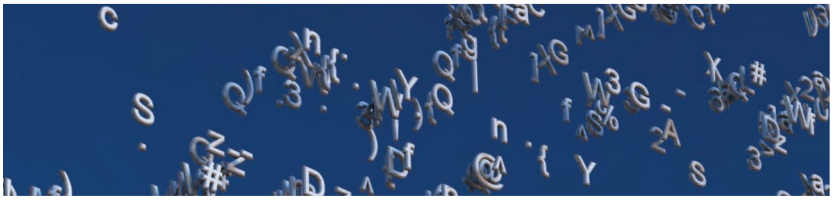
Evolution of Software Systems with Extensible Languages and DSLs

Sebastian Erdweg, Technische Universität Darmstadt, Germany

Stefan Fehrenbach and Klaus Ostermann, University of Marburg, Germany

// The extensible programming language SugarJ offers a process for gradually integrating domain-specific languages into existing software systems, thereby improving system maintenance. //



SOFTWARE SYSTEMS CONTINUOUSLY change and grow in complexity.[1] It comes as little surprise, then, that most development time and money isn't spent on the initial design and implementation of new software systems but on maintaining existing ones.[2,3]

Domain-specific languages (DSLs) offer a way to reduce maintenance costs. A DSL is a programming language specifically designed to support an application by providing domain concepts as language constructs. This enables programmers to map knowledge into source code and vice versa, which simplifies the creation, comprehension, and maintenance of domain-specific programs.[4–7]

The vast majority of existing software systems don't use DSLs. Instead, the long-standing success of C, C++, and Java has led to large procedural and object-oriented systems that don't benefit from the maintenance advantages that DSLs provide. We propose a new direction for programming language research—namely, to develop mechanisms and tools that support integrating the powerful DSL technologies we're already developing into existing software systems whose maintenance is so important.

In this article, we describe how the extensible programming language SugarJ supports the gradual evolution of existing software systems to apply DSLs and their maintenance advantages, we report on our experience of evolving and improving two existing software systems written in Java (Java Pet Store and Eclipse) by applying three DSLs, and we outline a research roadmap for developing tools that support maintainers in introducing DSLs into existing software systems.

Maintenance Advantages of DSLs

Just like regular programming languages, DSLs can feature their own syntax, semantics, static analyses, and editor support. Each of these language features has practical benefits when it comes to software maintenance.

As a running example, consider the following JPQL query that finds all items of category catID in a certain address range:

```
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
```


0740-7459/14/$31.00 © 2014 IEEE

```
BETWEEN : fromLong AND :toLong
AND i.disabled = 0
ORDER BY i.name
```

JPQL is part of the Java Persistence API and extends SQL with field selection via dot notation and named parameters prefixed by a colon. We've extracted the shown query from Java Pet Store, the reference application for building Ajax Web applications with Java. The query joins tables `Item` and `Product` based on the `productID` and selects all items that match the category parameter `catID` and required address range. The query returns the items ordered by their name.

The query is fairly readable and declaratively specifies the desired items and their ordering. This is due to the DSL syntax, which yields a smaller representational gap[8] and lets developers directly translate their domain understanding into code. In contrast, an encoding of the query as method calls or as a large concatenation of string fragments is less readable and requires programmers to translate between the domain concept and its encoding. In addition, the parser of the DSL guarantees that queries are syntactically well-formed at compile time, whereas method calls and concatenated strings mask DSL syntax errors until the program is run.

The DSL semantics gives meaning to the syntax and abstracts over the recurring patterns found in a domain concept's encoding, such as the application of string concatenation or calling conventions for methods. This has three major benefits. First, the DSL semantics eliminates boilerplate and promotes the "don't repeat yourself" mantra. Second, it enforces properties that otherwise would rely on user discipline. For example, the JPQL semantics ensures the proper escaping of Java values that are injected into the query, such as `:catID` or `fromLat`. Third, a DSL specifies the semantics of domain concepts once and for all. Changes to the behavior of a domain concept are local to the DSL definition and separate from DSL programs, a separation of concerns that improves source code modularity and maintainability.

A DSL can augment syntactic checks with static DSL analyses that detect violations of domain invariants at compile time. For example, in the JPQL query, we can statically ensure that references to the tuple variables `i` and `p` are resolvable and that, according to the database schema, the accessed columns exist. In contrast, applications that use JPQL through string encoding or method APIs run the risk of schema violations at run time.

