Software Evolution to Domain-Specific Languages

Stefan Fehrenbach[1], Sebastian Erdweg[2], and Klaus Ostermann[1]

[1] University of Marburg, Germany
[2] TU Darmstadt, Germany

Abstract. Domain-specific languages (DSLs) can improve software maintainability due to less verbose syntax, avoidance of boilerplate code, more accurate static analysis, and domain-specific tool support. However, most existing applications cannot capitalise on these benefits because they were not designed to use DSLs, and rewriting large existing applications from scratch is infeasible. We propose a process for evolving existing software to use embedded DSLs based on modular definitions and applications of syntactic sugar as provided by the extensible programming language SugarJ. Our process is incremental along two dimensions: A developer can add support for another DSL as library, and a developer can refactor more code to use the syntax, static analysis, and tooling of a DSL. Importantly, the application remains executable at all times and no complete rewrite is necessary. We evaluate our process by incrementally evolving the Java Pet Store and a deliberately small part of the Eclipse IDE to use language support for field-accessors, JPQL, XML, and XML Schema. To help maintainers to locate Java code that would benefit from using DSLs, we developed a tool that analyses the definition of a DSL to derive patterns of Java code that could be represented with a high-level abstraction of the DSL instead.

1 Introduction

Language-oriented programming [6,14,29] is the idea of decomposing large software systems into *domain-specific languages* (DSLs), which narrow the gap between the requirements of a software system and the implementation of these requirements. Examples of DSLs are state machines for behavioural modelling, XML for data serialisation, SQL for data querying, or BNF for parsing. According to language-oriented programming, a software system should be written in a combination of many existing DSLs and, possibly, newly designed languages specific to the application.

The ultimate goal of language-oriented programming is increased productivity and reduced maintenance effort [17]. DSLs address software maintenance from four directions. First, domain-specific *syntax* reduces the representational boilerplate associated with encoding domain concerns using regular programming constructs and allows developers to focus on the domain-relevant aspects of a program. Thus, DSLs improve understandability and modifiability of source code.

Second, domain-specific *static analysis* enables the encoding of domain invariants and compile-time detection of any violation. In contrast, if encoding domain concerns with regular programming constructs, errors are often only detectable through testing. For example, when dynamically generating an XML document through concatenating strings or calling an API such as JDOM, the validity of generated XML documents cannot be statically guaranteed. An explicit representation of the XML DSL enables a static analysis to guarantee that an XML document adheres to its schema in all possible runs of a program. In case of a violation, an analysis issues domain-specific error messages, which help programmers understand the problem. Consequently, DSLs improve the static safety and understandability of source code.

Third, domain-specific *semantics* abstract over recurring patterns found in the encoding of domain concerns, such as the application of string concatenation or calling conventions. For example, a domain-specific language can ensure proper escaping of injected code to prevent injection attacks, not relying on manually called escape commands. Since a DSL specifies the semantics of domain concerns once and for all, changes to the behaviour of domain concerns are local to the DSL definition and separate from DSL programs. This separation of concerns improves modularity and modifiability of source code, and allows programmers to focus on domain-relevant aspects instead of their encoding.

Fourth, domain-specific *editor support* communicates domain knowledge from the language implementation to developers: from domain-specific syntax highlighting to domain-specific content completion, editor support improves understandability and modifiability of source code. In summary, DSLs improve the quality and thereby the maintainability of source code.

Today, the vast majority of software systems are not designed in a language-oriented fashion. Instead, the long-standing success of C, C++ and Java has led to large procedural and object-oriented systems. The closest many of these applications come to making the best use of DSLs, is containing strings of SQL for database queries. As a consequence, those applications do not benefit from the maintenance advantages that DSLs provide.

Unfortunately, existing literature on language-oriented programming does not address existing code bases, but promotes methodologies useful only when employing them in the original design of an application [29,6,2]. Therefore, to introduce DSLs and their benefits into an existing application, we would have to rewrite the application from scratch. However, rewriting large parts of realistic applications all at once is infeasible [20].

Evolving existing Java applications to use DSLs requires a process that allows for adding new language extensions and incrementally adapting existing code to use them. In regular Java programming, libraries fulfil this role. On the one hand, they extend the standard library with new classes. On the other hand, they need to be imported explicitly in every file they are used, allowing existing code to coexist unmodified with new libraries and adapted code.

Embedded DSLs [17] allow for incremental introduction of DSLs using libraries. However, their flexibility and power is limited by the general flexibility of the

host language. The rigid syntax, type system and missing metaprogramming facilities of Java limit the applicability of embedded DSLs [21]. In particular, the maintenance benefits of concrete syntax with domain-specific editor support and advanced static analyses like XML Schema validation cannot be achieved with embedded DSLs.

We propose a solution for evolving existing code bases to DSLs based on library-based embedding of DSLs in extensible host languages. A sufficiently extensible host language avoids the disadvantages of regular embedding (rigid syntax, no domain-specific editors or static analyses) by extending the corresponding facilities of the host language. Specifically, we use our previous work on the Java-based extensible programming language SugarJ [8,10,11], which permits DSLs as libraries of Java that can define domain-specific syntax, semantics, analyses and editor support. In our prior work on SugarJ, we focused on the expressiveness that SugarJ provides in building new applications, much like other works on language-oriented programming. For this paper, we extended the SugarJ compiler to handle code bases that consist of standard Java source files and *jar* files, and SugarJ source files that employ language extensions. Thus, the SugarJ compiler supports a mix of unchanged legacy code and adapted code that uses DSLs.

To assist maintainers in identifying code locations in an existing software systems where a DSL is applicable, we developed a tool that analyses a DSL definition to extract a pattern that represents the code generated from the DSL. We match this pattern against existing source code to find potential application sites for the DSL and to guide maintainers.

