characters and *composition features* like nested lists. Additional features include, e.g., name spaces. Base languages are usually self-describing, i.e., they can be specified in themselves. Examples are: XML as the base language for all XML dialects like BPEL; MOF as the base language for UML, CORBA IDL etc.; Lisp S-expressions as the base language for Common Lisp as well as the Semantic Web DSL from Sect.4.
- A *general purpose language (GPL)* is programming or specification language for a wide variety of problems. Examples are C (Kernighan and Ritchie, 2008), Java (Gosling et al., 2005), or OMG's UML.
- A *domain-specific language (DSL)* is a programming or specification language dedicated to a particular problem domain. Examples are WS-BPEL (OASIS, 2007) or the DSLs from Sect.4.
- A *DSL stack* is a set of languages with *based-on* dependencies. A language $L1$ is based on another language $L2$ if $L1$ is specified or implemented in $L2$. DSLs may be based on other DSLs, GPLs, or on the base language.

Figure 4 shows the DSL stack for the Semantic Web DSL from Sect. 4. Languages further up in the stack are based on the lower ones. As the base language we use Lisp S-expressions. This is analogous to XML as base language of the W3C Semantic Web Stack. On top of the base language, we use Common Lisp as a GPL to implement DSLs for parts and for triples. They implement the semantics of RDF. On top of the base technologies, we have implemented general purpose Semantic Web extensions for classes and predicates. They provide semantics equivalent to RDF-S. In fact, they extend RDF-S insofar as they provide classes, subclasses, instances, and predicates as language constructs whereas in RDFS one is constrained to the lower-level triple notation. Again on top we have implemented predicates that are specific to the insurance business domain.

5.3 The Language Stacking Method

We postulate the following *requirements* for the different languages on the DSL stack before we define the language stacking method.

Base Language. A base language should have a *minimalistic intrusive syntax* and *built-in bootstrapping features*.

In Lisp, the S-expression syntax is minimalistic, allowing for defining arbitrary concepts like data structures, control structures, and aspects of all kinds.

Bootstrapping features allow the implementation of new language constructs. In Lisp, these are the macro processor in combination with the fact that code is represented as data. Other examples of bootstrapping features are the Meta-Object Protocol (MOP) in Groovy, or implicit conversions in Scala.

Intermediate DSLs are those DSLs, in which the application developer does not program directly, i.e., which are not on top of the DSL stack. They should focus on a particular problem domain and be *simple* yet *expressive* (Felleisen, 1990). The triple notation is an excellent example being extremely simple yet allowing for implementing all higher-level constructs like classes, subclasses, instances, and predicates.

Top DSL. The top DSL is the one in which the application developer programs directly, i.e. the one on top of the DSL stack. It should be most *adequate* for the application being developed: sufficiently *expressive*, *concise* and *comprehensible* for the domain expert.

DSL stacking is a method for language developers to provide DSLs for application developers. It consists of the following steps.

1. Select a base language that meets the requirements specified above.
2. If necessary, select a GPL implemented on top of the base language.
3. While the requirements for a top DSL are not yet met: select or develop DSLs on top of the stacked

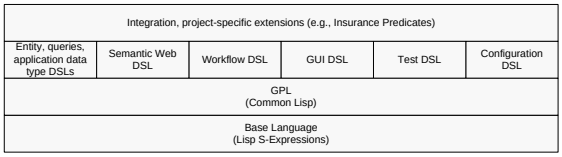

Figure 5: Language hierarchy.

languages. Those languages must meet the requirements for intermediate DSLs.

Language stacking as a method is to be used by the *language* developer. The *application* developer does not have to be aware of the DSL stack since he is programming in the top DSL only. Whether or not the top DSL includes language features of lower-level languages is a matter of *visibility*. In internal DSLs, all lower-level language features are visible by default. However, sometimes it is desirable to restrict language features by hiding lower-level DSLs, e.g., for security reasons or in order to reduce the complexity for the application developer.

5.4 Integrating Multiple Languages

So far, we have considered a single stack of DSLs only. When building large applications like business information systems, different tasks require different kinds of DSLs. Stacking all languages on the same base language allows to seamlessly integrate components implemented in different DSLs. This not only allows for invoking functions from any language to any other language at runtime. It also means integrated development: using the same editor, compiler, and test tools for different languages. Also – most importantly – integrated debugging of applications in different languages is possible.

To demonstrate this, we have implemented or used a number of Lisp-based DSLs for business information systems and have integrated them with the Semantic Web DSL from Sect. 4 (see Figure 5).

