Comparing Complexity of API Designs: An Exploratory Experiment on DSL-based Framework Integration

Stefan Sobernig
Institute for IS and New Media
WU Vienna, Vienna, Austria
stefan.sobernig@wu.ac.at

Patrick Gaubatz
Software Architecture Group
University of Vienna, Vienna, Austria
patrick.gaubatz@univie.ac.at

Mark Strembeck
Institute for IS and New Media
WU Vienna, Vienna, Austria
mark.strembeck@wu.ac.at

Uwe Zdun
Software Architecture Group
University of Vienna, Vienna, Austria
uwe.zdun@univie.ac.at

Abstract

Embedded, textual DSLs are often provided as an API wrapped around object-oriented application frameworks to ease framework integration. While literature presents claims that DSL-based application development is beneficial, empirical evidence for this is rare. We present the results of an experiment comparing the complexity of three different object-oriented framework APIs and an embedded, textual DSL. For this comparative experiment, we implemented the same, non-trivial application scenario using these four different APIs. Then, we performed an Object-Points (OP) analysis, yielding indicators for the API complexity specific to each API variant. The main observation for our experiment is that the embedded, textual DSL incurs the smallest API complexity. Although the results are exploratory, as well as limited to the given application scenario and a single embedded DSL, our findings can direct future empirical work. The experiment design is applicable for similar API design evaluations.

Categories and Subject Descriptors D.2.8 [*Metrics*]: Complexity measures; D.3.2 [*Language Classifications*]: Specialized application languages; D.1.5 [*Object-oriented Programming*]

General Terms Design, Experimentation, Measurement

Keywords Domain-Specific Language, Application Programming Interface, Complexity, Object Points

1. Introduction

Designing and implementing application programming interfaces (APIs) for reusable software components, such as class libraries, object-oriented (OO) application frameworks, or container frameworks, is a process of making critical decisions at the design level and at the implementation level. Decisions include, among others [19], the adoption of certain architectural patterns (e.g., inversion-of-control layer), the choice between composition- or inheritance-based integration, and parametrization techniques (e.g., argument passing strategies). These decisions affect the quality attributes of applications constructed from the resulting APIs. This is because application engineers, who use APIs to help develop their applications, are influenced by the API's complexity (e.g., the levels of expressiveness in terms of program sorts supported) and the API's perceived usability (e.g., kinds of error prevention).

In this paper, we consider constructing APIs by providing an *embedded textual* domain-specific language (DSL, [13, 18]) on top of OO frameworks. DSLs are special-purpose programming languages engineered for a particular problem or application domain. An *embedded* (or internal) DSL extends and widely integrates with its host language (e.g., Java, Ruby, XOTcl) by reusing the host language's syntactic and behavioural models. Embedded DSLs as language extensions typically inherit those characteristics from their host languages [18]. Textual DSLs offer a textual concrete syntax to its users.

DSLs are claimed to enable domain experts to understand, test, co-develop, and maintain DSL programs. In its role as an API, a DSL targets domain experts as the API users. For them, a DSL-based API wraps framework functionality using domain abstractions. Depending on the DSL's language model, framework functionality is sometimes made composable through declarative specifications. A DSL might also provide alternative execution and parametrization modes, adding to or substituting those offered by the host language.

The use of DSLs as APIs is motivated by the belief that DSLs have, for the most part, a positive impact on the quality attributes of the resulting DSL programs. In particular, their comprehensibility, maintainability, communicability, and reusability are said to be positively affected (see, e.g., [7–9, 20]). To give an example, DSLs are commonly judged as more *expressive* than a general-purpose programming language. This augmented expressiveness [20] is explained by DSLs exposing a comparatively small number of high-level domain abstractions as first-class language elements. Expressiveness is also said to be positively influenced by providing a predominantly declarative concrete syntax, which is ideally extracted from the respective application domain (e.g., some DSLs use a tabular notation inspired by spreadsheets). This increased expressive-

GPCE'11, October 22–23, 2011, Portland, Oregon, USA.

ness is expected to facilitate developing and maintaining programs written in DSLs.

Despite such fundamental claims on the benefits of DSLs, empirical evidence is limited. To date, very few studies on DSLs have been conducted [8, 9]. As for qualitative approaches, most data sources are opinion statements and experience reports, collected from domain experts in terms of "industry case studies" (see, e.g., [14]) and from DSL prototype implementations [22]. Details about collecting and processing the evidence are often missing.

A first rigorous, survey-based effort [9] attempts to confirm conjectures about general success factors as perceived by a small developer population using a single DSL. However, these qualitative findings do not touch the issue of DSL-based API complexity. Besides, qualitative findings cannot be traced back to the structural properties of an API at the code level. The few quantitative studies exhibit several limitations: In particular, the research questions cover DSL aspects other than API complexity from the perspective of API users. Important examples are exploratory evaluations of the maintainability property of the DSL implementations themselves [10] and comparing different DSL implementation techniques [11]. Other studies look at domain-specific *modeling* [4, 12], rather then domain-specific programming languages. As for the measurement methods applied, some study designs fall short by primarily measuring source lines of code (LOC; see, e.g., [4, 12, 24]).

This situation motivated us to conduct an exploratory experiment to capture descriptive statistics as indicators of API complexity comparing four different approaches to API design, including an embedded DSL. Guided by the quantitative observations, a qualitative analysis explores the research question whether API users receive the advantage of a reduced API complexity when using a DSL-based API on top of an OO framework, as compared to using alternative APIs (i.e., a class library, an abstract-class, or a container framework). For this experiment, we created four programs realizing the same, predefined, and non-trivial application scenario. This application scenario describes a service provider component for a distributed single-sign-on infrastructure based on the Security Assertion Markup Language (SAML; [16]). Each program, however, is integrated with a different OO framework, with each framework API representing another API design approach. To obtain complexity indicators, we conducted an Object-Points (OP) analysis [17] for the four programs. The qualitative evaluation, based on the initial quantitative observations of this experiment, provides an indication for the potential of DSLs to reduce structural API complexity.

We report on our experiment as follows: In Section 2, we give an introduction to the experiment design. After having introduced important experimental units, the notion of *object points* as an indicator for API complexity is provided. Section 3 adds details about the experiment procedure, including an introduction to the technical domain of the experiment, federated identity management using SAML, and the selected application scenario. First quantitative observations of the experiment are reported in Section 4, before Section 5 critically revisits our findings. After a discussion of related work in Section 6, Section 7 concludes the paper.

2. Experiment Design

In this section, we introduce the necessary terminology (see Figure 1) and give some background on the Object-Points (OP) measurement method. The OP method is exemplified by applying it to a selected detail of our experimental code base.

2.1 Experiment Units

Application programming interface (API) – The interface of a software component to be used by the developers to access the functionality provided by the component. The corresponding functionality is commonly implemented by a set of sub-components. Thus, an API may bundle several component (or object) interfaces. The API provides an abstract view on the possible collaborations of these sub-components. While providing access to public functionality, the API discloses a protocol for integrating the API in client applications. This protocol gives the necessary details about the integration techniques adopted by the API design, e.g., template and hook classes, callback variants, dependency injection, as well as deployment and configuration descriptors. As a result, an API does not only consist of code units (e.g., classes, interfaces, methods), but also of behavioral protocols and of auxiliary data items (e.g., configuration files).