To communicate domain knowledge to users, the DSL implementation can be augmented with DSL editor support. This support can range from syntax highlighting to specific outline views and code completion. For example, for JPQL, we can highlight keywords, named parameters, and column selections in different colors; we can provide code completion for the actual columns of a relation; or we can propose code templates such as a whole-join clause. Such editor support can vastly improve the editing experience for developers.

DSLs as Language Extensions

The most flexible way to realize a DSL is as a stand-alone language with its own parser, type checker, compiler, and integrated development environment (IDE). Such external DSLs offer maintenance advantages, but it's difficult to integrate programs written in external DSLs with parts of the software system written in other languages. This is important because DSLs typically focus on relatively small, specific domains—for example, JPQL by itself is insufficient for realizing a complete desktop application.

An alternative option is internal DSLs, which are realized by encoding domain concepts with standard language constructs in a general-purpose language. For example, the JDOM Java library is an internal DSL for describing XML documents where the domain concepts "element" and "attribute" appear as Java classes. This allows internal DSLs to readily integrate with existing code in the general-purpose language, such as standard libraries or other application code. On the downside, internal DSLs miss out on most DSL maintenance advantages because the general-purpose language limits the DSL syntax, static analyses, and editor support.

> Domain-specific syntax lets developers directly translate their domain understanding into code.

In previous work, we designed and implemented a Java-based extensible programming language and IDE called SugarJ.[9–11] Using SugarJ,

```
public extension JPQL {
  bnf syntax
    JavaExp ::= QCreate
    QCreate ::= JavaExp "." Query
    Query ::= SelectStmt | UpdateStmt | DeleteStmt
    SelectStmt ::= SelectCl FromCl [WhereCl] [GroupbyCl] [HavingCl] [OrderbyCl]
    SelectCl ::= "SELECT" ["DISTINCT"] {SelectExp ","}+
    ...
desugarings compileQuery
rules
  compileQuery  : QCreate(em, q) -> result
  where query   := |[ ~em.createQuery(~<query-to-string> q) ]|;
        vars    := <collect-all(?NamedInputParameter(<id>))> q;
        result  := <foldr(!query, set-param)> vars
  set-param : (p,e) -> |[ ~e.setParameter(~<as-string> p, ~<as-varref> p) ]|
  query-to-string : ... -> ...
}
```

FIGURE 1. An excerpt of the SugarJ extension for JPQL.

it's possible to combine the flexibility of external DSLs and the integration support of internal DSLs. SugarJ programmers can extend Java's syntax, static analysis, and editor support without changing the compiler or IDE. Instead, a SugarJ programmer can define an extension in a Java-like library that translates extended syntax to base syntax via *desugaring*, which translates extended syntax to base syntax. Figure 1 shows an excerpt of the SugarJ extension for JPQL.

The syntax declaration specifies that we extend Java's expressions by nonterminal **QCreate**, which expects a JPQL entity manager (as a Java expression) followed by a dot terminal "." and a query statement. The desugarings declaration instructs SugarJ to automatically apply the tree transformation **compileQuery**. The transformation matches **QCreate** AST nodes and produces a plain Java expression that calls **createQuery** on the entity manager and sets the Java parameters occurring in the query. Further examples and documentation appear elsewhere.[9,12]

To activate an extension in a Java source file, we simply import the library defining the extension; there's no need to configure the IDE or change the build process. The SugarJ compiler detects import statements that refer to extensions and activates them in the rest of the file by adapting the parser, desugaring, analyzer, and editor. Figure 2 illustrates the usage of a Java extension for XML in SugarJ.

SugarJ extensions ultimately desugar all extension code to plain Java code. This enables code that uses one or multiple extensions to interact with code that uses other extensions or no extensions at all. For example, the JPQL query in the previous section desugars into a method call that submits the query to the entity manager of the Java persistency API, but programmers can submit queries manually without using the JPQL extension. The integration support of DSLs that are implemented as language extensions is an important premise for the evolution of existing software systems to DSLs.

Evolution to DSLs

The evolution of a large software system is an inherently incremental process where code is gradually improved and adapted to new requirements. To promote the introduction of DSLs and their maintenance benefits during software evolution, we propose a simple incremental process:

1. Identify the problematic domain in existing code.
2. Design new language features to circumvent the problematic code.
3. Implement language features as a language extension.
4. Gradually adapt the existing code to use language extensions when useful.