- Application data types DSL, based on the Common Lisp type system
- Persistent entity DSL based on AllegroCache
- Entity query DSL based on AllegroCache
- Workflow DSL based on Allegro Common Lisp Multiprocessing
- GUI DSL: Allegro Common Graphics
- Test DSL: LispUnit
- Configuration DSL: ASDF

We have, then, implemented a prototypical insurance application based on those languages. Figure 6 shows an UML component diagram of the application. It is structured as a three-layer architecture

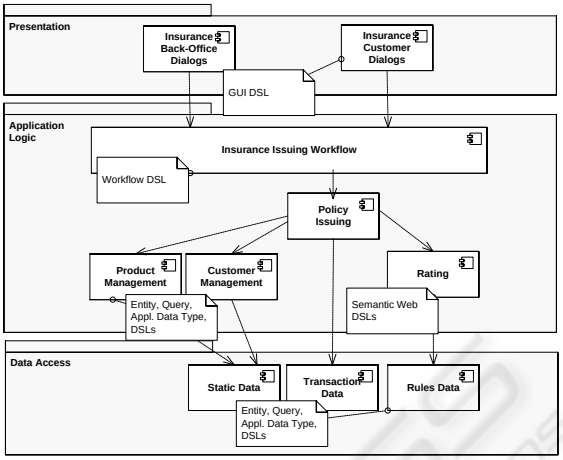

Figure 6: Insurance example component architecture.

in which each component is implemented in one or more DSLs. Insurance Back-Office and Customer Dialogs are implemented with the GUI DSL. The insurance issuing workflow is implemented in the workflow DSL. Product Management, Customer Management and Policy Issueing are implemented in DSLs for Business Information Systems including application data types, persistent entities and entity queries. The rating component is implemented in the Semantic Web DSL described in Sect. 4. Finally, static data, transactional data and rules data are stored and retrieved via the persistent entity DSL.

6 DISCUSSION

6.1 Evaluation of Goals

We now evaluate the approach presented against the criteria for developing high-quality solutions at reasonable cost set out in Sect. 1.

Sufficient Expressiveness. The Semantic Web DSL presented is based on the RDF/RDF-S technology and, hence, expose at least their expressiveness. Together with a Prolog-based rules and query DSL, the insurance rating logic could be expressed directly.

Conciseness and Comprehensibility. Statements should be expressed adequately, i.e., as concisely and comprehensibly as possible. Unnecessary boilerplate code was avoided. We have compared the code size of the insurance application implemented in the Semantic Web DSL to the size of the resulting N-triples code. We measured a saving of 74% in lines of code (one N-Triple part

per line) respectively 77% in code size (number of characters including white space). This will result in considerable less implementation and maintenance cost.

Not numerically measurable but obvious is the increased comprehensibility of the code – understandable and maybe even editable by business domain experts. This will result in higher quality of the solution.

Integration. We have integrated the insurance components, all implemented in different DSLs, with each other to form a business information system. Invocations $Dialog \rightarrow Workflow \rightarrow Rating \rightarrow Database$ did not require an additional middleware. Editing, compiling, and testing was all performed in one development environment (Allegro CL Express Edition). Particularly, integrated debugging of code implemented in different languages was possible and most helpful.

Performance. The Semantic Web DSL has been implemented as Lisp macros which are expanded at compile time and, therefore, cause no runtime performance penalty whatsoever.

Efficient DSL Development. The code for implementing the Semantic Web DSL on top of AllegroGraph described in Sect. 4 is less than 100 lines of Lisp code (loc) including comments. The total amount of code for all DSLs used in the insurance application as described in Sect. 5.4 is less than 1,000 loc. Development and testing of all DSLs was an effort of less than a person month. This can be regarded as an efficient means of implementing such a comprehensive set of DSLs. However, it shall be noted that the implementation is experimental and not production-ready. A major effort in production-ready DSL implementations lies in the meaningful handling of compiler errors, particularly if the DSL is to be used by non-programmers.

This demonstrates the validity of the approach in general and the Semantic Web DSL in particular.

6.2 DSL Stacking

Language stacking is an application of the principle of separation of concerns – like component-orientation and layered architectures. Different aspects are separated into several languages, like components in a system. Different languages are built on top of each other with strict dependencies, like in a layered architecture.

DSL stacking is a form of language stacking. However, there is an important difference to the Semantic Web language stacking. The designers of the Semantic Web standards have, as a standardization body, to stop bootstrapping at a reasonable level of generality. DSL stacking continues bootstrapping from there until the requirements for the top DSL are met, i.e., a most adequate language for the respective problem domain is found. This allows to gain the full benefit of Language-Oriented Programming.

7 CONCLUSIONS

To conclude, we summarize the key statements of this article. When you have a complex problem to solve you should proceed as follows.

1. Use a development environment that allows the design and implementation of DSLs.
2. For each subproblem, choose a DSL which is most appropriate: concise and comprehensible.
3. If such a DSL does not exist, develop one.
4. Stack the DSL on top of a lower-level language. The base language is the bottom of the stack.

Language-Oriented Programming is neither new nor a silver bullet but it helps in the development of applications with high quality at reasonable cost. For applications using XML-based Internet technology like Semantic Web standards, Language-Oriented programming is particularly useful – see also (Graunke et al., 2001). We have demonstrated the effectiveness by measurements of code savings of about three quarters in an experimental implementation of an insurance business information system.