In summary, this work makes the following contributions:

- We explain why incremental introduction of DSLs is a necessary requirement for the evolution of software systems to DSLs. We show that SugarJ is a framework that supports incremental introduction of DSLs. In particular, SugarJ organises DSLs as syntactic sugar in libraries and thereby supports adding DSLs and adapting applications incrementally.
- To demonstrate the applicability of incremental introduction of DSLs in extensible languages, we reengineered Sun Microsystem's Java Pet Store, which "is the reference application for building Ajax web applications on Java Enterprise Edition 5 platform" [22]. We incrementally introduced four DSLs into the Java Pet Store, in particular, for field-accessor generation, data serialisation (XML), static XML Schema validation (which reveals what appears to be a bug in the Java Pet Store), and data querying (JPQL). The Java Pet Store remains executable throughout our maintenance activity.[3]
- We demonstrate the scalability of incremental introduction of DSLs to large applications with a partially reengineered Eclipse code base.
- We extended SugarJ to support a mix of SugarJ and original Java files to facilitate its use in a large code base. Previously, one would have had to change every Java file to a SugarJ file before using SugarJ.

[3] The source code of the reengineered Java Pet Store including all DSL definitions is available at `http://sugarj.org`.

– We explore SugarJ's self-adaptable DSL mechanism to support the reuse of existing language definitions that occur, for example, in documentation.
– We developed an analysis that finds source-code locations in a software system at which a given DSL is applicable.

2 Problem Statement and Proposed Solution

Domain-specific languages have several advantages over general-purpose languages that influence the maintainability of programs [2,23,14,17,29]: They reduce syntactic boilerplate, enforce domain invariants, abstract over recurring patterns, and provide domain-specific tool support. Unfortunately, the majority of applications is not written in a language-oriented fashion, despite these well-known benefits for software maintenance. Their size makes rewriting them from scratch infeasible [20]. Nevertheless, these applications are still evolving and important to their users.

Since software evolution is inherently incremental, any process for introducing DSLs into existing applications must support incremental application along two dimensions: (i) adding support for more DSLs and (ii) converting more code to use the supported DSLs.

2.1 First Dimension: Support more DSLs

Most applications need to deal with multiple domains, and over their lifetime the number of different domains is only going to increase. For applications developed in a language-oriented fashion, specific domains that would benefit from DSLs are identified in an initial design phase. For existing applications, potential domains for improvement through DSLs are usually only identified while performing a maintenance task. For example, imagine a programmer needs to understand and modify the code shown in Figure 1(a) because the serialisation format requires change. This code is a literate excerpt from the Java Pet Store [22] and serialises an item to its XML representation. It uses string literals to represent the static parts of the resulting XML document, that is, element names such as "<item>" or "<price>". Dynamic values like the item's ID are concatenated in between the static element names and the document tree as a whole is assembled through calls to a StringBuffer's append method.

This representation of XML documents as strings is common but has several weaknesses. First, it is hard to read due to string concatenation, character escaping, and the interspersion with calls to append. All of this is boilerplate code that has nothing to do with XML. Second, the structure of the code does not reflect the structure of the XML document. For example, to add an element currency as child of price, we have to disassemble the string describing price to inject the currency at the right place. Third, the encoding is unsafe because domain invariants are not enforced. For example, XML documents must be well-formed, that is, start and end tags must match. In the string encoding, this invariant is not explicitly stated, let alone statically enforced. An ill-formed document

```
private String handleItem(String targetId) {
  Item i = cf.getItem(targetId);
  StringBuffer sb = new StringBuffer();
  sb.append("<item>\n");
  sb.append("  <id>" + i.getItemID()
            + "</id>\n");
  sb.append("  <price>" +
    NumberFormat.getCurrencyInstance(Locale.US)
                .format(i.getPrice())
            + "</price>\n");
  [...]
  sb.append("</item>\n");
  return sb.toString();
}
```

(a) String-encoded XML document from the Java Pet Store.

```
import sugar.Xml;

private String handleItem(String targetId) {
  Item i = cf.getItem(targetId);
  return <item>
           <id>${i.getItemID()}</id>
           <price>${NumberFormat
             .getCurrencyInstance(Locale.US)
             .format(i.getPrice())}
           </price>
           [...]
         </item>;
}
```

(b) Semantically equivalent to (a) but uses XML language support.

Fig. 1. Embedded XML using string embedding and language support.

will lead to a runtime error. Fourth, string concatenation is semantically unsafe because it allows for injection attacks that can only be prevented by passing each concatenation argument to an escaping function, which is a global refactoring. Fifth, the string encoding inhibits domain-specific tool support such as syntax colouring; all strings appear the same. For all these reasons, data serialisation with string-encoded XML is a problem domain in the Java Pet Store. There are many others, we address some of them in Section 4.

A DSL-based solution to the problems with XML could look like the code in Figure 1 (b). In general, a DSL should avoid string embedding and instead provide a higher level of abstraction. A more abstract representation that actually represents the inherent structure of the domain also allows for static analysis and dynamic checking. In XML we want to reject documents that do not adhere to a given schema. In XML and SQL we want to prevent the injection of unsafe Java runtime values into documents or queries.

Note that it is not sufficient to merely identify all domains for a single application and use a language that supports them all. For example, Scala has built-in support for XML which addresses some of the problems mentioned, but consider a new browser-based front-end that requires JSON serialisation. Continuous software evolution requires adding language support for new domains.

2.2 Second Dimension: Convert more Code

Having language support for domain-specific problems is nice. Unfortunately, existing code does not immediately benefit from such support. It has to be converted from the original domain-unspecific encoding (such as string concatenation) to the DSL (using the domain-specific syntax). However, it is undesirable to require maintainers to locate all possible application sites of a DSL at once.

Likely, the maintainer of a code base would want to convert code to an existing DSL at an opportune moment. For example, there might be a bug report claiming a missing element in an XML document that so far went unnoticed. To address

this issue, a maintainer might first refactor the relevant code to use an XML DSL with static validation against XML Schema, and then fix the resulting XML Schema compile-time error.

We need some modularity guarantees to achieve both dimensions of incrementality. Adding language support for a domain should not affect any existing code immediately. Reengineered code should activate language extensions explicitly. Also, reengineered code needs to coexist, or even better cooperate and coevolve, with unchanged code.