It's important to note that our process never requires a full rewrite of the software system; the system remains executable at all times. Specifically, it's possible to update small portions of code when the moment is opportune. For example, there might be a bug report describing an incorrect result of a database query. To address this bug, a maintainer can first adapt the query to use JPQL to gain a more readable query without boilerplate. Potentially, a validation of the query against the database schema already reveals the bug statically. Otherwise, JPQL editor features assist the maintainer in writing a corrected version of the query.

SugarJ provides three features

that enable such incremental evolution to DSLs. First, we implement DSLs as language extensions that describe *semantically transparent syntactic sugar*. This means that code behaves exactly the same before and after adaption to a DSL and, in particular, interaction with non-adapted code isn't affected. The end result is that adapted code becomes more maintainable.

Second, we organize language extensions as libraries and activate them locally and explicitly through import statements. This means that a maintainer can selectively activate DSLs per source file. Maintainers obtain the important invariant that unchanged files aren't affected by language extensions and thus don't require attention. This is especially important for large software systems.

Third, SugarJ extensions are composable: Programmers can use the syntax, static analyses, and editor support of different extensions simultaneously, even in an interleaved fashion.[13] For example, JPQL queries and XML documents contain Java expressions, which other extensions such as anonymous functions or tuples can further extend. The SugarJ compiler composes independent extensions automatically in most practical scenarios.[13]

The SugarJ compiler and IDE are available as open source (http://sugarj.org). To support extensibility for languages other than Java, we also built a compiler framework for syntactically extensible languages[14] and defined extensible variants of JavaScript, Prolog, Haskell, and Scala. We plan to build similar variants of languages for C and C++, which are widely used in legacy software systems.

```
import xml.Sugar;
import xml.schema.FileUploadResponseSchema;

public void postProcessingMethod(FileUploadStatus status, ...) {
    ...
 response.setContentType("text/xml;charset=UTF-8");
 response.setDateHeader("Expires", 1);
 String xml =
   @Validate{FileUploadResponseSchema}
   <response>
     <message>${responseMessage}</message>
     <status>${status.getStatus()}</status>
     <duration>${status.getUploadTime()}</duration>
     <duration_string>${status.getUploadTimeString()}</duration_string>
     <start_date>${status.getStartUploadDate()}</start_date>
     <end_date>${status.getEndUploadDate()}</end_date>
     <upload_size>${status.getTotalUploadSize()}</upload_size>
     <thumbnail>${thumbPath}</thumbnail>
     <itemId>${itemId}</itemId>
     <productId>${prodId}</productId>
   </response>;
 writer.write(xml);
 writer.flush();
}
```

FIGURE 2. The XML language extension enables validated XML literals where $ escapes to Java.

Case Studies

We applied our incremental process for evolution to DSLs to two existing software systems, the Java Pet Store and the Eclipse IDE. Here, we report on our experience of evolving these systems.

Java Pet Store

The Java Pet Store is the reference application for building Ajax Web applications with Java. The Pet Store consists of 40 Java source files containing 3,807 SLOC (source lines of code without comments and blank lines). Through manual inspection of the Pet Store, we identified three problematic domains in its source code. For these domains, we designed and implemented language features as extensions and used them to evolve the Pet Store's code base.