Program – The object-oriented software component which integrates and completes the OO framework (by means of the framework API) to realize the application scenario. In our experiment, we considered the program both in its source-code and language-dependent representation, as well as in a language-independent UML representation for comparing programs written in different programming languages (Java, XOTcl). The UML representation includes class and interaction models. For the actual measurement step, data was gathered on the syntactical structure of the programs in their UML representation. The program implementation sets a concise *working framework* [6], i.e., the feature chunk of the API needed to implement a specific application scenario. By capturing such a single working framework in terms of the program, the measurement results are eligible for an analysis comparing different APIs. For instance, the number and the types of working tasks are stable for a given application scenario.

Structural complexity – The notion of API complexity spans a considerable space of observable API design properties and the API's perceived usability [19] beyond the scope of a single, exploratory experiment. For example, an API's usability is affected by various cognitive components of various API stakeholders (developers, product users), including the abstraction level, learning style, and domain correspondence of API abstractions [6]. To allow observations on API complexity which can be directly linked to properties in the program structure, we limit ourselves to a working definition of structural, syntactic complexity observable from three properties of an object-oriented program [1]. First, the *size* of a program denotes the number of distinct entities (i.e., classes and objects) constituting the program. The interface size indicates the potential complexity resulting from all the possible object compositions used to implement the program. Second, the *interaction level* describes the permissive interactions between the entities (e.g., operation calls between objects). Third, the *parametric complexity* captures the complexity resulting from the number and the types of the members owned by each program entity (i.e., each class with the number of attributes and operation parameters having an object-type). To quantify these structural properties in an integrated measure construct, we apply an Object-Points analysis [17]. This working definition of structural API complexity implies a surrogate measurement: The API complexity is approximated by observing the above mentioned structural properties of a given program, constructed from a working framework of this API.

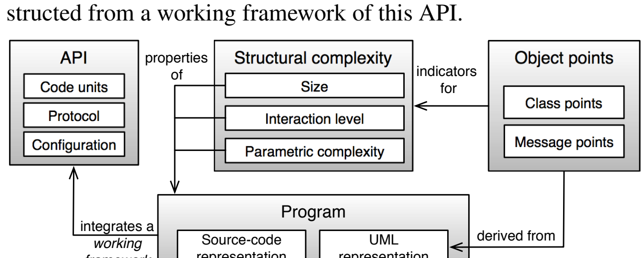


Figure 1: Overview of the Experiment Units

$$CP = \left(W_C \cdot |C| + \sum_{c \in C} |A_c| + W_{R_c} \cdot \sum_{c \in C} |R_c| + W_{O_c} \cdot \sum_{c \in C} |O_c| \right) \cdot \overline{N_C}$$

$C, \|C\| \ldots$	Set of classes, Class count
$\|A_c\| \ldots$	Attribute count per class $c, c \in C$
$\|O_c\| \ldots$	Operation count —"—
$\|R_c\| \ldots$	Relation count —"—

$$N_c = \prod_{s \in Super_c} \frac{|A_c| + |R_c| + |O_c|}{|A_c| + |R_c| + |O_c| + |A_s| + |R_s| + |O_s|}$$

$Super_c \ldots$	Set of directly generalizing classes of class c; for all $s \in Super_c, s \in C, s \neq c$
$N_c \ldots$	Novelty weight of class $c, c \in C$
$\overline{N_C}$	Avg. class novelty, $\frac{\sum_{c \in C} N_c}{\|C\|}$

$$MP = \left(W_{O_M} \cdot |O_M| + \sum_{o \in O_M} |P_o| + W_{S_o} \cdot \sum_{o \in O_M} |S_o| + W_{T_o} \cdot \sum_{o \in O_M} |T_o| \right) \cdot \overline{N_{O_M}}$$

$O_M \subset \bigcup_{c \in C} O_c \ldots$	Set of called operations
$\|O_M\| \ldots$	Called operation counts
$\|P_o\| \ldots$	Parameter count of operation $o, o \in O_M$
$\|S_o\| \ldots$	Source count —"—
$\|T_o\| \ldots$	Target count —"—
$N_o \ldots$	Novelty weight —"— $\in \{1, 0.5\}$
$\overline{N_{O_M}} \ldots$	$\frac{\sum_{o \in O_M} N_o}{\|O_M\|}$

Table 1: Class- and Message-Points Measures

2.2 Object-Points Analysis

For our measurement and data analysis, the structural data gathered from the programs is processed using an Object-Points (OP) analysis. In particular, we apply the OP analysis by Sneed [17], not be confused with object-oriented variants of Function Points (see e.g. [2]). The OP analysis is applied for predicting the development effort (i.e., the program size) based on UML implementation models of a given program.

Note that, in our experiment, the Sneed OP analysis does *not* serve for estimating development effort. We adopted the Sneed OP method to compute indicator measures for reflecting selected structural properties of an object-oriented (OO) program. The objects (and object members) enter the analysis as *counts* (e.g., object counts, or counts of messages exchanged between two objects in a given interaction). Predefined *weights* are then assigned to these selected counts. For our experiment, the weight values are adopted as proposed by Sneed [17]. The weighted counts represent the actual OP value for a given program. See Table 2 for a list of used weights, their value loadings in our experiment, and their explanation. The OP value for a program results from the summation of two intermediate measure values, i.e., the *class* and the *message points*.

The class points (CP) of a program result from the counts and the weights representing the number of classes in the respective program, as well as the structure of these classes (i.e., their attributes, their operations, and their structural relations with other objects). Therefore, the CP measure reflects the program size and the parametric complexity of the analyzed program. In turn, the message points (MP) count the number, the structure, and the uses of operations provided by the objects (i.e., in terms of their signature interface) reflecting the interaction level property of a program. As the programs and the properties (size, interaction level, parametric complexity [1]) act as surrogates of API complexity for the scope of a specific working framework of the entire API, the OP indicators only represent indirect measure constructs of structural API complexity.

When compared to alternative approaches of OO complexity measurement [1], the Sneed OP analysis allows for comparing program structures in different languages. This is because intermediate UML models serve as the data source and weightings can compensate for language-specific statement sizes of code units (e.g., method bodies) while avoiding the methodical issues of LOC measurement. In addition, the Sneed measure constructs capture certain structural dependencies directly as a source of complexity. For example, the message points based on a UML interaction capture the static interaction level of a given application scenario. The Sneed measure constructs also present certain practical advantages. Most importantly, the constructs allow for direct traceability between program structures (e.g., a UML class's record of owned operations) and the indicator values (e.g., the CP value as size indicator) for qualitative evaluations.

Weight	Loading	Description
W_C	4	The value of 4 has been adopted from the standard weight of entities in the Data Points analysis; see [17].
W_{R_c}	2	Also adopted from the standard Data Points method; a weight of 2 reflects the fact that a single relation affects exactly two entities; see [17].
W_{O_c}	3	This weight reflects the average size of methods found for a representative code base in the programming language under investigation. The value 3 is the standard value proposed in [17], expressing a ratio of 15:5 between the avg. method statement size of more high-level (e.g. Smalltalk) and more low-level programming languages (e.g. C++).
W_{O_M}	2	Similarly to W_{R_c}, the weight 2 reflects that a single message occurence involves two entities – the sender and the receiver; taken from [17].
W_{S_o}	2	The standard value adopted from [17].
W_{T_o}	2	The standard value adopted from [17].