2.3 Proposed Solution

We propose the following process for the evolution of an application to use DSLs. First, choose a problem domain. Existing DSLs with weak embeddings, such string embeddings of XML or SQL, are an obvious target but there are likely other domains that can be improved with language support. Second, design and implement a DSL as *syntactic sugar*, enriched with domain-specific static analysis and tool support, in SugarJ [11]. Third, use SugarJ to *modularly activate* the new DSL in some source files and incrementally rewrite code to use the DSL.

To scale our process to the evolution of large existing applications, we designed it around syntactic sugar that is modularly activated. Syntactic sugar is semantically transparent. Therefore, we achieve cooperation and coevolution of old and new or reengineered code, because both remain semantically compatible and thus interoperable at all times. This allows for incremental rewriting of large code bases.

Modular activation of DSLs means that DSLs must be activated explicitly per source file. Conversely, a DSL definition can only affect those source files that activate the DSL. This is important for software evolution of large code bases, because it gives maintainers the guarantee that DSLs have no effect for files left unchanged by the maintainer. This way a single project can use multiple conflicting DSLs in different parts of the code, which would be impossible in tools that require global activation of DSLs. Moreover, in contrast to global build-script based DSL activation, modular activation of DSLs retains the incremental compilation character of Java, so that only affected source files require recompilation.

We propose to use SugarJ [11] for the implementation of DSLs as syntactic sugar that is modularly activated. SugarJ organises DSL definitions in regular Java libraries so that regular Java import statements activate DSLs in a source file. It is easy to add new DSLs as libraries, and it is even possible to use multiple DSLs in a single file by importing all corresponding libraries.

3 Background: DSL Development with SugarJ

SugarJ is an extensible programming language based on Java that supports library-based language extension [8,11]. SugarJ and its IDE [10] make all aspects of Java extensible: syntax, semantics, static analysis and tool support. In particular,

```
package sugar;

import sugar.XmlSyntax;
import org.sugarj.languages.Java;
import concretesyntax.Java;

public extension Xml {
  context-free syntax
    Document -> JavaExpr {cons("XMLExpr")}
    "$" "{" JavaExpr "}" -> Element {cons("JavaEscape")}

  desugarings
    desugar-xml
  rules
    desugar-xml :
      XMLExpr(doc) ->
      |[ String.format(~xml-string, ~java-escapes) ]|
    where <xml-to-string> doc => xml-string;
          <xml-java-escapes> doc => java-escapes
    xml-to-string : ...
    xml-java-escapes : ...

    constraint-error :
      Element(lname, attrs, content, rname) ->
      [(lname, "element start and end tag need to coincide"),
       (rname, "element start and end tag need to coincide")]
      where <not(equal)> (lname, rname)

  colorer
    ElemName : blue (recursive)
    AttrName : darkorange (recursive)
    AttValue : darkred (recursive)
    CharData : black (recursive)
  folding
    Element
}
```

Fig. 2. Definition of the XML DSL in SugarJ.

SugarJ's extensibility is useful for embedding DSLs into Java [11]. In this section, we exemplify the development of a DSL with SugarJ using the XML DSL presented in the previous section.

SugarJ organises language extensions as regular Java libraries that, instead of a Java class or interface, define an extension with custom syntax, static analyses and tool support for a DSL. Figure 2 displays the SugarJ language extension that defines the XML DSL. Figure 1 (b) already showed how to use this extension, namely by importing the corresponding library sugar.Xml. SugarJ supports extension compositions [9], which is triggered by importing multiple extensions into a single scope. We explain the implementation of the different language aspects in turn.

Syntax. To define extended syntax, we employ the grammar formalism SDF [25] and write productions in a **context-free syntax** block. For example, the first production in Figure 2 declares that any valid syntax for the Document nonterminal is also valid syntax for the JavaExpr nonterminal. The second production enables writing a Java expression wrapped in **$**{...} in place of an XML element. Additionally, a production specifies the name of the corresponding node in the abstract syntax tree with a cons annotation.

We use the JavaExpr, Document and Element nonterminals to integrate XML syntax into Java syntax and Java expressions into XML. These nonterminals stem from the Java and XML base grammars defined in org.sugarj.languages.Java and sugar.XmlSyntax. We use import statements to bring the Java and XML syntax definitions into scope of the sugar.Xml library.

Semantics. The semantics of a DSL is given as a transformation from the extended syntax into SugarJ base syntax. In line with the notion of syntactic sugar, we call such a transformation a desugaring and use the Stratego transformation system [28] to implement it. A DSL defines transformations in a **rules** block. Each transformation has a name (before the colon), pattern-matches an abstract syntax tree (left-hand side of arrow) and produces another abstract syntax tree (right-hand side of arrow). Since the generation of abstract syntax trees is tedious for complex languages such as Java, we use concrete Java syntax within |[...]| for code generation [26]. To this end, we import the concretesyntax.Java library, which extends Stratego with support for concrete syntax [11].

The desugaring for XML matches on an XMLExpr node and transforms it into Java code that calls the String.format method of the standard Java library. Within concrete syntax, the ~ symbol allows us to escape back to Stratego code. In particular, we compute the arguments of String.format by applying the xml-to-string and xml-java-escapes transformations (definitions elided for brevity) to the embedded XML document. The former transformation pretty-prints the XML document and inserts a placeholder %s for each escape to Java. The latter transformation extracts the Java code from the XML document. Importantly, the string that results from the generated String.format invocation is semantically equivalent to the original string encoding; the DSL only provides syntactic sugar.

Static analysis. SugarJ represents static analyses as program transformations that transform the program under analysis into a list of errors. To this end, a programmer can define a special-purpose transformation named constraint-error. For XML, we have defined a static analysis that matches on XML elements and produces errors in case the start and end tag of the element differ. Accordingly, this static analysis verifies the domain invariant that embedded XML documents are well-formed. In case of an ill-formed document, our IDE uses the syntax tree that amends an error message (lname and rname in our example) to determine the position for displaying the domain-specific error message to the user.

Editor services. Finally, our SugarJ IDE enables domain-specific editor services such as syntax colouring, code folding, code completion, or reference resolution [10]. We provide an Eclipse plugin based on the Spoofax language workbench [18]. In Figure 2, we have defined XML-specific syntax colouring and code folding.