The first problematic domain we identified was a string-based encoding of JPQL queries. This string encoding requires escaping of special symbols, relies on lexical concatenation of string fragments, potentially leads to syntax and type errors at run time, and prevents editor support such as name resolution. We designed and implemented a language extension that integrates JPQL queries as a language feature into Java as shown in the JPQL example we previously referenced. We provide name resolution for tuple variables bound in a query's `FROM` clause, and we offer code coloring and code completion for common JPQL patterns.

```
import sugar.Accessors;

@Entity
@Table(name="PRODUCT")
public class Product implements Serializable {
 private String productID {set; con};
 private String categoryID, name, description, imageURL {get; set; con};

 public Product() { }

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

FIGURE 3. A language extension that avoids boilerplate accessor methods.

The second problematic domain we found was a string-based encoding of XML documents generated at run time for Ajax data exchange. Like the JPQL encoding, the XML string encoding requires escaping and lexical concatenation, and lacks any static checking. In particular, generated XML documents aren't validated against their XML schema. In previous work,[9] we designed and implemented a language extension for XML and XML Schema that supports compile-time syntax checking and validation of XML literals against a user-supplied XML schema. For the Pet Store, we adapted the desugaring transformation to generate code that's semantically equivalent to the existing string-based encoding. Figure 2 shows an excerpt of the revised code, where we use the XML and XML Schema extensions to represent and validate an Ajax message. Besides validation, we also derive code completion rules from an XML schema to guide developers in writing valid XML.

The third problematic domain we identified was the overwhelming number of accessor methods: getters and setters. While these methods are trivial in nature, their mass masks a program's core aspects. Most significantly, some classes of the Pet Store contain annotations according to the used object-relational mapping. For example, class `com.sun.javaee.blueprints.petstore.model.Product` has all standard getters and setters except for a single method, which is annotated `@Id`. The annotation marks that this method provides the primary key for the database binding. This important information is lost among the other nine accessor methods.

We designed and implemented a language extension that allows programmers to declare if there should be a getter or setter method for each field and if the field should be part of the initializing constructor. Figure 3 shows the revised `Product` class. Here, the accessor desugaring generates four getters, five setters, and one initializing constructor. While IDEs typically also support the initial generation of getter and setter methods, the generated methods persist in the source code and must be maintained. Using our extension, getter and setter methods never occur in the source code.

Table 1 summarizes our evolution of the Pet Store. For each DSL, we list the number of affected DSL objects, the number of affected source files, how the code was affected, and the size of the DSL implementation. For the latter, we distinguish the implementation code that we had to write anew from the code that we were able to reuse from previous DSL implementations.

While evolving the Pet Store, we didn't integrate the DSLs in all places possible. For example, there are still accessor methods and string-encoded XML documents left in the code. In fact, we didn't even look at all files of the Pet Store. Instead, we locally and incrementally adapted the code to use the DSLs where it seemed most beneficial. Such an incremental process might not be necessary for the comparatively small Pet Store, but local and incremental evolution greatly simplified the evolution task because we could focus on a single DSL object at a time and guarantee that the code in other files didn't accidentally break because of our refactoring.

Eclipse IDE

Eclipse is a large-scale, open source software project that comprises about 10 million lines of source code and is organized into just under 500 subprojects.[15] For this case study, we selected only two subprojects: `org.eclipse.core.variables` and `org.eclipse.jdt.core.tests.model`. They contain 16 and 216 Java source files, adding up to 286 and 586 SLOC, respectively. Because the code in these files interacts with code in many other Eclipse subprojects, it's important to retain interoperability throughout DSL

TABLE 1

Summary of the evolution of the Java Pet Store.

DSL	DSL objects	Files affected	Code affected	DSL implementation	DSL reuse
JPQL	14 queries	2 files	refactor 29 SLOC	101 SLOC	140 SLOC
XML	7 documents	3 files	refactor 59 SLOC	35 SLOC	160 SLOC
XML Schema	5 schemas	3 files	validate 59 SLOC	20 SLOC	713 SLOC
Accessors	112 methods	13 files	eliminate 336 SLOC	65 SLOC	0 SLOC

TABLE 2

Summary of the evolution of the Eclipse IDE.

DSL	DSL objects	Files affected	Code affected	DSL implementation	DSL reuse
XML	56 documents	3 files	refactor 449 SLOC	2 SLOC	193 SLOC
XML Schema	2 schemas	3 files	validate 397 SLOC	0 SLOC	733 SLOC
Accessors	9 methods	3 files	eliminate 86 SLOC	3 SLOC	62 SLOC

integration. We achieve this because our DSLs are semantically transparent syntactic sugar.