Table 2: Weights in the Sneed Object-Points analysis

Class points, CP – The CP measure is calculated from the UML class model. The measure is constructed as an adjusted summation of four summands, each being accompanied by a fixed weighting factor adopted from [17]. The summands are the class count $|C|$, the sum of attribute counts $|A_c|$, the sum of operation counts $|O_c|$, and the sum of relation counts $|R_c|$. The class count $|C|$ is the cardinality of the set of classes in the model and reflects the program size. The class count is corrected for the weight W_C. For each class $c \in C$, counts for owned attributes are established. Note that associations with neighbor classes are also counted as attributes (rather than relations). Depending on the navigability (unidirectional, bidirectional) of an association, it is recorded either for the non-navigable association end or both ends. The count of owned operations is then calculated, with the sum of operation counts for all classes being weighted by W_{O_c}. The weight corrects the count for its statement value, i.e., the average size of an operation implementation in terms of statements. The weighted operation count is therefore an indicator of class size.

The final summand is the sum of relation counts over all classes. In our experiment, relations mean generalization relations to one (or multiple) classes. Generalizations are counted for both the generalizing and the specializing ends, to reflect the two-way dependency introduced by this relationship (e.g., comprehending the feature set of by a specializing class requires the partial study of the generalizing class). The applied weighting factor W_{R_c} with a value of 2 accounts for the fact that a single relation affects exactly two entities.

The summation is then adjusted for the average class novelty $\overline{N_C}$. This factor computes from the product of novelty weights N_c per class (see Table 1). Provided that a class c is related to at least one generalizing class (which is reflected by a relation count $|R_c| > 0$), the novelty is calculated as the normalized ratio between the counts of attributes, operations, and relations *owned by* the specializing class and the total counts of both the specializing as well as the generalizing class. This novelty factor balances between the increased complexity caused by generalization relations (which is captured by a $|R_c| > 0$) and the locality of refinements in the specializing class, with the latter facilitating API elaboration. At the extremes, an N_c value of 0 represents a class without generalization relationships, without ownership of attributes and without operations. $N_c = 1$ means that a class c defines its entire record of attributes, operations, and relations in a freestanding manner; i.e., without inheriting any of these from a generalizing class.

Message points, MP – The MP measure is expressed over data drawn from both the UML class and the UML interaction model of the analyzed program. Most importantly, it is defined over a subset of all operations defined in the class model, i.e. the set of operations O_M actually used in the interaction. This corresponds to all operations referenced by the call events registered with message occurrences in the interaction.

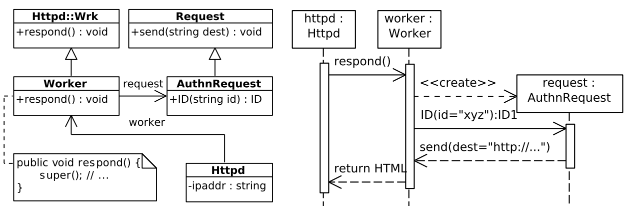
As can be learned from Table 1, and similar to the CP measure, there are four summands based on weighted counts and a general novelty weight. The first summand is the number of operations referenced by message occurrences in the model, multiplied by W_{O_M}. The weight indicates that every message occurance involves two entities – the sender and the receiver. This called operations count $|O_M|$ indicates the *general* level of interaction between the collaborating objects.

The remaining summands are specific to each called operation o. First, the sum of parameter counts $|P_o|$ (including input and output parameters of an operation) is established. This reflects the parametric complexity at all operation call sites. Second, the sources and targets of each called operation in terms of the sending and receiving lifelines of the corresponding messages are collected. This yields the source counts $|S_o|$ and the target counts $|T_o|$, with each receiving a fixed weight (W_{S_o} and W_{T_o}). While the general operation count $|O_M|$ indicates the degree of interaction, the source and target counts stand for the intensity of the given object interactions. Complexity in the static program behavior is thus captured in terms of message occurrences.

For each of the called operations $o \in O_M$, a per-operation novelty weight N_o is applied. Called operations which combine with an operation provided by a generalizing class (also contained in the class model) are adjusted by a weight of 0.5. Replacing (overwriting) operations or newly defined ones enter the MP calculation with their full count values. The per-operation novelty compensates for the repeated inclusion of message points components when computed for two (or more) combinable operations along a generalization hierarchy. At the same time, any $N_o > 0$ reflects that each called operation, whether it refines another operation or not, adds another layer of complexity (e.g., parametric complexity with varying parameter sets). The actual weighting component of the MP measure is the average novelty weight over all called operations $\overline{N_{O_M}}$.

Object points, OP – The aggregate OP value results from summing the two partial results, i.e. the MP and the CP values. While the original definition of the OP measure [17] involves a third summand for expressing the Use Case (UC) complexity (e.g., based on a UML use case model of the underlying application scenario), we can omit this summand in our experiment. This is because in our comparative experiment based on a single application scenario, we take the UC complexity as a constant.

In our experiment, the OP score is used as an absolute value characteristic for each program structure. When contrasting the OP scores of different programs, the range of OP scores offers explicable thresholds for ordering the programs to each other. For instance, in such a range of data points, a relatively higher OP score points to an increased structural API complexity of a given program relative to the others.



(a) Classes (b) Messages

Figure 2: An Exemplary UML Model

2.3 Applying the Sneed OP Analysis

In the following, we give an overview of the Sneed Object-Points (OP) method by looking at an introductory example (see Figure 2). Consider a measurement applied to the simple UML model example given in Figures 2a and 2b. The input data for the actual OP measurement is given in Tables 3a and 3b.

The interaction model identifies three classes as the participants in the abstracted application scenario: `Httpd`, `Httpd::Wrk`, and `AuthnRequest`. As for the class-points (CP) calculation, these constitute the set C. In table 3a, the per-class counts of owned attributes, operations, and relations are depicted, collected from the class model. While data collection on attributes and operations is straightforward for this introductory example, we will now discuss the relationships between classes and the class novelty weightings.

First, table 3a records the generalization relations between `AuthnRequest` and `Request`, as well as `Worker` and `Httpd::Wrk`. The generalizations are marked for both the generalizing and the specializing ends. Second, the uni-directed association `worker` is counted as a second attribute on behalf of the non-navigable association end `Httpd`. Similarly, `request` is counted on behalf of `Worker`. Third, the generalization relations are also reflected in the class novelty ratios $N_{AuthnRequest}$ (0.5) and N_{Worker} (0.6). These weightings are computed as the ratio of non-inherited members of `AuthnRequest` to the total number of members owned by `AuthnRequest` and `Request`: $\frac{2}{4}$. This ratio indicates the extent to which `AuthnRequest`'s parametric complexity is explained by its member structure alone. Using the CP formula from Table 1, we compute the total CP score of our example model: $(4 \cdot 5 + 3 + 2 \cdot 3 + 3 \cdot 4) \cdot \frac{4.1}{5} = 33.62$.

Returning to Figure 2b, the data for deriving the message points (MP) value in Table 3b can be collected from the interaction model. The first observation is that the three message occurrences refer to a set O_M of three distinct operations being called. This simplifies the subsequent counting because all the source and target counts amount to 1. The generalization relationship between the classes `Worker` and `Httpd::Wrk` results in a novelty adjustment for `Worker.respond`, with the operation overloading the

`Httpd::Wrk.respond` operation (see Figure 2b). The total MP score results from the term $(2 \cdot 3 + 3 + 2 \cdot 3 + 2 \cdot 3) \cdot \frac{2.5}{3} = 17.5$. This yields a final OP score of our example model as the sum of the CP and MP values: $33.62 + 17.5 = 51.12$.