In summary, we defined the XML DSL from the previous section in SugarJ as a language extension: We provide a syntactic extension to integrate the domain syntax and semantics, use program transformations to encode domain invariants as static analyses and leverage the extensibility of our IDE plugin to support domain-specific editor services.

4 Evolution to DSLs in Practice

We conducted two case studies to gather experience with applicability of DSLs in existing software systems and to confirm the applicability of SugarJ for incrementally evolving an existing software system to use DSLs.

Our first case study is based on the Java Pet Store [22], an interactive web application developed by Sun Microsystems as a reference application for Java Enterprise Edition. Following the process proposed in Section 2.3, we incrementally identified four problem domains and designed and implemented corresponding DSLs in SugarJ. We used these DSLs to incrementally reengineer part of the Java Pet Store to improve subsequent maintainability; the code remained executable at all times. This case study shows that SugarJ enables evolution of an existing software system to use DSLs.

As a second case study, we reengineered a deliberately small part of the implementation of the Eclipse IDE to use DSLs. For the Eclipse IDE, it is essential to retain modular reasoning, which allows developers to assume local effects for local changes. In particular, a local improvement through a DSL should not affect other code. In SugarJ this is witnessed through Java-style separate compilation: Source files that are compiled separately cannot influence each others meanings. This case study shows that SugarJ enables local and small-scale evolution in large software systems.

4.1 Java Pet Store

First Iteration: XML As an interactive web application, the Java Pet Store makes use of Ajax technologies for data exchange between server and browser. In particular, it uses string-embedded XML for data serialisation. In Section 2.3, we already discussed the drawbacks of the string embedding of XML and designed a DSL, which integrates XML documents into Java more directly as syntactic sugar. The implementation of the XML DSL as a SugarJ library was illustrated in the previous section.

Within the Java Pet Store, use of XML is cross-cutting multiple classes and methods. In total, we reengineered 59 lines of legacy XML code to use the XML DSL. Our syntactic integration of XML and static analysis for well-formedness did not reveal any bugs in the original code, but increase confidence in its correctness.
Second Iteration: Field-Accessor Declarations

The second problem domain we identified are the getter and setter methods that clutter the code of the Java Pet Store, following the *JavaBeans* standard. The resulting amount of accessor methods is considerable and all of it is dispensable boilerplate. For example, consider the class definition shown in Figure 3 (a), which is a literate but shortened excerpt from the Java Pet Store. This class models a product with five properties: a product ID, an associated category ID, a name, a description and a URL to some image. These properties are private fields behind public getters and setters, and initialised by a constructor. The only nontrivial aspect of this class is the `@Id` annotation on the `productID` getter,

```
public class Product {
  private String productID, categoryID, [...];
  public Product(String productID,
                 String categoryID, [...])
  { this.productID = productID;
    this.categoryID = categoryID; }
  public String getCategoryID(){return categoryID;}
  public void setCategoryID(String categoryID)
  { this.categoryID = categoryID; }
  @Id
  public String getProductID(){return productID;}
  public void setProductID(String productID)
  { this.productID = productID; }
  [...]
}
```

(a) Java class definition from the Java Pet Store.

```
import sugar.Accessors;

public class Product {
  private String productID {set; con};
  private String categoryID, description,
                 name, imageURL {get; set; con};

  @Id
  public String getProductID() {
    return productID;
  }
}
```

(b) Reengineered class definition using the field-accessor DSL.

Fig. 3. The original Product class has 5 properties with corresponding accessors.

which marks it as a primary key for the object-relational mapping employed by the Java Pet Store.

The main problem with this class definition is the large amount of boilerplate code. Modern Java IDEs try to address this by automatically generating getters and setters. However, this is insufficient because it does not solve the maintainability issue. For a maintainer who reads such code, it is not immediately clear what fields are truly private, publicly readable, or publicly readable and writable. Furthermore, actual application-specific code that deviates from the standard template for accessors is masked by the large amount of boilerplate code. For instance, the actually interesting `@Id` annotation of the productID getter is easily overlooked in a file consisting mostly of getter, setter, and constructor boilerplate code.

Based on these observations, we designed a DSL that abstracts over the boilerplate associated with field accessors. In our DSL, programmers declare the desired accessors instead of implementing them. The unshortened reengineered class definition from Figure 3 (a) is shown in Figure 3 (b). The syntax is inspired by C#'s syntax for properties. The annotations **get** and **set** declare getters and setters respectively, **con** makes a field part of the initialising constructor. To achieve compatibility with existing code, the new annotations desugar to usual field-accessor method implementations. Therefore, we were able to apply the field-accessor DSL locally in some files without affecting others.

Third Iteration: JPQL The third problem domain we identified in the Java Pet Store is its string-based embedding of the Java Persistence Query Language (JPQL) used to query databases. Figure 4 shows a JPQL query from the Java Pet Store. The string encoding of JPQL shares many of the problems we previously saw in the XML example, but its handling of dynamic data is much better: Within a query, a programmer can use a parameter (identifier prefixed by a colon) in place of a regular JPQL expression. After processing the query string into a `Query` object, the programmer calls the query's `setParameter` method to provide

```
public List<Item> getItemsByCategoryByRadiusVLH(...) {
    Query query = em.createQuery(
     "SELECT i " +
     "FROM Item i, Product p " +
     "WHERE i.productID=p.productID " +
     "AND p.categoryID = :categoryID " +
     "AND((i.address.latitude BETWEEN :fromLatitude AND :toLatitude) " +
     "AND (i.address.longitude BETWEEN :fromLongitude AND :toLongitude )) " +
     "AND i.disabled = 0 " +
     "ORDER BY i.name");
    query.setParameter("categoryID",catID);
    query.setParameter("fromLatitude",fromLat);
    query.setParameter("toLatitude",toLat);
    query.setParameter("fromLongitude",fromLong);
    query.setParameter("toLongitude",toLong);
    return query.getResultList();
}
```

Fig. 4. JPQL query from the Java Pet Store.

```
import sugar.JPQL;

public List<Item> getItemsByCategoryByRadiusVLH(...) {
    Query query =
     em.SELECT i
          FROM Item i, Product p
          WHERE i.productID = p.productID
            AND p.categoryID = :catID
            AND i.address.latitude BETWEEN :fromLat AND :toLat
            AND i.address.longitude BETWEEN :fromLong AND :toLong
            AND i.disabled = 0
          ORDER BY i.name;
    return query.getResultList();
}
```

Fig. 5. JPQL query from Figure 4 using the JPQL DSL.

dynamic data for the parameters in the query. Thus, when used appropriately, JPQL prevents injection attacks. However, there is not guarantee because string concatenation in queries is still possible.