DSL stacking allows for cost-efficient realization of Language-Oriented Programming. We have demonstrated this by implementing a complete Semantic Web DSL in less than 100 loc and by implementing DLSs for application data types, persistent entities, entity queries, workflow, GUI, test, and configuration in less than 1000 loc.

Those convincing numbers are partially due to the prototypical character of the implementation and existing powerful lower-level DSLs like Allegro Graph on which the new DSLs could be stacked. They are also partially due to the fact that Lisp as an extensible language is particularly suitable for implementing internal DSLs.

A downside for applying Language-Oriented Programming and DSL stacking in large-scale industrial projects is that extensible programming languages are currently not in mainstream use. This has a number of consequences: the stock of programmers is limited, development and production tools are less perfected.

However, we notice a revival of extensible programming languages. Within the last decade, quite a number of such languages have been implemented on top of mainstream platforms and their number is increasing rapidly. Examples are implementations of Python, Ruby, Groovy, and Scala on the Java Platform or F# on the .NET platform. Furthermore, there are a number of Lisp implementations on the Java platform, e.g., ABCL, Clojure, Jatha, and CLForJava.

What needs to be done? Professional tool support for Language-Oriented Programming and DSL stacking, integration with mainstream platforms like Java and the development of useful DSLs to be taken off the shelf.

REFERENCES

Aho, A. V., Kernighan, B. W., and Weinberger, P. J. (1988). *The AWK Programming Language*. Addison-Wesley.

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The Semantic Web: A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities: www.scientificamerican.com/article.cfm?id=the-semantic-web. *Scientific American*.

Brauer, J., Crasemann, C., and Krasemann, H. (2008). Auf dem Weg zu idealen Programmierwerkzeugen - Bestandsaufnahme und Ausblick. *Informatik Spektrum*, 31(6):580–590.

Clocksin, W. F. and Mellish, C. (1987). *Programming in Prolog, 3rd Edition*. Springer.

Dmitriev, S. (2005). Language Oriented Programming: The Next Programming Paradigm. http://www.onboard.jetbrains.com/is1/articles/04/10/lop/.

Feldman, S. I. (1979). Make — A Program for Maintaining Computer Programs. *Software: Practice & Experience*, 9:255–265.

Felleisen, M. (1990). On the Expressive Power of Programming Languages. In Jones, N., editor, *Proceedings 3rd European Symposium on Programming (ESOP'90) Copenhagen, Denmark, May , 1990*, volume 432 of *Lecture Notes in Computer Science*, pages 134–151, Berlin, Heidelberg, Springer,.

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

Goldfinger, R. (1961). Problem-oriented programming language structure. *Communications of the ACM*, 4(3):138.

Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005). *The Java language specification*. Addison-Wesley, Upper Saddle River, NJ, 3. ed., 1. print. edition.

Graunke, P. T., Krishnamurthi, S., Van Der Hoeven, S., and Felleisen, M. (2001). Programming the Web with High-Level Programming Languages. In David Sands, editor, *Proceedings 10th European Symposium on Programming (ESOP 2001), Genova, Italy, April 2001*, volume 2028 of *Lecture Notes in Computer Science*. Springer.

Greenfield, J. (2004). Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. http://msdn.microsoft.com/en-us/library/ms954811.aspx.

Irons, E. T. (1961). A Syntax Directed Compiler for ALGOL 60. *Communications of the ACM*, 4(1):51–55.

Irons, E. T. (1970). Experience with an Extensible Language. *Communications of the ACM*, 13(1):31–40.

Kay, A., Piumarta, I., Rose, K., Ingalls, D., and Amelang, D. (2008). STEPS Toward The Reinvention of Programming: 2008 Progress Report Submitted to the National Science Foundation (NSF): VPRI Technical Report TR-2008-004. Technical report, Viewpoints Research Institute.

Kay, A. C. (1993). The Early History of Smalltalk. In *HOPL Preprints*, pages 69–95.

Kernighan, B. W. and Ritchie, D. M. (2008). *The C programming language*. Prentice Hall PTR, Upper Saddle River, NJ, 2nd ed., 43. print. edition.

McCarthy, J. (1960). Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195.

OASIS (2007). Web Services Business Process Execution Language Version 2.0: OASIS Standard. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.

Simonyi, C., Christerson, M., and Clifford, S. (2006). Intentional software. In Tarr, P. L. and Cook, W. R., editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006)*, pages 451–464.

Steele, G. L. (1984). *COMMON LISP: The Language*. Digital Press, 12 Crosby Drive, Bedford, MA 01730, USA.

van Deursen, A., Klint, P., and Visser, J. (2000). Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35:26–36.

Ward, M. P. (1994). Language-Oriented Programming. *Software - Concepts and Tools*, 15(4):147–161.

World Wide Web Consortium (W3C) (2010). Semantic Web: http://www.w3.org/standards/semanticweb/, last visited 30/4/2010.