$c \in C$	$\lvert A_c \rvert$	$\lvert R_c \rvert$	$\lvert O_c \rvert$	N_c
Httpd	2	0	0	1
Httpd::Wrk	0	1	1	1
Worker	1	0	1	0.6
Request	0	1	1	1
AuthnRequest	0	1	1	0.5
20	3	6	12	0.82
CP		**33.62**		

(a) Class points (CP)

$o \in O_M$	$\lvert P_o \rvert$	$\lvert S_o \rvert$	$\lvert T_o \rvert$	N_o
Worker.respond	0	1	1	0.5
AuthnRequest.ID	2	1	1	1
Request.send	1	1	1	1
6	3	6	6	0.833
MP		**17.5**		

(b) Message points (MP)

Table 3: The CP and MP Scores of the Exemplary Model

3. Experiment Procedure

For the experiment, we selected an application domain and picked a single use case from this domain to be realized by the programs during the experiment (see Section 3.1). In a second step, we investigated existing software components in this application domain. We put special emphasis on adopting software components that provide different API designs (see Section 3.2). Having identified four characteristic software components, the actual experiment involved implementing four programs based on these reusable components (see Section 3.3). In a final step, the resulting programs were documented in terms of UML models (see Section 3.4). Based on these UML representations, the object points for each program were calculated.

3.1 Domain

In the context of computer network security, *digital identities* are created and assigned to *subjects* (e.g., human users). Thus, digital identities allow for identifying different subjects unambiguously. To prove the ownership of a digital identity, a subject has to authenticate. This authentication step involves presenting credentials, such as user name and password pairs. With each subject owning several digital identities (e.g., for different web sites), remembering and managing multiple credentials can becomes tedious task. A *Federated Identity Management* system provides the infrastructure for a so-called single sign-on (SSO) service. In an SSO-scenario, an *Identity Provider* issues a digital identity token for a particular subject. This token is then accepted by a number of *service providers* (also called *relying parties*). Thereby, the digital identity token provides the subject with a *federated identity* that can be used to access different services.

The Security Assertion Markup Language (SAML) [16] provides a well-established XML-based framework that supports creating and operating federated identity management environments. In SAML, different types of assertions can be expressed to provide a subject with corresponding digital tokens. The SAML standard defines a precise syntax for defining assertions and rules for exchanging them. SAML is defined as a flexible framework that can be extended and adapted for a range of use cases, including SSO. The SSO use case is directly supported by SAML's *Web Browser SSO Profile*. The programs realized during the experiment implemented SSO service providers using SAML's HTTP POST binding.

3.2 API Designs

For conducting the experiment, we selected three existing software artifacts for the technical domain of SAML: OpenSAML[1], simpleSAMLphp[2], and JAXB[3]. As we could not identify any prior DSL-based approach, we decided to develop a DSL on top of an underlying infrastructure for XML data binding: xoSAML[4]. Each of these software components realizes a predominant object-oriented API design, that is, a class library, an abstract class framework, a container framework, and an embedded textual DSL.

OpenSAML — This Java component provides a partial domain model for SAML (e.g., SAML messages, assertions), an XML data binding infrastructure tailored towards the needs of SAML (e.g., unmarshalling SAML documents into Java objects and vice versa), and utility classes for realizing SAML protocols (e.g., context objects, message en- and decoders, signature validators). With this, OpenSAML realizes a conventional reuse approach in terms of a *class library*. The classes are packaged into a library namespace, offering themselves for direct reuse (without instantiation protocol) in client programs. For an integration developer, navigating and picking from various concrete class hierarchies is necessary.

simpleSAMLphp — This component written in PHP does not aim at providing support for the entire protocol domain offered by SAML (such as OpenSAML), nor does it create means to generically map between object and XML document representations of SAML artifacts. Rather, simpleSAMLphp limits itself to helping realize preselected scenarios in the SAML/SSO use case. Its reuse strategy is that of a small-scale class library, without imposing any particular integration protocol (e.g., subclassing).

JAXB — The Java-based component delivers a reference implementation for the *Java Architecture for XML Binding* specification and, thus, embodies a generic XML-object binding framework. The XML/object mapping is based on a generator infrastructure, performing transformation from XML Schema descriptions to Java code models. For our experiment, we generated Java representations from the SAML schemas. JAXB also realizes a container framework based on Java beans, managing object lifecycles (e.g., bean factories) and component dependencies explicitly (e.g., dependency injection). In contrast to the other projects, JAXB does not provide any SAML utilities (e.g., message context objects, validators).

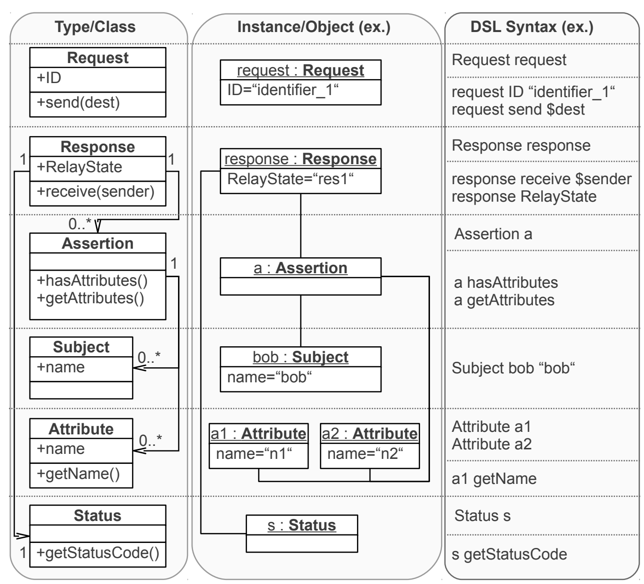


Figure 3: Core Language Model of xoSAML (simplified)

xoSAML — This component represents an embedded, textual DSL for specifying SAML assertions and corresponding SAML request and response messages. xoSAML is implemented in the host language *Extended Object Tcl* (XOTcl), a Tcl-based object-

[1] OpenSAML: `http://opensaml.org/`

[2] simpleSAMLphp `http://simplesamlphp.org/`

[3] JAXB: `http://jaxb.java.net/`

[4] xoSAML: `https://bitbucket.org/pgaubatz/xosaml`

oriented scripting language [15]. xoSAML supports creating, manipulating, and exchanging SAML assertions and messages in a minimized, declarative textual notation. xoSAML integrates with a generator framework for mapping XML schemas to class structures; SAML entities are so provided as an XOTcl class library. Figure 3 depicts some examples that show the different abstraction layers of our DSL. In particular, it shows a (simplified) class diagram of the DSL's language elements and an excerpt of the object diagram for the SSO scenario. Corresponding DSL statements exemplify xoSAML's concrete syntax.

3.3 Application Scenario

Having identified the four reusable software components (OpenSAML, simpleSAMLphp, JAXB, and xoSAML), we continued by implementing the application scenario. We obtained four different implementations of a single-sign-on (SSO) service provider (SP). The actual application scenario is illustrated by Figure 5 in terms of a simplified UML sequence diagram. The sequence diagram shows the messages exchanged for a single SAML authentication request. Initiated by the browser side, a request for accessing a web resource is received by the SP. The SP returns an authentication request which is redirected to the authoritative identity provider. The identity provider verifies the credentials and returns an authentication response. The response contains assertions to be evaluated by the SP to decide about the authorization state and about granting access to resources. The SP implementations, therefore, covers handling the authorization request and evaluating the assertions.