Even though the JPQL string embedding avoids some of the problems the XML string embedding has, it is still problematic: Queries are not parsed and thus may contain syntax errors; query parameters are dynamically resolved, can be misspelled or forgotten; a query may illegally refer to a tuple variable not bound within the **FROM** clause; there is no editor support for queries; string concatenation is necessary to break long lines.

To address these problems, we implemented language support for JPQL as a DSL in SugarJ. A reengineered version of the previous query is shown in Figure 5. The reengineered query is statically syntax checked and does not require string concatenation to break lines. Instead of indirectly injecting dynamic data into a query, parameters (colon-prefixed identifiers) in our DSL refer to Java variables directly. Hence, a programmer needs to manage fewer namespaces and cannot forget calling setParameter. Our DSL desugars the reengineered query into the original one and generates all setParameter calls to relate the query namespace to the Java namespace.

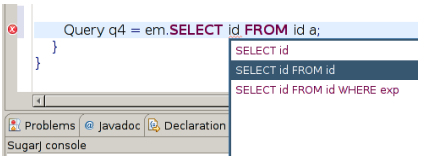

Fig. 6. Content completion for JPQL in Eclipse.

To guarantee that queries do not refer to unbound tuple variables, we implemented a domain-specific static analysis. It traverses a query and checks that every variable in the query is bound within the query's **FROM** clause. Since SugarJ executes the analysis before desugaring at compile time, we statically ensure that no unbound variables can occur at runtime for reengineered queries and we provide domain-specific error messages in case of the developer made a mistake. Furthermore, the JPQL DSL includes editor support in the form of syntax highlighting, code folding and JPQL-specific code completion, as illustrated in Figure 6.

The BNF Meta-DSL JPQL has many features and therefore its definition is rather involved. For example, a grammar provided by Oracle as part of the documentation of the JavaEE consists of 217 lines.[4] Unfortunately, Oracle employed a different grammar formalism than the one used in SugarJ. Therefore, we cannot directly reuse their grammar. However, DSLs in SugarJ are self-applicable, that is, a programmer can implement a DSL for writing other DSLs. We call this kind of DSL a *meta-DSL*. In particular, the dialect of BNF used by Oracle can be implemented as a meta-DSL in SugarJ, which enables us to reuse Orcale's grammar for the JPQL DSL.

Technically, the BNF meta-DSL is implemented as syntactic sugar on top of SugarJ's standard grammar formalism SDF. Accordingly, a BNF grammar desugars into an SDF grammar. It is even possible to mix BNF productions and SDF productions within a single library, which we have done to integrate JPQL into Java and Java variables as parameters into JPQL. The BNF meta-DSL can be reused with only minor changes in other contexts where BNF and its extensions are used for describing languages. For example, using a similar meta-DSL it would be possible to reuse the host of available ANTLR grammars.

We believe that self-applicability is particularly useful in the context of maintaining legacy applications, where it is more likely that a language description already exists in some form, for example, as documentation. Our embedding of BNF into SugarJ to reuse Oracle's JPQL grammar gives some evidence that a self-applicable DSL mechanism is not only theoretically desirable but indeed useful in practice.

Fourth Iteration: XML Schema After implementing and using the DSLs described above, we returned to the XML DSL described in Section 3 and added support

[4] `http://docs.oracle.com/javaee/5/tutorial/doc/bnbuf.html`

```
import xml.schema.XmlSchema;

public xmlschema FileUploadResponseSchema {
  <xsd:schema targetNamespace="jpsfur">
  <xsd:element name="response" type="FileUploadResponse"/>
  <xsd:complexType name="FileUploadResponse">
    <xsd:sequence>
      <xsd:element name="message" type="string" />
[...]
```

Fig. 7. Excerpt of the XML Schema definition for file upload responses.

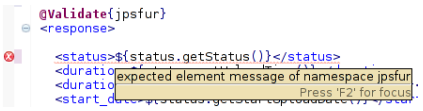

Fig. 8. Element <response> is missing its first child <message>.

for XML Schema validation. XML Schema allows programmers to specify the structure required from XML documents. We have built language support for XML Schema in SugarJ that performs XML Schema validation at compile time as a domain-specific analysis.

For example, Figure 7 shows an XML schema for file upload responses as they occur in the Java Pet Store. A programmer activates XML Schema validation for an XML document by annotating the document with **@Validate**{namespace}, where an XML schema for namespace must have been locally imported. In addition to the standard XML well-formedness checks, XML schema validation guarantees the presence or absence of tags and attributes and thus protects against incomplete data and misspelling, as seen in Figure 8. Static validation of schemas is particularly valuable if the serialisation format is to be changed. After changing the schema accordingly, compile-time error messages will point the maintainer to code that still needs to be adapted to the new format.

We defined three schemas by reverse engineering the XML documents that are actually used in the Java Pet Store. Thanks to these schemas we discovered several inconsistencies regarding XML documents in the Java Pet Store. First, the handling of composed words in tags is inconsistent: The XML response to a file upload contains both camelCase and under_score element names. Second, the XML encoding for categories contains redundant information, as seen below.

```
@Validate{jpsc}
<category>
  <id>${c.getCategoryID()}</id>
  <cat-id>${c.getCategoryID()}</cat-id>
  [...]
```

The elements id and cat-id always contain the same value. Third, there is an inconsistency between two instances of the XML representation of items, either using an element prod-id or product-id. The Java Pet Store front-end does not seem to use the generated XML documents and instead uses a JSON representation of essentially the same data. We believe these inconsistencies are previously undiscovered bugs in the Java Pet Store.