We refactored the two subprojects by integrating the DSLs for XML and accessor methods described in the case studies. Table 2 summarizes the conducted changes. We eliminated a few accessor methods and found XML documents for generating the Eclipse-specific configuration files `.project` and `.classpath`, which we refactored and validated using our DSLs.

Note that we deliberately only changed a tiny portion—namely, six—of the Eclipse source files. That is, we made local improvements to some of the source files in Eclipse without affecting Eclipse's overall functionality. The syntactic sugar requires only local changes where it's used.

Also note that we reused almost all the DSL implementation from the Java Pet Store in Eclipse; only minor adaptions were needed, thereby confirming that DSL implementations can be reused across software systems, which further reduces the investment for applying DSLs.

Tool Support for DSL Integration: A Roadmap

As our case studies confirm, extensible languages can help gradually integrate DSLs into existing software systems, independent of the size of the system at hand. Thus, extensible languages support software evolution as a way to integrate DSLs. However, according to the incremental process we described earlier, the integration of a DSL into a software system is a manual process, requiring developers to refactor code by hand to use the DSL instead of the original code. For a large software system such as Eclipse, the overall maintenance benefit of manually refactored code fragments is probably too small to be noted. To really exploit the maintenance benefits of DSLs in large-scale software systems, we need tool support that partially automates the integration of DSLs.

To this end, we offer a roadmap for the development of three tools that assist software maintainers in integrating DSLs into large-scale software systems. We expect that DSLs for problematic domains have already been implemented as language extensions, and we don't target fully automatic DSL integration. Instead, we propose tools that analyze a DSL definition as well as the code base, to guide maintainers and help them apply a DSL, thus maximizing the DSL's coverage and maintenance benefits.

Identifying DSL Application Sites

Large code bases address many different concerns. Because a DSL is specific to a single domain or concern, we expect that it isn't applicable for most code. In fact, we can expect that the DSL is only applicable to a relatively small part of a small number of files, compared to the overall software system. After all, DSLs draw their power from their specificity.

So how is a maintainer supposed to identify the parts of a software system where a DSL will be most beneficial? The first tool we describe will help answer this question: It analyzes a DSL's desugaring transformation, which translates the

extended syntax into the base language's constructs. From this transformation, the tool derives a pattern representing those base programs that the transformation can generate. Thus, code that matches the derived pattern is likely replaceable by the DSL code. As a simple example, this tool would analyze the transformation definition of the JPQL DSL and determine that desugared code contains method calls to `createQuery` and `setParameter`.

> So how is a maintainer supposed to identify the parts of a software system where a DSL will be most beneficial?

As a second step, our tool matches the derived pattern against the whole code base to identify the concrete source-file locations at which the analyzed DSL is likely to be applicable. Depending on the derived pattern and the code base, this matching could yield a very long list of potential application sites. To assist a maintainer in selecting relevant application sites, our tool will sort the list so that sites rank higher if a DSL application has higher impact—that is, if more code can be replaced to use the DSL.

Because hand-written redundant code is likely to be less consistent and schematic than generated code, the matching process should be fuzzy and detect parts of the code that are only similar (but not identical) to code that could be generated by a desugaring.

Refactoring to DSL Code

After identifying relevant application sites for a DSL, the next step for a maintainer is to replace the existing code with the DSL code. Not only is it tedious to manually refactor existing code to use a DSL, but such refactoring needs to preserve the original code's behavior, which can be difficult to achieve without tool support.

Our second tool will assist maintainers in applying a DSL by using an interactive refactoring dialog. The user selects a DSL and a potential application site for this DSL. The refactoring dialog will then propose ways to instantiate the DSL so that the original code's behavior is preserved. User interaction is required here for two reasons. First, alternative ways to instantiate a DSL could achieve the same behavior. Thus, the maintainer must select one of several alternatives, similar to how code recommendation systems work. Second, our tool might not be able to infer all the details of the DSL instantiation. In this case, the user must use the refactoring dialog to fill in details of a partially inferred DSL program and our tool will check that the program preserves the original behavior. To check behavior preservation, our tool will compare the original code with the desugared DSL code.

We envision that existing research on bidirectional programming[16] will be useful to automate the process of instantiating a DSL that desugars into a given program.

DSL Integration Completeness