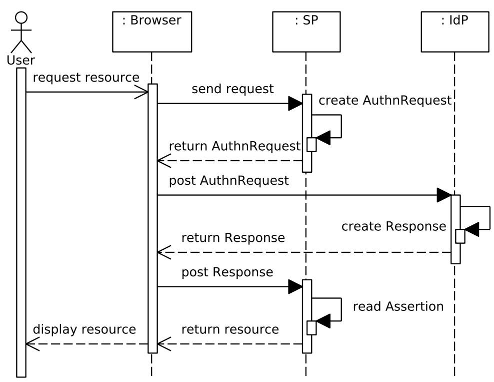


Figure 5: SSO in a Web Context

3.4 UML Implementation Models

The code bases of the resulting programs reflect various API designs and disparate programming languages (i.e., Java, PHP, XOTcl) as well as abstraction levels (e.g., general-purpose vs. domain-specific languages). To obtain language-independent representations of the four programs, each program was documented via UML sequence and UML class diagrams. The UML models were created in a manual design review after having completed the implementation task.

To collect structural data from the relevant working framework [6] of the API examined, we applied selection criteria for establishing the UML models. As for the class model, we selected classes defined by the reused software components (e.g., OpenSAML, xoSAML) which were refined or instantiated by the program. In addition, those declared in the scope of client programs were incorporated. Based on this initial set of classes, we further included classes by traversing the generalization relationships. The traversal was terminated when having collected all the classes that provide a feature accessed by the program (e.g., an attribute or a called operation). By documenting all relevant operation calls, the UML sequences served for identifying the list of classes to consider.

4. Quantitative Analysis

In our experiment, two data sets were created. The first describes the underlying code bases and the intermediate UML representations (see Table 4). The second represents the OP measure computations (see Table 5).[5]

To begin with, the programs realized during the experiment are limited in code size: None of them exceeds 150 *lines of source code* (LOC). The four programs differ in terms of their LOC, as stated by SLOCCount [21]. While the OpenSAML-based implementation amounts to more than 151 lines of Java code, the programs interfacing with xoSAML and JAXB come at a LOC size of 144 and 120 lines, respectively. The smallest code base could be realized using simpleSAMLphp (see Table 4).

Inspecting the UML packages using SDMetrics [23], we learn that the code base complexity is not directly reflected by the program structure. From this structural viewpoint, OpenSAML exhibits the most extensive structural composition. The OpenSAML-based program is built using more than 30 classes with more than 8 operations each, in average. As a package JAXB is slightly smaller, yet structurally comparable with 20 classes and, in average, 11 operations each. xoSAML and simpleSAMLphp fall into a category in terms of class sizes. However, the two programs differ by the amount of operations per class. simpleSAMLphp reports relatively large operation records per class, with each counting an average of 28 operations.

As for inter-class relations $|R|$, the programs built from OpenSAML and from JAXB are characterized by a relatively high number of realized relations (e.g., attributes types, parameter types of owned operations, associations, dependencies). However, in average, each class in JAXB is connected to more neighbor classes (1.45) than an average class in OpenSAML (1). Similarly, the xoSAML and the simpleSAMLphp programs exhibit nearly the same relation counts, however, an average simpleSAMLphp class is linked to one class more than the average xoSAML class (see Table 4).

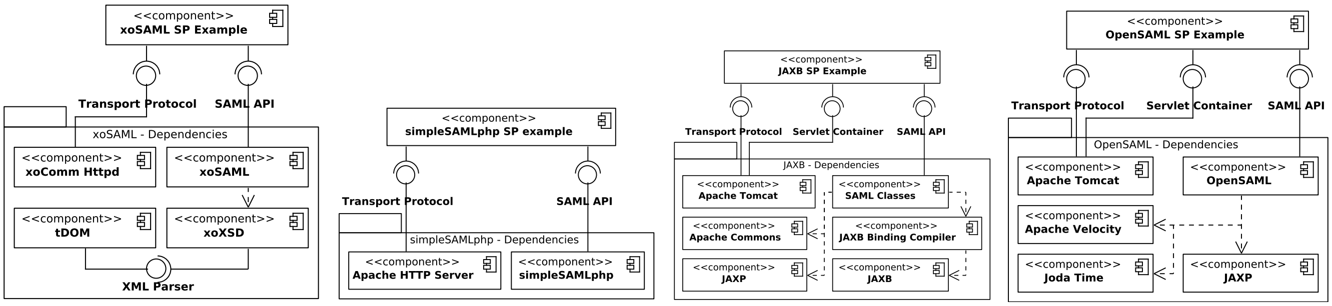
Program using...	OpenSAML	JAXB	simpleSAMLphp	xoSAML
LOC	151	120	48	144
# classes $\lvert C\rvert$	31	20	8	15
# operations $\lvert O\rvert$	254	222	110	37
# relations $\lvert R\rvert$	30	28	15	14
Cohesion $H = \frac{\lvert R\rvert + 1}{\lvert C\rvert}$	1	1.45	2	1

Table 4: Descriptive Statistics on the Code and Model Representations [21, 23]

Table 5 shows the class- (CP), the message- (MP) and the object-points (OP) scores calculated for the four programs examined according to the OP procedure introduced in Section 2.2. The data rows representing each program are sorted by their ascending OP value. Within brackets, the normalized OP scores, adjusted to the lowest OP value, are given. We can make the following observations:

- The program constructed from the xoSAML DSL features the *lowest* OP score.
- The program sizes expressed in LOC (see Table 4), and the LOC-based ranking of the programs, do not relate to the order by OP value.

[5] The XMI representations of the UML models and the preprocessed data collection for the Object-Points analysis are available from `http://swa.univie.ac.at/~patrick/op.zip`. The programs' source code is maintained at `https://bitbucket.org/pgaubatz/xosaml` (see directory `examples/ServiceProvider/`).



(a) xoSAML (b) simpleSAMLphp (c) JAXB (d) OpenSAML

Figure 4: An Overview of Required Components

- The program sizes in terms of the total number of classes $|C|$ and operations $|O|$ follow the pattern of the OP values.
- Despite its OP-based top rank, the xoSAML program does not have the smallest code base.
- The simpleSAMLphp program accounts for an MP value smaller than xoSAML's, while still having a higher OP score.
- The simpleSAMLphp program has a higher OP score than its xoSAML equivalent, while having a code base which is three times smaller than xoSAML's.
- JAXB and OpenSAML have comparably high OP values, with a consistent structure of the CP and MP summands.
- The OP scores of the high-OP programs JAXB/OpenSAML is 3.5 to 4 times the OP value of xoSAML.

	CP ($\overline{N_C}$)	MP ($\overline{N_{O_M}}$)	**OP**
1. xoSAML	179.76 (≈ 0.79)	87 (1)	266.76
2. simpleSAMLphp	270.53 (≈ 0.64)	85.81 (0.89)	356.34
($\triangle$ xoSAML)	*(1.51)*	*(0.99)*	*(1.34)*
3. JAXB	755.46 (≈ 0.98)	188 (1)	943.46
($\triangle$ xoSAML)	*(4.2)*	*(2.1)*	*(3.54)*
4. OpenSAML	860.75 (≈ 0.84)	224 (1)	1084.75
($\triangle$ xoSAML)	*(4.79)*	*(2.57)*	*(4.07)*

Table 5: The OP Analysis Scores

5. Qualitative Analysis

In this section, we revisit our research question about whether API users benefit from a lowered complexity when using an API based on an embedded, textual DSL on top of an OO framework. For this, we review the quantitative observations documented in Section 4. We also point out relations to related work and its limitations. As already stated, the quantitative and qualitative observations below are first and tentative results; no firm recommendations can be derived therefrom.