DSL	Usage in Java Pet Store	New DSL implementation code	Reused code
Accessors	avoid 506 lines of boilerplate in 13 classes	65 LoC	0 LoC
XML	check 59 lines in 7 XML documents	35 LoC	160 LoC
XML Schema	validate 51 lines in 5 XML documents using 3 schemas	20 LoC	713 LoC
JPQL	check 29 lines in 14 JPQL queries	101 LoC	140 LoC
BNF	reuse parts of the JPQL grammar	131 LoC	0 LoC

DSL	Usage in Eclipse	New DSL implementation code	Reused code
Accessors	avoid 86 lines of boilerplate in 3 classes	3 LoC	62 LoC
XML	check 449 lines in 56 XML documents	2 LoC	192 LoC

Fig. 9. Reengineering results and DSL implementation effort.

4.2 Eclipse

With the Eclipse case study, our goal is not to show new and interesting DSLs for IDE development. Rather, we aim to answer the question whether our approach of incremental introduction of DSLs and incremental adaption of code scales to very large code bases. We chose Eclipse because of the availability of its source code, its stability in terms of the plugin API combined with active development, and most importantly its size. According to a comparison of Eclipse's and Netbeans' code sizes in 2011 [15], Eclipse comprises 10 million lines of source code and is organised into just under 500 top-level folders which roughly equate to subprojects.

For this case study we checked out an arbitrary selection of 194 top-level folders from Eclipse's CVS. Out of these 194, we chose two top-level folders, namely org.eclipse.core.variables and org.eclipse.jdt.core.tests.model, for reengineering using the Accessors and XML language extensions, respectively. Together, these two folders contain 523342 lines of code in Java files.

Specifically for this case study, we extended the SugarJ compiler to support using a mix of Java and SugarJ source files. Previously, it would only accept SugarJ files. This was not a problem in the Java Pet Store, since every Java file is also a valid SugarJ file, except for the file extension. Nevertheless, renaming all source files of a project is contrary to our goal of incremental introduction.

We reused the existing DSL implementation code almost unchanged. The XML library was missing syntax rules for XML comments in its grammar, which required two new lines of SDF code. In Eclipse, field names are by convention prefixed with an f. We adapted the Accessors library's desugaring transformation to respect this convention.

4.3 Results

We reflect on the goals and expectations described at the beginning of this section. In summary, we expected easy identification of problem domains, need for language composition, and reuse of language libraries.

We successfully used SugarJ's syntactic-sugar based DSLs to improve the code quality of the Java Pet Store considerably. In Figure 9, we show an overview of

the extent of code affected by our reengineering efforts. The main purpose of the Accessors DSL was eliminating boilerplate code and its application exceeded our expectations. It saves almost 10% of the Java back-end code of the Java Pet Store. The XML and JPQL DSLs improve static safety with domain-specific analyses, readability with domain-specific syntax and editing experience with domain-specific editor support. Their application sometimes increases and sometimes reduces code size. We attribute increases to easier line breaks for more natural code formatting in the respective DSL and reductions to more concise integration of dynamic data into static DSL code. In the table we report lines of reengineered code. By manual inspection of the Java Pet Store's source code, we found ample opportunity for improvement with DSLs. Besides the DSLs we implemented, there are further areas that would benefit from language support.

During this case study we often switched between implementing DSLs and adapting code to use them. We also did not always adapt all code at once. This shows that incrementality works as desired in both dimensions: making new DSLs available and adapting parts of the code base to use them. In the reengineered Java Pet Store, there is one file that uses two DSLs at once: the JPQL and Accessors DSLs. These DSLs compose without conflict. This confirms our expectation that language composability is needed in practice and that different DSLs rarely interact unintentionally.

The implementation effort for new DSLs was reduced by reusing existing code. Figure 9 lists the lines of new language-library-implementation code that were written as part of this case study and the amount of code that was reused from previous work. SDF's declarative nature makes new syntax definition easy. For example, there is no need to know details about parsing algorithms to avoid left-recursive productions. We believe that the focus on syntactic sugar especially helps reducing the complexity of implementing DSLs. All new desugaring transformations employed in this case study are straightforward. Nonetheless, reuseability is essential in reducing the costs of DSL implementation. The XML DSL reuses the previously existing XML syntax. The XML Schema DSL is almost entirely reused from previous work [11] since it only operates on the abstract XML syntax. The XML schemas themselves are implemented in 42 lines of code on average. All DSLs implemented in this case study are immediately reusable for future reengineering efforts.

With 10 million lines of source code, Eclipse is a huge project. Any process for improving its maintainability has to be incremental, because programmers cannot be expected to change all of Eclipse's code at once. For this case study, we introduced two language extensions in two different parts of the code base. At this point, only a small part of Eclipse has been reengineered to use these DSLs. There are more opportunities to use both the XML and Accessors DSLs, and others. Nevertheless, the whole project is fully functional, because the nature of syntactic sugar makes changes necessary only local to its point of use. Thus, the Eclipse case study shows that our process is incrementally applicable and therefore scales to large code bases.

```
String.format(
  Alt( "< ? ? > ? </ ? >",
       " \"?\" ",
       "%s",
       ""),
  new Object[] { ?* }
)
```

(a) Pattern for XML.

```
Alt(
  ?,
  ?.createQuery("?"),
  ?.createQuery("?")
    .setParameter("?", id:?)
)
```

(b) Pattern for JPQL.

```
public void id:{set ? ?*} (? id:?) {
  this.id:? = id:?;
}
public Alt(Boolean,?)
Alt(id:{is ? ?*}, id:{get ? ?*}) () {
  return id:?;
}
```

(c) Pattern for field-accessors.

Fig. 10. Automatically extracted patterns for code that can be refactored to use a DSL.

5 Automatically Locating Code for DSL Usage

Finding existing code that can be refactored to use a DSL is not always easy, especially when working with a large code base. To assist maintainers, we developed a tool called *sweet tooth*[5] that takes the definition of a SugarJ DSL and a Java source file as input, and computes a ranked list of source locations at which the DSL could be used.