After refactoring code to use a DSL, maintainers might change the DSL program to repair deficiencies or activate new features. However, it's often desirable to change the DSL's back end of a DSL; for example, to retarget JPQL to an in-memory database or to use another serialization format such as JSON instead of XML. Typically, DSLs support changes to the back end particularly well because the DSL implementation abstracts the details of the back end from the user. Unfortunately, in our evolutionary setting, the DSL's abstraction barrier is only partially maintained: existing code that hasn't yet been refactored to use the DSL shares implementation details with the DSL's back end. For example, if we changed JPQL to call methods of an in-memory database instead of sending a query to a dedicated database engine, string-encoded JPQL queries left over in the code base would conflict and yield errors at compile or run time, thus preventing maintainers from benefiting from the DSL abstraction.

To address this problem, our third tool will analyze the code base to validate that the DSL integration is complete—that is, that no DSL application sites are left over. There are two ways to achieve this: by using the refactoring tool and letting the programmer confirm that all remaining candidates for using the DSL are false positives, which would require the candidate list to be complete; or by characterizing completeness via an architectural constraint, such as the constraint that a specific API is never used directly (but only indirectly via the DSL). Such architectural constraints are easy to check, and corresponding architecture constraint languages could be defined and enforced within SugarJ.

If our tool can guarantee complete integration for a DSL, this means that the domain semantics are modularized in the DSL's definition. Thus, DSL behavior can be modularly changed by adapting the back end. This can be a crucial asset in the long-term maintenance of a software system, where adoption of new technologies is an important issue.

References

1. M.M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution," *Proc. IEEE*, vol. 68, no. 9, 1980, pp. 1060–1076.
2. B.P. Lientz and E.B. Swanson, *Software Maintenance Management*, Addison-Wesley, 1980.
3. S.W.L. Yip and T. Lam, "A Software Maintenance Survey," *Asia-Pacific Software Eng. Conf.* (APSEC), 1994, pp. 70–79.
4. F. Hermans, M. Pinzger, and A. van Deursen, "Domain-Specific Languages in Practice: A User Study on the Success Factors," *Proc. Conf. Model Driven Engineering Languages and Systems* (MODELS), LNCS 5795, Springer, 2009, pp. 423–437.
5. T. Kosar, M. Mernik, and J.C. Carver, "Program Comprehension of Domain-Specific and General-Purpose Languages: Comparison Using a Family of Experiments," *Empirical Software Eng.*, vol. 17, no. 3, 2012, pp. 276–304.
6. T. Kosar et al., "Comparing General-Purpose and Domain-Specific Languages: An Empirical Study," *Computer Science and Information Systems*, vol. 7, no. 2, 2010, pp. 247–264.
7. A. van Deursen and P. Klint, "Little Languages: Little Maintenance?," *Software Maintenance*, vol. 10, no. 2, 1998, pp. 75–92.
8. C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 2nd ed., Prentice Hall, 2002.
9. S. Erdweg et al., "SugarJ: Library-Based Syntactic Language Extensibility," *Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA), 2011, pp. 391–406.
10. S. Erdweg et al., "Growing a Language Environment with Editor Libraries," *Proc. Conf. Generative Programming and Component Eng.* (GPCE), 2011, pp. 167–176.
11. S. Erdweg, "Extensible Languages for Flexible and Principled Domain Abstraction," PhD thesis, Dept. Mathematics and Computer Science, Philipps-Universität Marburg, 2013.
12. S. Fehrenbach, S. Erdweg, and K. Ostermann, "Software Evolution to Domain-Specific Languages," *Proc. Conf. Software Language Engineering* (SLE), LNCS 8225, Springer, 2013, pp. 96–116.
13. S. Erdweg, P.G. Giarrusso, and T. Rendel, "Language Composition Untangled," *Proc. Workshop Language Descriptions, Tools, and Applications* (LDTA), 2012, pp. 7:1–7:8.
14. S. Erdweg and F. Rieger, "A Framework for Extensible Languages," *Proc. Conf. Generative Programming and Component Eng.* (GPCE), 2013, pp. 3–12.
15. D.M. Germán and J. Davies, "Apples vs. Oranges?: An Exploration of the Challenges of Comparing the Source Code of Two Software Systems," *IEEE Mining Software Repositories* (MSR), 2011, pp. 246–249.
16. J.N. Foster et al., "Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem," *Transactions on Programming Languages and Systems* (TOPLAS), vol. 29, no. 3, 2007, pp. 17:1–17:93.

ABOUT THE AUTHORS

SEBASTIAN ERDWEG is postdoctoral researcher at Technische Universität Darmstadt, Germany, where he investigates metaprogramming, language design, and language semantics. Sebastian is the lead developer of the extensible programming language SugarJ. He received a PhD in computer science from Philipps-Universität Marburg.

STEFAN FEHRENBACH is a graduate student at the University of Marburg, Germany. His primary interest is in the practical application of programming language research, concentrating on extensible languages and domain-specific languages in legacy applications. Stefan received a BSc in computer science from Philipps-Universität Marburg.

KLAUS OSTERMANN is a faculty member in the department of mathematics and computer science at the University of Marburg, Germany. His research concentrates on programming languages and methodology. Klaus received a PhD in computer science from Technische Universität Darmstadt.