5.1 Observations

API Complexity and the xoSAML DSL – The embedded, textual DSL (xoSAML) examined for the selected application scenario (SAML service provider) results in the most favorable OP score (266.76) among the API designs compared. This comparatively low OP score reflects that an API user is only exposed to an API feature chunk of low structural complexity for the DSL-based integration: The chunk's size is limited in terms of participating classes ($|C| = 15$) and the smallest number of operations per class ($35/15 \approx 2.5$). The xoSAML API chunk shows a relatively weak connectedness of classes ($H = 1$), resulting from the small number of associations and generalizations between the classes. While this reduced connectedness decreases the level of novelty, xoSAML's average novelty factor $\overline{N_C} = 0.79$ indicates that the API chunk implements a considerable share of the functionality (i.e., operation implementations) used in the application scenario. Taking into account all these components, and given the humble API chunk size in terms of classes, we obtain the smallest class-points (CP) score in the experiment.

This CP score even compensates for a relatively higher object interaction level compared to simpleSAMLphp's one, as hinted at by a message points (MP) score of $87 > 85.51$. The comparable MP scores reflect the structural similarity of the two APIs interaction design in terms of called operation counts $|O_M|$ and the operation's parameter counts ($|P_O|$, max. 2). Besides, xoSAML and simpleSAMLphp, offer a programming model requiring step-by-step calls to clearly separated query-or-command operations (referred to as *command-query APIs* in [7]). This is reflected by source $|S_O|$ and target counts $|T_O|$ of 1 (i.e., there is mostly one message occurrence of a given operation in the call sequence). The slight difference in the OP scores is explained by deviating novelty factors $\overline{N_{O_M}}$ of 1 and 0.89, respectively.

xoSAML DSL vs. PHP Class Library – We would have expected a more substantial OP score difference between the DSL-based program (xoSAML) and the one based on the PHP class library (simpleSAMLphp). However, tracing back the OP scores to the simpleSAMLphp program's structure reveals that simpleSAMLphp, in contrast to the other frameworks, does not require any HTTP-related processing as part of the API feature chunk used. Instead, transport handling is purely delegated to the Apache HTTP request processor. This translates into a comparatively lower OP score for simpleSAMLphp. In addition, the simpleSAMLphp's average class novelty factor balances the high per-class operation counts, with $\overline{N_C} = 0.64$ the lowest in the experiment indicating a larger share of operations than in xoSAML being implemented outside the API chunk.

xoSAML DSL vs. Java OO Frameworks – While the OP analysis indicates that the DSL-based program incurs the least API complexity as approximated by the Sneed OP values, we did not expect the considerable difference of 250% and 300%, respectively, between the DSL-based implementation and the two Java-based programs. The structural complexity of the OpenSAML- and JAXB-based programs results from the multitude of classes (i.e., class counts $|C|$ of 20 and 31, respectively) in the API feature chunk. The class counts reflect that the SAML/XML artifacts are provided as Java classes in a direct mapping. The average size of per-class operation records (with approx. 8 operations each) and the two highest average class novelty factors ($\overline{N_C}$ of 0.84 and 0.98, respectively) also contribute to the considerably higher CP scores (860.75 and 755.46). In addition, there is a parametrization overhead resulting from container and beans management (factories), especially for JAXB.

Expressiveness and API Complexity – Expressiveness is a key dimension of API design [19]. DSL literature considers DSLs beneficial in terms of tailoring the expressiveness for the domain experts as API users. By tailoring, literature often refers to a DSL exposing only a *reduced* set of domain abstractions [20] at a higher abstraction level as compared to the underlying host language abstractions [10]. While our experiment design does not cover the issue of abstraction level, which entails aspects such as domain correspondence [6], some observations hint at a refined notion of expressiveness. While a naive reading might relate expressiveness to the mere number of distinct domain abstractions (e.g., the class count $|C|$), our experiment shows that this is not sufficient. For example, the simpleSAMLphp API chunk with only 8 classes is yet parametrically complex due to the relatively high number of operation counts per class (e.g., an API user must acquaint herself with a complex signature interface). This and similar effects are reflected by the CP measure construct, reporting a higher CP score for simpleSAMLphp (270.53) than for xoSAML (179.76) with 14 classes as domain abstractions.

A related observation is that the JAXB program shows a lower CP score (755.46) than the OpenSAML-based program (860.75). JAXB is a container framework directly operating on SAML/XML artifacts through a generic XML-object mapping infrastructure, while OpenSAML introduces intermediate and aggregating abstractions in terms of a SAML domain model. Hence, we would have expected JAXB to exhibit a CP score higher than the OpenSAML's one. This, however, is not the case (see Table 5). Instead, we found that for the OpenSAML implementation 31 classes were needed (compared to 20 for JAXB). This gives rise to the conjecture that designing a domain model (plus utility classes), rather than providing direct access to the SAML/XML protocol and assertion entities, can be detrimental to the framework integration effort.

LOC Measurement – Related quantitative approaches for comparing DSL and non-DSL programs are based on LOC measurement [4, 12, 24]. The known limitations include a lack of standardized rules for data collection (especially in a multi-language setting), an inverse relationship to notions of labor productivity [5], and major threats to construct validity. For instance, structural complexity (e.g., interaction level) and change locality cannot be captured by LOC-based measures alone.

When contrasting the LOC-based program sizes (see Table 4) and the OP scores of the programs in our experiment (see Table 5), there is no correspondence between the two resulting complexity rankings. In fact, the OP measurement reports the lowest OP score for xoSAML (266.76), while, at the same time, the corresponding LOC size (144) compares to the those of the high-OP programs JAXB (120) and OpenSAML (155). This indication, albeit our experiment design being limited, supports a critique of LOC-based measurement.

5.2 Threats to Validity

This exploratory experiment and its design pose several threats to construct validity, to internal validity, and to external validity. The threats have the potential of making wrong quantitative and qualitative observations.

Construct validity – The Object-Points (OP) measure constructs impose threats to our experiment's validity. This is because a) the Sneed OP approach per se is not used in its originating analysis context, b) the measure constructs (class and message points) have not been confirmed as appropriate means for approximating API complexity by prior empirical work, and c) the choice of weight loadings opens a range of methodical issues.

The Sneed OP analysis was originally calibrated for estimating development effort of large-scale software projects by predicting program sizes from UML models and statement size estimators, established by prior empirical mining in existing code bases (e.g., average method sizes in terms of language statements). Consequently, the comparatively small sizes of our programs might introduce a bias towards an over- or underestimation of structural properties (e.g., the weighted interface size). However, as we compare projects of comparable size in our experiment, the OP measurement for each project would be similarly affected; effectively voiding the negative effects for the comparative analysis. That being said, we cannot rule out that the OP analysis produces distorted results when applied to our small-scale projects of not more than 150 LOC and 30 UML classes.