Sweet tooth first analyses the DSL definition to derive a syntax-tree pattern for the generated code, and then matches this pattern against a Java source file. We illustrate the patterns derived for the DSLs of our case study in Figure 10 (manually transcribed to use concrete Java syntax instead of abstract syntax trees). The question mark `?` denotes an unknown subtree, the symbol `?*` denotes an unknown list of subtrees. The form `Alt(x,...,x)` denotes alternatives in the pattern. The annotation `id:` denotes that the following tree is an identifier.

For example, for the XML DSL (see definition in Figure 2), sweet tooth analysed the recursive-descent XML pretty printer `xml-to-string` to derive a list of alternative strings that can be produced. The generated code of JPQL calls the `createQuery` method of a pre-existing entity-manager object on which the method `setParameter` is called if there is at least one parameter in the JPQL query. Compare this pattern to actual JPQL code from the Java Pet Store in Figure 4. For field-accessors, the desugaring generates setter and getter functions. The name of the setter function `id:{set ? ?*}` is composed of the string `set` followed by at least one character `?` (for which the desugaring ensures it is upper case), followed by any number of characters `?*`. The generated getter function is similar, but, depending on the type of the field, the method gets a different name using either the prefix `is` or `get`.

Technically, sweet tooth derives these patterns by extracting the transformation from a SugarJ library, normalising the transformation to a core transformation language [27] for easier analysis, and performing *abstract interpretation* of the core transformation. In our abstract interpreter, we directly represent abstract values as patterns, which thus are the result of abstract interpretation. The abstract interpreter of sweet tooth supports transformations that use conditional constructs (which lead to alternative patterns) and recursion (which is truncated after few steps). Importantly, the pattern derived by sweet tooth is

[5] Source code available online: `http://github.com/seba--/sweet-tooth`

complete in the sense that it captures all programs that can possibly be generated by the transformation. However, sweet tooth is currently limited since we only reimplemented abstract versions of a part of the standard library of core Stratego.

Finally, sweet tooth can match the derived patterns against a concrete Java program. If multiple alternatives of a pattern match the same source location, sweet tooth ranks the matches according to the specificity of the pattern. That is, the more concrete syntax-tree nodes or string snippets a pattern contains, the higher the score of this pattern. A match of a pattern that does not contain any unknown subtrees (e.g., System.out.println()) has score 1, whereas a match of a pattern without any concrete syntax-tree node or string snippet (e.g., Alt(?, ?*)) has score 0. This way a maintainer can efficiently detect those source locations that would benefit most from using the DSL. We have successfully applied sweet tooth to locate code applicable to our DSLs for JPQL and field-accessors.

6 Discussion

The choice of the case to study is an important aspect of a case study with respect to generalisability [13]. With a varied selection of implemented DSLs and the Java Pet Store's status as a research object and reference application, we are confident that the results presented here can be generalised to many other existing applications.

The Accessors DSL shows how simple language extensions can address the specific needs of an application. Similar extensions are imaginable for BigDecimal support in banking applications or parallel looping constructs [1]. In our case, conciseness was the most obvious benefit. In general, another important benefit is localising design decisions, which makes them easy to change and reason about.

The XML and JPQL DSLs improve code quality: Domain-specific syntax reduces visual boilerplate and improves readability; domain-specific static analysis helps to avoid errors; domain-specific semantics isolate code patterns and design decisions; domain-specific editor support aids editing and understanding code.

Domain-specific semantics are of particular interest with respect to safety. For example, Java's semantics prevent access to arbitrary program memory by using array indices that exceed an array's length. This language feature makes Java programs immune to one source of exploits that C programs are frequently vulnerable to because a programmer forgot to restrict access themselves. Sugar libraries provide the tools for programmers to enforce similar restrictions in DSLs. For example, the JPQL DSL prevents injection attacks because queries are proper syntactic entities and query parameters are inserted by the sanitising setParameter method instead of string concatenation. Using sweet tooth, maintainers can find all code that not yet uses the securer DSL and thereby enforce security across the entire code base.

A common criticism of domain-specific languages is that they are hard to design and implement [23]. Our case study cannot confirm this. Since we rely on syntactic sugar, design and implementation of simple DSLs is often reduced to recognising a pattern in existing code, extracting a skeleton of static Java

code and filling it with the variable parts. Concrete Java syntax [26] makes this particularly easy. Moreover, most transformations will be easy because the domain semantics are most likely implemented in a traditional class library already, like in the JPQL example.

Our case study demonstrates that it is possible to incrementally introduce DSLs into existing Java applications using SugarJ. New DSLs support can be added at any time and code can be adapted to use it at opportune times. We hope that in the future, adding DSLs and reengineering code to use them will be as common-place as more traditional refactorings are today. This calls for research into DSL-specific code smells and means to guide maintainers in making decisions what kinds of problems are best addressed using DSLs.

To this end, we developed sweet tooth, which identifies code that can be refactored to use a DSL. While our tool already is helpful in locating applicability of DSLs, there are two immediate avenues for improving sweet tooth. First, sweet tooth should not only locate code for DSL usage, but also propose a refactored DSL program. Probably, we won't be able to retain completeness for this kind of automatic program transformation, but have to apply heuristics. Second, when testing DSL applicability, sweet tooth currently applies syntactic matching of the pattern. Often, this is insufficient because semantically equivalent programs written in a different style are not matched. To also match alternative representations, we plan to extend sweet tooth so that it applies semantic matching via equational reasoning when trying to match a program against a pattern.

7 Related Work

Ward and Fowler independently coined the term language-oriented programming for a software design that focuses on DSLs [29,14]. In particular, Ward argues that DSLs improve the maintainability of a software system, mainly due to a reduction in code size. However, like any other work on domain-specific languages that we are aware of, Ward only addresses the design of newly created applications, that is, the use of DSLs must be anticipated from the start. In contrast, we demonstrate the incremental introduction of DSLs into existing applications.