Closely related to the above threat are the characteristic Object-Points (OP) component weights and the issue of deciding on their value loading for an experiment (e.g., the class W_C, source W_{S_O}, and target weights W_{T_O}; see Section 2.2). In the original application context of the Sneed OP analysis, the weight values are either adopted from closely related measurement instruments (e.g., the Data Points analysis) and/or are mined for a target programming language to establish an empirical predictor, e.g., for the average method size specific to a given language. For our experiment, we adopted the standard values documented for the Sneed OP analysis [17]. While this has practical advantages (e.g., reusing default values established over multiple multi-language projects), it bears the risk of underestimating observations, if the weights are too small, or of overestimating them in the inverse case. As for the standard W_{O_C} value, reflecting the average method size in terms of statements, this caused us a particular tension: Sneed [17] states that this weight should be based on the average size of methods (in terms of language statements) for a particular programming language. For an experiment design, this requires to perform a calibration (i.e. the estimation of the average size of methods). This, however, assumes the availability of a critical number of code bases in the targeted language; a condition not satisfiable for the xoSAML DSL. Hence, we reverted to the default value of 3. To mitigate this general threat, we consulted the available literature on OP to investigate the origin of the standard values. In addition, we exchanged emails with Harry M. Sneed to clarify their justifications. Alas, for W_{S_o} and W_{T_o}, we were not able to recover details about the choice of 2. This undermines attempts of qualitatively evaluating observations based on message points (MP) scores.

Internal validity – The experiment design might have caused our observations not to follow directly from the data collected, i.e., the program structures implemented and processed. The base artifacts of the experiment were four different APIs and four resulting programs, implementing the single sign-on use case. While these four implementations realize the same application scenario (SSO service provider), the power and the completeness of their functionality differ with respect to framework and language specifics. We so risk having compared different functionality, resulting in a misleading measurement. To mitigate this risk, we took care to define a precise application scenario, based upon which the API feature chunk was extracted. In addition, the program implementations, while developed by one author alone, were reviewed by a second repeatedly. Given the small program sizes (i.e., 48-151 LOC), we considered this sufficient, at least for avoiding obvious design and implementation mistakes. Also, when relying on a single developer, there is the risk of introducing a bias due to a continued learning effect between one implementation and another. We strove for containing this threat by developing the four programs in parallel.

A key decision when designing the experiment was that of a surrogate measurement: For the analysis, data is gathered on the syntactical structure of the programs constructed from a specific API. The analyzed program sets a concise *working framework* [6], i.e., the feature chunk of the API needed to implement a specific application scenario. Considering programs integrating a given API chunk as measurement proxy bears a considerable risk, e.g., by the requirement to limit the program to the API chunk and the application scenario. Analyzing the entire API, however could equally introduce a bias because each API varies in terms of the provided features (turning, e.g., into a comparatively increased interface size and a greater interaction level). By observing the interface as used to construct a given program, the resulting measurement is more eligible for a comparative analysis. Most importantly, some factors influencing the perceived complexity [6] can be considered constant. For instance, the number and the type of working tasks to complete is specific to a given scenario.

An important risk follows from the choice of surrogate-based measurement of API complexity over a program structure implementing a concise API feature chunk (i.e., a working framework [6]). This requires the program code to be strictly limited to assembling and configuring services offered by the framework APIs, without introducing code and structural noise due to implementation deficiencies etc. To minimize this threat, we took the following actions. First, API protocol steps (authorization request, assertion handling) were implemented in terms of single operation declarations; or, alternatively, a single method with a switch-threaded control flow. Second, declaring new classes and new operations for the scope of the program was to be limited to a minimum. Ideally, the programs as pure assembly and configuration components should be implemented by a single operation. Exceptions were the requirement to derive final from abstract classes, when expected by the framework API protocol, or implementing operations from required interfaces. While the simpleSAMLphp program did not require a single class declaration, the API design of OpenSAML and JAXB made it necessary to define new classes.

The need for UML representation of the analyzed programs is a further source of possible distortion: On the one hand, the selective inclusion of UML model elements into the OP analysis (see Section 3.4), required to capture the chosen API feature chunk, might result in a inappropriate mapping of source code to model structures. On the other hand, bridging between language and UML abstractions involved critical decisions. For example, in xoSAML's host language XOTcl the equivalent for attributes can be considered both UML operations and UML attributes. The transformation rules, while applied uniformly to all projects, might have introduced an unwanted bias.

Another concern is that one of the four framework APIs, the xoSAML DSL, was designed and implemented by one of the authors because no DSL-based API implementation for the SAML domain had existed to our knowledge. Although we put a lot of effort in implementing a generic facility, the risk remains that we were biased towards a research prototype known to be explored in our experiment.

For the same reason, we might have been biased when evaluating and selecting the Object-Point method for our experiment design, adopting a measurement approach leaning itself towards our DSL implementation. For this initial experiment, we avoided this threat by selecting a standard variant of the OP analysis documented in literature [17]. The Sneed OP variant was picked by senior researchers unaware of the DSL implementation details. The selection criteria were the suitability for measuring diverse API designs, written in different OO languages (XOTcl, Java), and the fit for approximating our working notion of API complexity.

External validity – The generalizability of the experiment results is constrained. Our experiment is limited in its scope and exploratory in nature. As for the scope, the initial findings were drawn from experiences with a single application scenario and with a single embedded, textual DSL. xoSAML is not representative for the considerable variety of embedded DSLs available. While xoSAML shares some common characteristics of embedded DSLs (e.g., a command-query style, use of a method chaining operator) and exposes many general-purpose features of the hosting OO scripting language XOTcl [15], these similarities are not sufficient for the criterion of representativeness. In addition, and as already stated, our working notion of complexity does not cover many relevant dimensions of API complexity. Nevertheless, the experiment is repeatable for DSL-based APIs in other domains.

6. Related Work

In this section, we discuss related work on predominantly quantitative evaluation approaches for DSL-based software projects.

Bettin [4] documents a quantitative study on measuring the potential of domain-specific modeling (DSML) techniques. Using a small-scale example application, Bettin compares different software development approaches: the traditional (without any abstract modeling), the UML-based and the DSML-based approach. Bettin implemented the same example application using every single development approach and compared the development efforts needed. The efforts are measured by counting Lines of Code (LOC) and necessary input cues called Atomic Model Elements (AME; e.g., mouse operations or keyboard strokes). The findings are that the DSML-based approach required the least effort, with the UML-based approach ranking second and so preceding manual modeling.

Zeng et al. [24] introduce the AG DSL for generating dataflow analyzers. Their evaluation approach examines manual and generated creation of AG and corresponding C++ programs, by comparing the program sizes in terms of LOC and by contrasting the execution times. The evaluation results suggest for a single application scenario that the manually-written DSL code is more than ten times smaller than its C++ equivalent.

Merilinna et al. [12] describe an assessment framework for comparing different software development approaches (e.g., DSL-based vs. manual coding). The framework is also applied to DSMLs. The primary unit of measurement in this framework is time; for example, the amount of time needed to learn the DSML, or, the time to implement a specific use case. The approach therefore does not directly cover program structure as source of API complexity.

A DSL-based refactoring of an existing simulator software in the domain of army fire support is presented by Batory et al. [3]. Similarly to our approach, they conducted a quantitative evaluation using class complexity measurement. Therein, the class complexity is quantified by the number of methods, the number of LOC, and the number of tokens/symbols per class. By comparing the class complexities of both the original and the new DSL-based implementation, they state their DSL's positive impact on the program complexity, by reducing it approximately by a factor of two.