Bennet and Rajlich list language abstraction as one important research direction in their roadmap for software maintenance and evolution [3]. They mention legacy applications as a problem and express some concern about whether software reengineering is a feasible solution. They base this concern partly on Sneed's work [20], which quantifies the cost of reengineering software.

Bianchi et. al. propose an incremental process for reengineering software and argue that it avoids the problems of previous non-incremental approaches [4]. However, their main focus is data migration in the context of a legacy COBOL application. It is not clear how to apply their process to introducing DSLs.

In this work, we employed SugarJ [11,10] to implement and apply DSLs for the following reasons. First, we target existing Java applications and SugarJ is based on Java. Incorporating language based abstractions into applications written in Haskell, Ruby, Scheme, Dylan, C++, and others is possible via em-

bedded DSLs [17,16,5]. Unfortunately, Java has very restricted syntactic options, no metaprogramming and a rather unexpressive type system which makes this approach unsuited for our case. Second, SugarJ implements language extensions as libraries. In contrast to extensible compilers [7,19,24] and language workbenches [18,12] the focus on libraries enables easy extension and modular reasoning about source code. And finally, we are familiar with SugarJ, its strengths and weaknesses, and as our case study shows its support for DSL development is sufficient. A detailed comparison can be found in our previous work [9,8].

8 Conclusion and Future Work

We explored how DSLs can be incrementally introduced into legacy applications to improve maintainability: reduce boilerplate, improve readability, increase static safety through domain-specific analyses, and improve navigation and writing through domain-specific editor support. In this paper, we focused on the technical feasibility of incrementally introducing DSLs into an existing code base without requiring a full rewrite. Our solution is based on introducing high-level language abstractions via semantically transparent syntactic sugar that is modularly activated by library imports.

In future work, we want to explore tool support for introducing DSLs into legacy applications. In particular, we want to answer the following questions: Can we guarantee that a DSL can be introduced into an application conflict-free? Based on sweet tooth, can we reliably detect all application sites for a DSL in a code base? Once we detect a potential application site of a DSL, can we provide an automatic refactoring of the legacy code into code that uses the DSL?

Acknowledgements. We would like to thank Paolo G. Giarrusso, Christian Kästner, and the anonymous reviewers for valuable feedback. This work is supported in part by the European Research Council, grant No. 203099.

References

1. J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *OOPSLA*, pages 31–42. ACM, 2001.
2. D. Batory, C. Johnson, B. MacDonald, and D. von Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. *TOSEM*, 11(2):191–214, 2002.
3. K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: A roadmap. In *FOSE*, pages 73–87. ACM, 2000.
4. A. Bianchi, D. Caivano, V. Marengo, and G. Visaggio. Iterative reengineering of legacy systems. *Transactions on Software Engineering (TSE)*, 29(3):225–241, 2003.
5. C. Brabrand and M. I. Schwartzbach. Growing languages with metamorphic syntax macros. In *PEPM*, pages 31–40. ACM, 2002.
6. S. Dmitriev. Language oriented programming: The next programming paradigm, 2004.
7. T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *OOPSLA*, pages 1–18. ACM, 2007.

8. S. Erdweg. *Extensible Languages for Flexible and Principled Domain Abstraction*. PhD thesis, Philipps-Universiät Marburg, 2013.
9. S. Erdweg, P. G. Giarrusso, and T. Rendel. Language composition untangled. In *LDTA*, pages 7:1–7:8. ACM, 2012.
10. S. Erdweg, L. C. L. Kats, T. Rendel, C. Kästner, K. Ostermann, and E. Visser. Growing a language environment with editor libraries. In *GPCE*, pages 167–176. ACM, 2011.
11. S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: Library-based syntactic language extensibility. In *OOPSLA*, pages 391–406. ACM, 2011.
12. S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning. The state of the art in language workbenches. In *SLE*, 2013. to appear.
13. B. Flyvbjerg. Five misunderstandings about case-study research. *Qualitative Inquiry*, pages 219–245, 2006.
14. M. Fowler. Language workbenches: The killer-app for domain specific languages? Available at `http://martinfowler.com/articles/languageWorkbench.html`, 2005.
15. D. M. Germán and J. Davies. Apples vs. oranges?: An exploration of the challenges of comparing the source code of two software systems. In *MSR*, pages 246–249. IEEE, 2011.
16. J. Gil and K. Lenz. Simple and safe SQL queries with C++ templates. *Science of Computer Programming*, 75(7):573–595, 2010.
17. P. Hudak. Modular domain specific languages and tools. In *Proceedings of International Conference on Software Reuse (ICSR)*, pages 134–142. IEEE, 1998.
18. L. C. L. Kats and E. Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *OOPSLA*, pages 444–463. ACM, 2010.
19. N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *CC*, volume 2622 of *LNCS*, pages 138–152. Springer, 2003.
20. H. Sneed. Planning the reengineering of legacy systems. *IEEE Software*, 12(1):24–34, 1995.
21. G. L. Steele, Jr. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, 1999.
22. Sun Microsystems. Java Pet Store, 2002. Available at `http://www.oracle.com/technetwork/java/index-136650.html`, accessed Nov. 14, 2012.
23. A. van Deursen and P. Klint. Little languages: Little maintenance? *Software Maintenance*, 10(2):75–92, 1998.
24. E. Van Wyk, L. Krishnan, D. Bodin, and A. Schwerdfeger. Attribute grammar-based language extensions for Java. In *ECOOP*, volume 4609 of *LNCS*, pages 575–599. Springer, 2007.
25. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.
26. E. Visser. Meta-programming with concrete object syntax. In *GPCE*, volume 2487 of *LNCS*, pages 299–315. Springer, 2002.
27. E. Visser and Z.-e.-A. Benaissa. A core language for rewriting. *Electronic Notes in Theoretical Computer Science*, 15:422–441, 1998.
28. E. Visser, Z.-E.-A. Benaissa, and A. P. Tolmach. Building program optimizers with rewriting strategies. In *ICFP*, pages 13–26. ACM, 1998.
29. M. P. Ward. Language-oriented programming. *Software – Concepts and Tools*, 15:147–161, 1995.