Kosar et al. [11] compare ten DSL implementation approaches by looking at indicators of implementation and end-user effort. In contrast to our work, Kosar et al. focus on technical options for DSL implementation, such as source-to-source transformations, macro processing, interpreting vs. compiling generators etc. Kosar et al. address a different aspect than our experiment, namely the DSL implementation itself.

In the direction of evaluating DSL implementation techniques, Klint et al. [10] investigate whether development toolkits for external DSLs (e.g., ANTLR or OMeta) have an observable impact on the maintainability of the resulting DSL implementations. Their study reports on assessing the maintainability of six alternative implementations of the same DSL using different host languages and DSL toolkits. Based on this DSL portfolio, a quantitative measurement is performed to compute an array of structural complexity indicators: volume (number of modules, module members, and module sizes in LOC), cyclomatic complexity, and code duplication ratios. Klint et al. conclude by stating that using DSL toolkits does not necessarily reduce structural complexity (that is, increases a DSL's maintainability), yet DSL generators decrease the need for boilerplate code.

7. Conclusion

Based on a literature review on domain-specific languages (DSLs), as well as our own experiences in building DSLs, we identified the need for exploring quantitative measurement approaches, capable of relating code-level properties of APIs to alleged effects of DSLs on selected quality attributes. Available empirical evidence appears too limited, particularly when expanded into statements on process and business performance: For example, experience reports hint at considerably reduced development times for DSL-based software products, an improved time-to-market, and substantial reductions in development and delivery costs (e.g., for developer or customer trainings; see e.g. [3, 4, 24]).

In this paper, we report first quantitative observations on the structural complexity of programs constructed from different types of APIs, obtained from an exploratory experiment in the domain of federated identity management. We adopted the Sneed Object-Points (OP) analysis [17] for quantifying API complexity specific to different programming interface designs. The OP scores derived from our experiment point towards a reduced API complexity when providing an embedded, textual DSLs as an API. If one accepts the working definition of structural API complexity in this paper and the Sneed OP measure constructs as adequate indicators, this observation is a clear direction for follow-up, confirmatory studies. By thoroughly reporting our experiment design, as well as by providing our raw data[5], this paper enables the reader to reproduce our results and to adopt the experiment procedure for such follow-up experiments.

As future work, we will address the identified threats to validity. That is, we will further explore the usefulness of the OP analysis to evaluate different API design approaches by refining its measure constructs. For this, we will also review alternative measure constructs (as offered by OO complexity measurement [1]) to complement an OP analysis.

Acknowledgments

We gratefully thank Jurgen Vinju for helping improve the paper as a report of empirical research; and Harry M. Sneed for clarifying details about the Object-Points analysis in private communication. Thanks are also due to the anonymous reviewers for their helpful and detailed comments.

References

[1] R. K. Bandi, V. K. Vaishnavi, and D. E. Turk. Predicting Maintenance Performance Using Object-Oriented Design Complexity Metrics. *IEEE Transactions on Software Engineering*, 29:77–87, 2003.

[2] R. D. Banker, R. J. Kauffman, and R. Kumar. An Empirical Test of Object-Based Output Measurement Metrics in a Computer Aided Software Engineering (CASE) Environment. *Journal of Management Information Systems*, 8(3):127–150, 1992.

[3] D. Batory, C. Johnson, B. MacDonald, and D. von Heeder. Achieving Extensibility through Product-Lines and Domain-Specific Languages: A Case Study. *ACM Trans. Softw. Eng. Methodol.*, 11(2):191–214, 2002.

[4] J. Bettin. Measuring the Potential of Domain-Specific Modelling Techniques. In *Proceedings of the 2nd Domain-Specific Modelling Languages Workshop (OOPSLA), Seattle, Washington, USA*, pages 39–44, 2002.

[5] J. Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality*. McGraw-Hill, 3rd edition, 2008.

[6] S. Clarke and C. Becker. Using the Cognitive Dimensions Framework to evaluate the usability of a class library. In *Proceedings of the 15h Workshop of the Psychology of Programming Interest Group (PPIG 2003), Keele, UK*, pages 359–336, 2003.

[7] M. Fowler. *Domain Specific Languages*. The Addison-Wesley Signature Series. Addison-Wesley Professional, 1st edition, 2010.

[8] P. Gabriel, M. Goulão, and V. Amaral. Do Software Languages Engineers Evaluate their Languages? In *Proceedings of the XIII Congreso Iberoamericano en "Software Engineering"*, 2010.

[9] F. Hermans, M. Pinzger, and A. van Deursen. Domain-Specific Languages in Practice: A User Study on the Success Factors. In *Proceedings of the 12th International Conference Model Driven Engineering Languages and Systems (MODELS 2009) Denver, CO, USA, October 4-9, 2009*, volume 5795 of *Lecture Notes in Computer Science*, pages 423–437. Springer, 2009.

[10] P. Klint, T. van der Storm, and J. Vinju. On the Impact of DSL Tools on the Maintainability of Language Implementations. In C. Brabrand and P.-E. Moreau, editors, *Proceedings of Workshop on Language Descriptions, Tools and Applications 2010 (LDTA'10)*, pages 10:1–10:9. ACM, 2010.

[11] T. Kosar, P. M. López, P. Barrientos, and M. Mernik. A preliminary study on various implementation approaches of domain-specific languages. *Information and Software Technology*, 50(5):390–405, 2008.

[12] J. Merilinna and J. Pärssinen. Comparison Between Different Abstraction Level Programming: Experiment Definition and Initial Results. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07), Montréal, Candada*, number TR-38 in Technical Report, Finland, 2007. University of Jyväskylä.

[13] M. Mernik, J. Heering, and A. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[14] MetaCase. Nokia Case Study. Industry experience report, MetaCase, 2007.

[15] G. Neumann and U. Zdun. XOTcl, an Object-Oriented Scripting Language. In *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*, Austin, Texas, USA, 2000.

[16] OASIS. Security Assertion Markup Language (SAML) V2.0 Technical Overview. http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0-cd-02.pdf, 2008.

[17] H. M. Sneed. Estimating the costs of software maintenance tasks. In *Proceedings of the International Conference on Software Maintenance (ICSM'95), Opio (Nice), France, October 17-20, 1995*, pages 168–181. IEEE Computer Society, 1995.

[18] M. Strembeck and U. Zdun. An Approach for the Systematic Development of Domain-Specific Languages. *Software: Practice and Experience*, 39(15):1253–1292, 2009.

[19] J. Stylos and B. A. Myers. Mapping the Space of API Design Decisions. In *2007 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007), 23-27 September 2007, Coeur d'Alene, Idaho, USA*, pages 50–60. IEEE Computer Society, 2007.

[20] A. van Deursen and P. Klint. Little languages: Little Maintenance? *Journal of Software Maintenance*, 10(2):75–92, 1998.

[21] D. A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`, last accessed: October 14, 2008.

[22] D. Wile. Lessons learned from real DSL experiments. *Science of Computer Programming*, 51(3):265–290, 2004.

[23] J. Wüst. SDMetrics. `http://sdmetrics.com/`, last accessed: May 27, 2011, 2011.

[24] J. Zeng, C. Mitchell, and S. A. Edwards. A Domain-Specific Language for Generating Dataflow Analyzers. *Electronic Notes in Theoretical Computer Science*, 164(2):103–119, 2006.